HAL
open science

(Dynamic (programming paradigms)) Performance and expressivity

Didier E Verna

To cite this version:

Didier E Verna. (Dynamic (programming paradigms)) Performance and expressivity. Software Engineering [cs.SE]. LRDE - Laboratoire de Recherche et de Développement de l'EPITA, 2020. hal-02988180

HAL Id: hal-02988180

https://hal.science/hal-02988180v1

Submitted on 4 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE D'HABILITATION À DIRIGER LES RECHERCHES
Sorbonne Université
Spécialité Sciences de l'Ingénieur

(DYNAMIC (PROGRAMMING PARADIGMS))
PERFORMANCE AND EXPRESSIVITY

Didier Verna
didier@lrde.epita.fr
Laboratoire de Recherche et Développement de l'EPITA (LRDE)
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre CEDEX

Soutenue le 10 Juillet 2020

Rapporteurs:
Robert Strandh Université de Bordeaux, France
Nicolas Neuß FAU, Erlangen-Nürnberg, Allemagne
Manuel Serrano INRIA, Sophia Antipolis, France

Examinateurs:
Marco Antoniotti Université de Milan-Bicocca, Italie
Ralf Möller Université de Lübeck, Allemagne
Gérard Assayag IRCAM, Paris, France

DOI 10.5281/zenodo.4244393

Résumé (French Abstract)

Ce rapport d'habilitation traite de travaux de recherche fondamentale et appliquée en informatique, effectués depuis 2006 au Laboratoire de Recherche et Développement de l'EPITA (LRDE). Ces travaux se situent dans le domaine des langages de programmation dynamiques, et plus particulièrement autour de leur expressivité et de leur performance.

Plutôt que de développer en profondeur un aspect particulier de ces travaux (ce qui serait redondant avec la publication académique correspondante déjà effectuée), ce rapport est conçu à la fois comme une "accroche" pour les travaux existants (dans l'espoir d'éveiller la curiosité du lecteur), et comme une "bande-annonce" du travail futur (dans l'espoir de solliciter de nouvelles collaborations).

Le corps de ce rapport est composé 8 chapitres, correspondant chacun à un projet de recherche en particulier. Ces projets sont plus ou moins indépendants les uns des autres, mais traitent tous des paradigmes de programmation rendus disponibles par le contexte dynamique. Les points abordés s'étalent sur à peu près toute la chaîne langagière : calcul sur les types, aspects syntaxiques (comme avec les "domain-specific languages"), aspects sémantiques (comme avec des paradigmes orientés-objet dynamiques de haut niveau), et aspects purement théoriques (comme avec les diagrammes de décision binaires). Pour chaque projet, un résumé du travail déjà effectué est proposé, avec mention de la publication académique correspondante. Les perspectives d'évolution sont également décrites.

Contents

Foreword **3**

1 Introduction **4**
1.1 Motivation 4
1.2 Implementation 5
1.3 Research Strategy 6

2 Compiler Performance **7**
2.1 Project Description and Current Results 7
2.2 Future Work 9

3 DSL Design and Implementation **11**
3.1 Project Description and Current Results 12
3.2 Future Work 13

4 Context-Oriented Optimization **15**
4.1 Project Description and Current Results 15
4.2 Future Work 17

5 Rational Type Expressions **19**
5.1 Project Description and Current Results 19
5.2 Future Work 21

6 Type Calculus Performance **23**
6.1 Project Description and Current Results 23
6.2 Future Work 26

7 Worst-Case Binary Decision Diagram (BDD) Study **27**
7.1 Project Description and Current Results 27
7.2 Future Work 28

8 Dynamic Object Orientation **30**
8.1 Project Description and Current Results 30
8.2 Future Work 32

9 Exception-Oriented Programming **34**
9.1 Project Description and Current Results 34
9.2 Future Work 36

10 Conclusion **38**

A Personal Bibliography **39**

B Third-Party Bibliography **41**

Acknowledgments **49**

Foreword

Starting with my Ph.D., the beginning of my career as a researcher focused on virtual reality, artificial intelligence, and cognitive science. Since then, my scientific interests have gradually shifted to programming languages, paradigms, and the study of their expressivity and performance.

This habilitation report relates only work that has been done after that shift, that is, after 2006. Perhaps it is worth mentioning that, due to my other (and regular) non-scientific activities, those 14 years correspond to roughly 9 years of equivalent full-time employment. I also wish to stress that the work related here is not mine only, but the result of collaborations with many students, including one completed Ph.D., 5 Masters level internships, and a dozen or so undergraduate collaborations.

With the rise of *gradual typing* (Siek and Taha, 2006, 2007), the distinction between statically and dynamically typed programming languages tends to blur, Racket being probably the most advanced example to date in that regard (Tobin-Hochstadt and Felleisen, 2006). Nevertheless, the vast majority of contemporary programming languages are still designed around one primary type system, and I will call a language primarily designed around a dynamic type system a *dynamic programming language* (or just *dynamic language* for short).

When I speak of dynamic programming paradigms, I mean programming paradigms for dynamic languages, that is, paradigms enabled by dynamic type systems. This concept is unrelated to the domain of *dynamic programming* (Sniedovich, 2010), which is both a mathematical and computer science method for solving potentially complicated problems. Therefore, in the expression "dynamic programming paradigms", it is the paradigms which are dynamic, not the programming. I chose to make the associativity explicit in the title precisely to resolve that ambiguity. In other words, this report is about (dynamic (programming paradigms)), as opposed to ((dynamic programming) paradigms).

Under this common banner, the work accomplished until now has tackled many different aspects of software engineering, not necessarily connected to each other, and has lead to the publication of 1 book chapter, 4 international journal papers, 24 international conference papers, and about 30 additional publications in various other forms. Had this report been about one aspect in particular, it would have been redundant with at least one of the aforementioned publications. Therefore, I rather chose to write it as an overview of both what has been done already, and what is to come. In other words, I hope that the reader will receive the material presented here as both a "teaser" for the existing bibliography, and a "trailer" for future work.

In order to clearly distinguish our own publications from the work of others, I have split the usual bibliographic chapter in two (appendices A on page 39, and B on page 41). In addition to that split, every reference to our own bibliographic material is accompanied with a marginal note such as the one you are seeing here: Verna (2020)[1].

1 Verna, D. (2020). Dynamic programming paradigms: Performance and expressivity. Habilitation Report. Sorbonne Université.

Chapter 1

Introduction

Our research activity deals mainly with dynamic programming languages and their associated paradigms, expressivity, and performance. We are particularly interested in such concepts as Domain-Specific Language (DSL), functional and object-oriented programming, extensibility, reflexivity, and meta-programming.

1.1 Motivation

From a historical perspective, the development of industrial-scale programming languages is bottom-up: abstraction grew layer after layer, on top of imperative and sequential programming, and the principles stated by John Von Neumann ruling the hardware. This bias may be explained in two different ways. Theoretically speaking, it is simpler to build on top of what already exists. Hence, the first relatively high-level programming languages, *e.g.*, Fortran (1977) remained close to the machine. Practically speaking, very abstract languages designed in a top-down fashion, such as Lisp (McCarthy, 1960), a Fortran contemporary, were not viable solutions for the industry. The hardware or compilers (when available) failed to deliver the necessary performance, their features were too advanced to be fully understood or even just deemed interesting, or their "far out" design was simply perceived as too frightening. The industry hence followed the path of more efficient languages, at the expense of a potentially higher level of expressivity.

With the rapid growth of computing power and theoretical knowledge (particularly in the domain of compilers), some programming paradigms known as early as in the 60's eventually regained momentum. Scripting languages (re)appeared in the 90's, functional programming began to get the attention it deserved in mainstream languages only a few years ago (see for instance the addition of anonymous functions to C++11 and Java 8). Finally, the rise of the Web vastly contributed to putting dynamic languages (such as PHP and JavaScript) back at the heart of industrial preoccupations.

Today, the industry has finally acknowledged the importance of being multi-paradigm in its applications, but the technical debt often leads to the adoption of heterogeneous solutions, combining for example a static language at the core (for performance), a scripting language at the user level (for suppleness), and possibly a DSL for the non-technical end-user. The extreme diversity (and disparity) of the software components involved may have a negative impact on the final product in terms of evolution and maintainability. In this context, it becomes crucial to develop

a genuine craftsmanship in multi-paradigm software engineering, perhaps in a more unified, homogeneous, form. This concern is at the heart of our research.

1.2 Implementation

From a practical perspective, a technical challenge to pursue this goal is to integrate a maximum number of different paradigms within the same environment, in order to be able to study their interactions. As mentioned earlier, we put the emphasis on the functional and object-oriented approaches, in addition to the classical imperative and procedural ones. We also like to stress extensibility, hence an interest in DSLs, reflexivity, and meta-programming. Finally, a major concern is also to study the characteristics of the compromise between expressivity (genericity in particular) and performance, as abstraction notoriously comes at a cost.

The next step is thus to decide on the most appropriate platform for experimentation. Most of the current multi-paradigm approaches are somewhat *ad hoc* and heterogeneous for the simple reason that there is in fact not a lot of choice. Object-oriented languages such as C++ (ISO/IEC, 2017) and Java (Gosling et al., 2019) are not (or hardly) functional. Scripting languages such as Python (van Rossum, 2012) or Ruby (Thomas et al., 2009) are not particularly efficient —hence the existence of projects such as PyPy (Bolz et al., 2009), providing alternative implementations. Julia (Karpinski and Bezanson, 2019) is an interesting and more modern approach to a dynamic (yet highly optimized) language, but the support for object orientation is (intentionally) quasi-nonexistent. Scala (Odersky et al., 2010) is a very interesting language when it comes to paradigm mixture (in particular functional and object-oriented), but we find the syntax quite cumbersome, and the support for compiling to native code rather than to the Java Virtual Machine (JVM) is still embryonic.

In this landscape, the Lisp family of languages looks particularly attractive. In the end, the choice of a preferred language for experimentation doesn't need to be (and most likely never is) completely objective, as long as the produced research, on the other hand, is scientifically legitimate. Our preference goes to Common Lisp (Ansi, 1994) for various reasons. In general we are very sensitive to the "Lisp philosophy", that is, mostly the minimalism of its core and its homoiconic nature (McIlroy, 1960; Kay, 1969), which deeply relates to our aesthetic preferences (Verna, 2018a)[1]. We also like the more pragmatic approach of this dialect, whereby being practical always remains a goal, sometimes at the expense of elegance or a theoretically purer design. Finally, Common Lisp is the only Lisp dialect to have an official, industry-level standard, which is important for two reasons. Firstly, interesting research results may be directly tested and applied to already existing industrial applications using this language. Secondly, having an official standard (while remaining extensible) also means that our working environment is much more stable that the alternatives.

1 Verna, D. (2018a). Lisp, Jazz, Aikido. *The Art, Science and Engineering of Programming Journal*, 2(3).

On more concrete and objective terms, using Common Lisp for experimentation has various advantages. In terms of expressivity, it is both functional, object-oriented, imperative, and procedural. It offers different typing policies, different forms of scoping and evaluation. Clos, the Common Lisp Object System, is a very expressive class-based object-oriented layer (DeMichiel and Gabriel, 1987; Bobrow et al., 1988; Keene, 1989; Gabriel et al., 1991), and surpasses the abilities of more classical approaches. Besides, the dynamic nature of Clos allows, with relative ease, the incorporation of

experimental object-oriented paradigms such as context orientation (Hirschfeld et al., 2008), filtered (Costanza et al., 2008), and predicate dispatch (Ernst et al., 1998; Ucko, 2001). Common Lisp is also a reflexive language, offering both introspection and intercession, and for which reflexivity is both structural and behavioral (Maes, 1987; Smith, 1984): the CLOS Meta-Object Protocol (MOP) (Kiczales et al., 1991; Paepcke, 1993) and the so-called *reader macros* are tools allowing the programmer to extend or modify both the syntax and the semantics of the language itself, making it very suitable to DSL design and implementation, for instance. Its powerful programmatic macro system offers yet another meta-programming facility.

Nevertheless, the very high level of abstraction of the language doesn't cut the programmer off from lower level access (*e.g.* bit-wise manipulation, internal representation, foreign interfaces, *etc.*) or optimization (*e.g.* optional weak static typing, destructive versions of normally purely functional procedures, *etc.*). To summarize, Common Lisp is a language offering many different programming paradigms at the same time, with many different levels of abstraction, and with a very high level of extensibility. It is thus particularly well suited to our final objective.

1.3 Research Strategy

From an academic perspective, the research strategy we adopt is articulated around three axes.

1. Producing innovative research in software engineering. Here, we try to design or study the new programming paradigms that the dynamic context makes possible, we investigate the use of existing ones to new applicative contexts, or we generalize them and enrich them with new features. Finally, we also try to study the comparative merits of those paradigms in terms of expressivity and/or performance, and we analyze their interaction, notably in terms of composability and orthogonality (Hunt and Thomas, 1999).

2. Contributing to the languages community in general. While we prefer the Lisp language as our primary platform for experimentation and prototyping (for reasons that were explained in Section 1.2 on the preceding page), we try to propagate the obtained results to a larger community. In particular, we extract those results that can be generalized in a purely theoretical form, and we study their applicability to other dynamic programming languages.

3. Contributing to the Lisp community in particular. In some situations, experimenting and prototyping in Lisp can reveal both strengths and weaknesses of the language itself. In the cases where we consider the qualities of Lisp as crucial, we make a point in producing an otherwise missing academic bibliographic foundation for the language (Lisp being originally rather targeted to the industry, hence with comparatively little academic audience or recognition). In the cases where, on the other hand, room for improvement is found, we may be led to propose extensions to the language, alternative implementations for features within existing compilers, or even propose modifications to its current specification.

Chapter 2

Compiler Performance

Two Masters internships •
Two conference papers, including one best paper award •

The purpose of this project is to evaluate the performance of Lisp compilers, comparatively to languages known for their efficiency, such as C or C++. We are interested in evaluating the impact of dynamicity, or otherwise showing that, given the exact same abstraction level, the resulting performance is equivalent. The analysis of empirical results also constitutes valuable feedback for the concerned vendors.

Long after the standardization process (Ansi, 1994), and after people really started to care about performance (Gabriel, 1985; Fateman et al., 1995; Reid, 1996), Common Lisp (in fact, Lisp in general) is still widely perceived as a "slow language". Programmers looking for the utmost performance turn more naturally to languages such as C (ISO/IEC, 2011) or C++, perhaps today also Go (Google, 2018) and Rust (Klabnik and Nichols, 2018), the rationale being that statically typed languages, at the expense of some level of expressivity, are in general more efficient.

Until about 10 years ago, existing literature on the performance of Lisp was advertising merely 60% of that of equivalent C code (Neuss, 2003; Quam, 2005). Such an achievement could be seen either as satisfactory, from a dynamic point of view (Boreczky and Rowe, 1994), or as a show stopper for critical applications, performance-wise. Hence, we considered it important to correct that view, and demonstrate that in actuality, it would be possible to lose *nothing* in performance by using Lisp, the corollary being that *a lot* would be gained in expressivity.

2.1 Project Description and Current Results

This project consists of studying the comparative behavior and performance of Lisp with benchmarks on various algorithms, paradigms, and data types or structures. Over its duration, this project is split into different parts.

The first part deals with fully dedicated (or "specialized") code, that is, programs built with full knowledge of the algorithms and the data types to which they apply. The second and third part are devoted to studying the impact of genericity, through dynamic object orientation and static meta-programming, respectively. The rationale here is that adding support for genericity would at best give equal performance, but more probably entail a loss of efficiency. Consequently, if fully dedicated Lisp code is

already unsatisfactory with respect to, say, C versions, it would be useless to try and go further.

When it comes to comparing the performance of Lisp code against C or C++ versions, we need to emphasize that we are in fact comparing *compilers* as much as *languages*. Moreover, speaking of "equivalent" C, C++, or Lisp code is also somewhat inaccurate. For instance, when we write functions, we end up comparing sealed function calls in C with calls to functions that may be dynamically redefined in Lisp. When we measure the cost of object instantiation, we compare a C++ *operator* (`new`) with a Lisp *function* (`make-instance`), *etc.* All those subtle differences mean that it is actually impossible to compare exclusively either language, or compiler performance.

2.1.1 Part 1

In the first part of this project, which is completed today, we experimented with a few simple image processing algorithms written both in C and Lisp. In Verna (2006)[1], we demonstrated that, given the state of the art in Common Lisp compiler technology, most notably, open-coded arithmetics on unboxed numbers and efficient array types (Fateman et al., 1995, sec. 4, p.13), the performance of equivalent C and Lisp programs are comparable; sometimes even better with Lisp. More specifically, we demonstrated that the behavior of equivalent Lisp and C code is similar with respect to the choice of data structures and types, and also to external parameters such as hardware optimization. We further demonstrated that properly typed and optimized Lisp code runs as fast as the equivalent C code, or even faster in some cases.

1 Verna, D. (2006). Beating C in scientific computing applications. In *ELW'06, 3rd European Lisp Workshop*, Nantes, France.

2.1.2 Part 2

The second part of the project deals with dynamic genericity through CLOS (Keene, 1989), the object-oriented layer of Common Lisp. Our purpose here is to evaluate the behavior and efficiency of instantiation, slot access, and generic dispatch in general, actually splitting part 2 in three individual sub-steps. Currently, the first sub-step (part 2.1: instantiation) is complete and has been reported in Verna (2009)[2].

In this paper, studies on both C++ via GCC and Common Lisp via three different compilers were presented, along with cross-comparisons of interesting configurations. We demonstrated the following points.

2 Verna, D. (2009). CLOS efficiency: Instantiation. In *ILC'09 International Lisp Conference*, pages 76–90, MIT, Cambridge, Massachusetts, USA. ALU (Association of Lisp Users).

- When safety is privileged over speed, the behavior of instantiation is very different from one language to another. C++ is very sensitive to the inheritance hierarchy and not at all to the slot type. Lisp on the other hand, is practically immune to the inheritance hierarchy, but in the case of structures, sensitive to the slot type.
- While turning on optimization in C++ leads to a reasonable improvement, the effect is tremendous in Lisp. As soon as the class to instantiate is known or can be inferred at compile-time, the instantiation time can be divided by a factor up to one hundred in some cases, and to the point that instantiating in Lisp becomes actually *faster* than in C++.

2.1.3 Feedback

In spite of these very satisfactory results, the project in its current state has also exhibited several weaknesses of the language or its various implementations. That

information is valuable feedback for the vendors, as it provides clear and explicit directions for improvement. Here are some of them. Note however that the following concerns may no longer be valid, as the situation could have evolved since the study was originally published.

It is not completely trivial to type Lisp code both correctly *and* minimally (that is, without cluttering the code), and *a fortiori* portably. Current compilers have a tendency to behave very differently with respect to type declarations, and provide type inference systems of various quality. The Common Lisp standard probably leaves too much freedom to the compilers in this area.

Another weakness of Lisp compilers seems to be inlining technology. Some simply don't support user function inlining, the poor performance of others seems to indicate that inlining defeats their register allocation strategy. It is also regrettable that none of the tested compilers are aware of the integer constant division optimization (Warren, 2002, Chap. 10) from which inlining can greatly benefit.

On the object-oriented side, our study exhibited extremely divergent behaviors with respect to slot `:initform` type checking (probably slot type checking in general), in particular when safety is preferred over speed. Here again, it is regrettable that the Common Lisp standard leaves so much freedom to the implementation.

2.2 Future Work

As mentioned earlier, the project is split into three parts, and is currently halfway through part 2. The obvious plan for future work is thus to complete the missing bits. On top of that, several additional perspectives are worth describing in a little more detail. They are given below.

Our current experimental results occasionally exhibit oddities or surprising behaviors which are not easily explained. Some of these oddities have been reported to the concerned maintainers; some have even already been fixed or are being worked on. The remaining ones should be further analyzed, as they would probably reveal even more room for improvement in concerned implementations.

Currently, the project focused intentionally on micro-benchmarks for elementary operations. In a longer term, similar experiments should be conducted on more complex algorithms (with more local variables, function calls, *etc.*), for instance so that we can spot potential weaknesses in the various register allocation policies, as the inlining problem tends to suggest.

It would be interesting to measure the impact of compiler-specific optimization capabilities, including architecture-aware ones like the presence of Single Instruction, Multiple Data (SIMD) instruction sets, or even the state of the art in GPU access.

Until now, and because the initial focus was on elementary operations, Garbage Collection (GC) timings were intentionally left out of the study. When comparing so different languages however, we observe that it is difficult to avoid taking into account the differences of expressivity, and in that particular matter, the fact that memory management is automatic on one side, and manual on the other side. Even when looking at the Lisp side alone, we plan on including GC timings in the benchmarks, simply because GC is part of the language design, and also because different compilers use different GC techniques, which adds even more potential variance to the overall efficiency of one's application.

Our current experimental results were obtained on a specific platform, and with a limited number of compilers. It would be interesting to measure the behavior and performance of the same code on other platforms, and also with other compilers. We do have an automated benchmarking infrastructure to ease that process, although it is still somewhat rudimentary. In the long term, it would also be nice to set up a truly automated, web-accessible benchmarking platform, so that we can not only collect experimental results, but also track improvements or regressions over time.

Although not part of the Ansi Common Lisp standard, some implementations provide a certain degree of unification between structures and standard objects (for instance, accessing structure slots with `slot-value`). These features have not been tested at all, but it should be interesting to see how they behave.

Finally, when the project reaches part 3, we will likely end up comparing static meta-programming tools such as C++ templates *vs.* Lisp macros to generate fully dedicated code. Given the favorable results from part 1, comparing the performance of the generated code is not expected to provide much more information. On the other hand, the cost of abstraction in meta-programming lies in the compilation phase rather than in the execution one. Thus, it will be more interesting to compare the performance of Lisp and C++, in terms of compilation times rather than execution ones.

Chapter 3

DSL Design and Implementation

One Masters internship •
One book chapter, two conference papers •

There exist different approaches to DSL design and implementation. One of them consists of rewriting all or several parts of the usual language infrastructure (parser, interpreter or compiler, etc.). Another one prefers to reuse the infrastructure of an already existing language. The purpose of this project is to explore the advantages of the latter approach. Through experimental studies, we also show that the obtained performance is better with Lisp as the underlying language than with alternatives such as Converge (Tratt, 2005), Meta-Lua, or Meta-OCaml (Kiselyov, 2014).

Domain-specific language design and implementation is inherently a transverse activity (Ghosh, 2010; Fowler, 2010). It usually requires knowledge and expertise in both the application domain and language design and implementation from the product team, two completely orthogonal areas of expertise. From the programming language perspective, one additional complication is that being an expert developer in one specific programming language does not make you an expert in language design and implementation —only in *using* one of them. DSLs, however, are most of the time completely different from the mainstream languages in which applications are written. A General Purpose Language (GPL), suitable for writing a large application, is generally not suited to domain-specific modeling, precisely because it is too general. Using a GPL for domain-specific modeling would require too much expertise from the end-users and wouldn't be expressive enough for the very specific domain the application is supposed to focus on.

As a consequence, it is often taken for granted that when a DSL is part of a larger application, it has to be completely different from the GPL of the application. But what if this assumption were wrong in the first place? Following Fowler (2005) and Tratt (2008, sec. 2), DSLs can be categorized in two different families.

A *standalone*, or *external* DSL is written as a completely autonomous language. In such a case, a whole new compiler / interpreter chain needs to be implemented, presumably with tools like Lex, Yacc, ANTLR, *etc.* Such an approach provides the author with complete control (from syntax to style of execution), but leads to high development costs, potential code bloat, and high maintenance costs (Vasudevan and Tratt, 2011; Ghosh, 2011).

A second approach consists of reusing the capabilities of a host GPL, hence avoiding the need for rewriting a complete infrastructure. This approach leads to so-called *embedded* (Graham, 1993) or *internal* DSLs, an idea which is probably at least 50 years old (Landin, 1966). In order to implement an embedded DSL, one needs to express it in terms of the differences from the host GPL. To that aim, two possibilities exist.

The first one consists of translating the DSL program into a host GPL one by means of external program transformation tools such as Stratego/XT (Bravenboer et al., 2006) or Silver (Wyk et al., 2008). As for standalone DSLs, this strategy, leading to so-called *embedded heterogeneous DSLs*, is often motivated by the lack of extensibility of the underlying GPL. An alternative view, however, is that some GPLs are flexible enough to permit implementing a DSL merely as an extension to themselves. The resulting DSLs are called *embedded homogeneous*, and in such a case, "extension" means the ability to modify some aspects of the original GPL, in syntactic or even semantic terms.

3.1 Project Description and Current Results

Embedded DSLs offer the advantage of lightening the programming burden by making it possible to reuse software components from the host GPL. The purpose of this project is to explore the specificities of the embedded homogeneous approach.

First of all, this approach has several more or less obvious advantages over the alternatives, even more so when the DSL is not supposed to produce standalone executables, but instead, is part of a larger application (for example, a scripting, configuration or extension language). In such a situation, the final application, as a whole, is written in a completely unified language. While the end-user does not have access to the whole backstage infrastructure, and hence does not really see a difference with the other approaches, the gain for the developer is substantial. Since the DSL is now just another entry point for the same original GPL, there is essentially only one application written in only one language to maintain. The second advantage is that the DSL layer is usually very thin, because instead of implementing a full language, one needs to implement only the *differences* with the original one. Finally, when there is no more distinction between DSL and GPL code, a piece of DSL program can be used both externally (as a public interface) and internally, at no additional cost.

What may be less obvious, however, is such questions as which language features are critical for the embedded homogeneous approach, how they interact which each other, and how or where exactly to use them. In Verna (2012a)[1], we answer those questions and we demonstrate how, when put together in the most appropriate way, those features (mostly related to the extensibility of the language) can blur the frontier between GPL and DSL, sometimes to the point of complete disappearance. It is worth mentioning that using the extensibility of a language for DSL purposes is an idea that predates even the very concept of DSL, as demonstrated by Denert et al. (1975) and Pagan (1979). Also, the comparative literature on language extensibility is abundant already (van Deursen et al., 2000; Vasudevan and Tratt, 2011; Tratt, 2008; Elliott, 1999), but as is often the case, the Lisp language is unfortunately sorely missing from it. Thus, Verna (2012a) also aims at filling this gap. In addition to that work, which exposes mostly general considerations, we applied those ideas to a much more specific use-case, namely to embed a LaTeX layer directly in Lisp. This work was reported in

1 Verna, D. (2012a). Extensible languages: Blurring the distinction between DSLs and GPLs. In Mernik, M., editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 1. IGI Global.

Verna (2012c)[2] and Verna (2013)[3].

Apart from expressivity, another concern in DSL design and implementation is the performance of the resulting programs. Specifically in the case of the homogeneous embedded approach, a critical factor to performance is the amount of meta-programming involved which, in turn, depends on the exact features of the host GPL. Here again, the lack of literature motivated us to fill the gap. A preliminary study was already conducted and demonstrated that Lisp-based DSLs performed more efficiently than equivalent competitors. This study, however, is currently published only as a Masters internship student report.

2 Verna, D. (2012c). Star TEX: the next generation. In Beeton, B. and Berry, K., editors, *TUG'12, 33rd TEX Users Group Conference*, volume 33. TEX Users Group.

3 Verna, D. (2013). The incredible tale of the author who didn't want to do the publisher's job. In Beeton, B. and Berry, K., editors, *TUG'13, 34th TEX Users Group Conference*, volume 34. TEX Users Group.

3.2 Future Work

Since our comparative study of embedded homogeneous DSL implementations is published only as a technical report, the first and obvious next step is to make it up to date and polish it for academic publication. Also, a number of research areas are still to be explored. We list the most important ones below.

As noted by Kamin (1998) and Czarnecki et al. (2004), the embedded DSL approach has a number of drawbacks, among which are sub-optimal syntax and poor error reporting.

Since an embedded DSL is expressed in terms of *differences* from the original GPL, it may indeed be a lot of work to "bend" the original syntax sufficiently towards the desired one, if it is very different. In such a case, the amount of work could possibly turn out to be the same as in implementing a full parser. An interesting research topic is thus to explore, across all languages suitable to embed DSLs homogeneously, how far the new syntax can depart from the original one, and at exactly what cost.

Error management is also an important aspect in the design and implementation of a DSL. However well designed your DSL is, it does not necessarily prevent the end-user from making syntactic or semantic mistakes. One problem with the embedded approach is that the reporting of errors is usually poor, uninformative, or even confusing. Indeed, in an embedded homogeneous DSL, the reported errors are naturally related to the underlying host GPL instead of specifically those of the DSL, and may not make any sense for the end-user.

Research on better error reporting techniques for embedded DSLs exists (Tratt, 2008), but again, is missing Lisp, in spite of potentially useful (and rather unique) features that we would like to investigate. In particular, we have the intuition that the Common Lisp *Condition System*, with its ability to distinguish condition handlers form restart points (Seibel, 2005, chap. 19) could help improve error management in DSLs, even more so when accompanied with the automatically embedded debugger. On the other hand, some aspects of error reporting are likely to remain challenging, in particular when source code information is desirable or when macro-expansion is involved. There is still much to be done in these areas.

Some aspects of extensibility for DSL design and implementation are still controversial today, and hence worthy of an in-depth investigation. Two important such aspects are dynamic *vs.* static typing, and lazy *vs.* strict evaluation.

Grounding an embedded DSL in a dynamic language has several advantages. First of all, it is often undesirable to clutter DSL code with type annotations, especially from the perspective of a non programmer end-user. Even though some DSL-friendly

static languages such as Haskell provide a type inference system that allow static types to remain implicit to some extent, it is still much easier to implement a DSL when static type annotations are *never* required. On the other hand, potential type errors occur at run-time in a dynamic language, thus affecting the end-user, something also undesirable. In the static camp, some research already exists around the idea of *a priori* type checking for a DSL, rather than *a posteriori* in the host language (Taha and Sheard, 1997). Another interesting path to follow is that of gradual typing (Siek and Taha, 2006, 2007), such as provided in the Racket language for example, or using an optional but strong static type system in an otherwise dynamic language, as in the case of Shen (Tarver, 2015). Note that Racket is known to be a very good candidate for embedding DSLs (Tobin-Hochstadt et al., 2011).

One particular aspect of purely functional languages that is known to help in DSL implementation is *laziness*, or *normal order* evaluation (Elliott, 1999, section 8, Kiselyov and chieh Shan, 2009, section 5). The most frequently emphasized advantages are the ability to define potentially infinite data structures (a gain in expressivity), and new control primitives such as extended conditionals (which do not systematically need the values of all their arguments). On the other hand, if the DSL needs imperative constructs and side effects, a purely functional host language will get in the way.

In the Lisp family of languages, implementing a lazy evaluation scheme is possible, notably at the macro level, but it certainly is less natural or straightforward than benefiting from native laziness such as in, say, Haskell. Also, as we have already mentioned, working at the macro level comes with its own problems, for instance in terms of error reporting and debugging. For all these reasons, we think that there is still room for interesting research on mixing lazy and strict evaluation schemes in the context of DSL design and implementation.

Chapter 4

Context-Oriented Optimization

One Bachelor internship •
Two conference papers •

Context orientation is a very dynamic form of programming, in which the execution context *influences the definition and structure of the software components involved (e.g. classes, instances, methods, etc.). In this project, we propose a novel use for context orientation, in which this paradigm counter-intuitively serves for optimization purposes.*

Genericity aims at providing a very high level of abstraction in order, for instance, to separate the general shape of an algorithm from specific implementation details. Reaching a high level of genericity through regular object-oriented techniques has two major drawbacks: code cluttering (*e.g.* class / method proliferation) and performance degradation (*e.g.* dynamic dispatch cost). In this project, we explore the potential use of the context-oriented programming paradigm in order to maintain a high level of genericity, without sacrificing, either the performance, or the original object-oriented design of the application.

4.1 Project Description and Current Results

The idea of using context orientation for optimization purposes was born as part of a more general investigation on the compromise between genericity and performance. Most of the time, performance-critical applications are written in static languages. When genericity is also desirable, very sophisticated static meta-programming techniques can be used, notably through the use of templates in C++ (Géraud and Levillain, 2008). While the resulting code, being fully dedicated, can run very efficiently, it is often quite obfuscated and hence hard to write, read, and maintain. On top of that, the obligatory distinction between the development, compilation, and execution phases, makes that approach hardly suitable to interactive applications.

Being more into the dynamic and interactive world, we found it interesting to explore the compromise between genericity and performance from that alternative point of view, with the intuition that the code would be easier to work with (especially for interactive applications), but also more challenging performance-wise. On the other hand, with the existence of optional static type systems in some languages, or gradual typing in others, it would also become possible to use dynamic meta-programming techniques to generate fully dedicated code, just like in the static camp.

The use of context orientation for optimization purposes constitutes one exploratory path in this more general agenda.

In order to ground the research into a realistic application domain, we chose the field of image processing, as it is already an area of expertise in our lab (Levillain et al., 2010). To that aim, we started by developing a first prototype, called Climb, the Common Lisp Image Manipulation Bundle (Senta et al., 2012)[1]. Although still experimental, the library not only provides an extensible core of data structures and algorithms, but also a DSL and a graphical modeling language, in order to ease the writing of complex processing chains.

1 Senta, L., Chedeau, C., and Verna, D. (2012). Generic image processing with Climb. In *ELS'12, 5th European Lisp Symposium*, Zadar, Croatia.

The idea behind genericity is to be able to write algorithms only once, independently of the data types to which they may be applied. In the context of image processing, being fully generic means being independent from the image formats, pixel types, storage schemes *etc.* To this aim, Climb provides different abstractions, among which are the following. A *site* is an abstract view of a pixel location. Image sites may be 2, 3, or nD coordinates, and the grids need not be rectangular (for example, some images types have hexagonal grids). A *site set* represents a collection of sites (a whole image, a neighborhood, *etc.*) over which it is possible to iterate. Climb also has abstract pixel *values* such as Boolean (for black and white images or masks), RGB, RGBA, *etc.* Values can in turn be encoded in different forms (integers, floats, *etc.*). Finally, image processing algorithms can be written on top of a core of abstract primitives such as site iterators, generic value accessors *etc.*

This very high level of abstraction makes it trivial to work on peculiar kinds of images such as graph-based ones, that is, images where the pixel adjacency relation is represented by a graph instead of a grid. Graph nodes are just a special kind of site, site iterators and value accessors work on them out of the box, so that any image processing algorithm written in Climb will also work on graph-based images with no modification to the code.

In Verna and Ripault (2015)[2], we added support for context-oriented programming in Climb, and we demonstrated how this paradigm could be used, in a somewhat counter-intuitive manner, to optimize the algorithms otherwise written in a highly generic fashion. The general idea is that instead of cluttering the design with a proliferation of classes and hierarchies for every single type of grid, site, or value, contextual *layers* will automatically switch the object-oriented model according the characteristics of the image currently being processed. Two examples of contextual optimization are given below, one behavioral, one structural.

2 Verna, D. and Ripault, F. (2015). Context-oriented image processing. In *COP'15, Context-Oriented Programming Workshop.*

One example of behavioral optimization lies in the handling of static type annotations in CLOS. Instead of providing a whole hierarchy of say, RGB values, one for each kind of value type (*e.g.* `fixnum`, `float`, *etc.*), it becomes possible to provide a *single*, layered, RGB value class, each layer adding the appropriate type annotation to the slots. In a similar fashion, all methods performing arithmetic operations on these slots can in turn be equally layered and annotated, so that with full optimization, we end up with fully dedicated, static code instead of generic one.

One example of structural optimization lies in the storage scheme for pixel values. In the fully generic version, nD images are stored in n-dimensional arrays of values, the values being in turn instances of different classes, such as the aforementioned RGB one. This abstract representation is not efficient, as a lot of indirections and accessor calls are involved for pixel access. Suppose, however, that we know in advance that the image is a 3D rectangular grid of RGB `fixnum` values. A much more compact and iteration-friendly storage scheme is to use a single 1D vector of inlined `fixnum` triplets

(one for each channel). Here again, a carefully designed set of layers makes it possible to automatically switch the internal image storage scheme to the most appropriate representation, depending on the context, and also to switch the implementation of all methods accessing that storage.

Preliminary performance measurements were conducted on several cases of either behavioral, or structural contextual optimization. They were reported in Verna and Ripault (2015) and show promising results.

4.2 Future Work

Our experiments with context-oriented optimization use ContextL (Costanza and Hirschfeld, 2005) as the underlying infrastructure. We currently don't know the potential performance impact of the infrastructure itself. ContextL goes through the Clos Mop to dynamically generate classes that represent combined layers, and dynamically dispatches on layered functions. Consequently, even in the optimized versions, an overhead is to be expected for every layered function call, the order of magnitude being that of a multi-method dispatch. This overhead could end up being a real problem for very short layered functions called very often, as is frequently the case in the image-processing domain. Another potential problem is with context switches. Even though layer activation can be implemented efficiently (Costanza et al., 2006), it is nevertheless possible that the cost becomes proportionally important in cases where we would end up doing a lot of dynamic context switches. We plan to investigate these issues in the future.

Despite promising preliminary results, we have encountered several limitations that we intend to address in the future as well. Some of them are outlined below.

In ContextL, layers are defined only as symbols and do not retain state *per se*. This, in our opinion, limits their expressivity. In order to compensate, we had to define layered functions to obtain information about the currently activated layers in Climb ("layered closures" in some sense). Ideally, layers should be stateful, perhaps Clos objects.

Another related concern is the modeling of relations between layers, in particular, joint activation logic. ContextL provides no mean to express logical relations between layers. Hence, layers supposed to be active at the same time need to be all (de)activated manually, and it is possible to put the library in an illogical state, resulting in run-time errors.

We also quickly faced the well-known "coercion problem". For example, some image processing chains may involve parallel branches in which different optimizations are active. When two or more parallel branches are joined back together, data coming from different contexts need to be "reunited", and there is currently no clean / automatic way to coerce objects from one context to another. On top of that, we still have to investigate the use of both different threads and different contexts for parallel branches, if that is even possible.

Context-oriented programming is one possible answer to a general question we have been investigating: expressing optimization as a cross-cutting concern, that is, without losing either genericity or the original design.

The contextual solution described here is not the only possible answer to this problem, although it seems quite convenient so far. For example, instead of precompiling the processing chains equipped with a context-oriented infrastructure, we

can also wait for an image being loaded, and then generate, compile, and execute a fully dedicated program. In the long term, we plan to incorporate this approach, along with other paradigms in the study, and provide an in-depth comparison of their respective merits.

Aspect-oriented programming (Kiczales et al., 1997) is a related paradigm which has been used for optimization already (loop fusion, memoization, pre-allocation of memory, *etc.*). Mendhekar et al. (1997) details a case-study of aspect-oriented programming for an image processing library. We also intend to compare with a mixins approach (Smaragdakis and Batory, 2001), and a purely functional one. Research comparing aspects, mixins and monads already exist (Hofer and Ostermann, 2007; Oliveira, 2009).

Finally, we also intend to investigate the use of more recent paradigms enabled in dynamic object-oriented environments, such as predicate (Ernst et al., 1998; Ucko, 2001) or filtered dispatch (Costanza et al., 2008).

Chapter 5

Rational Type Expressions

One Bachelor internship, one defended Ph.D. •
Two conference papers •

In dynamic languages, sequences such as lists and vectors may hold values of arbitrary types. In this project, we explore a way to express the fact that such a sequence may still present regularities in the types of the contained values. We also explore the idea of destructuring such a sequence by pattern-matching, and the potential applications of such a facility.

Programming languages support sequences of values in various forms, such as lists (notably in functional languages) and vectors or arrays. In the specific case of dynamic languages, sequences may contain values of any type. We call these *heterogeneous sequences*. Even when a sequence is heterogeneous, there can be some regularity in the sequence of value types. For example, a property list alternates names (say, symbols), and values of any type. An interesting problem is how to express, in a declarative way, the regular structure of such sequences. This project explores one possible path to answer this question, and the consequences of such a feature in terms of expressivity and performance.

5.1 Project Description and Current Results

In dynamic languages allowing it, static type annotations provide clues for the compiler to make optimizations in performance, space, safety, debuggability, *etc.* Because type information is available at run-time, application programmers may as well make explicit use of types within their programs, for example, to dispatch polymorphic behavior by hand, based on the type of a value (one central feature of the object-oriented paradigm).

This project was born out of an observed weakness of the Common Lisp type system, when it comes to sequences, such as lists or vectors. So-called *specialized arrays* are a special kind of arrays, the values of which are of a restricted type (Ansi, 1994, Section 15.1.2). Thus, it is possible to express that a vector is general (may contain anything), homogeneous (for example, a vector of floats), but *not* heterogeneous with some regularity (for example, a vector rigorously alternating strings and numbers), and neither an irregular but known list of elements types. In a similar way, the `cons` type specifier (Ansi, 1994, System Class `cons`) allows restricting the types of its `car`

and `cdr`, but there is no standard way to declare the types (whether homogeneous, heterogeneous, regular or not) for all the elements of a list.

5.1.1 Features

This project introduces the concept of *rational type expression* for abstractly describing patterns of types within sequences. The concept is envisioned to be intuitive to the programmer in that it is analogous to patterns described by regular expressions (Hopcroft et al., 2006, Chapters 3 & 4). Just as the characters of a string may be described by a rational expression such as $(a \cdot b^* \cdot c)$, matching strings such as `"ac"`, `"abc"`, and `"abbbbc"`, the rational type expression $(string \cdot number^* \cdot symbol)$ will match vectors like `#("hello" 1 2 3 world)` and lists like `("hello" world)`. While rational expressions match character constituents of strings according to character equality, rational *type* expressions match elements of sequences by type.

In addition to the concept of rational type expression, which is a theoretical formalism, this project also introduces an S-Expression based, prefix denotation for it, called *regular type expression*. For example, the regular type expression `(:1 string (:* number) symbol)` corresponds to the rational type expression $(string \cdot number^* \cdot symbol)$.

Rational type expressions are plugged into the Common Lisp type system by way of a specific type called `rte`, parameterized by a regular type expression. The members of such a type are all the sequences matching the given regular type expression. For example, it becomes possible to *declaratively* check that the type of `value` is a list of two numbers by writing `(typep value (and list (rte (:1 number number))))`. Contrast this with *programmatically* accessing the two elements and checking for their types.

Such a declarative system to describe patterns of types within sequences has great utility for program logic, code readability, and type safety. Rational/regular type expressions were presented in Newton et al. (2016)[1] and Newton and Verna (2018a)[2], along with several preliminary use cases. We briefly describe them below.

Because Common Lisp strings are sequences of characters, it is possible to use an `rte` type to perform traditional regular expression pattern matching on strings. The performance of `rte` for this task can even be better than that of Perl-Compatible alternatives such as `cl-ppcre` (Weitz, 2015), at the expense of being less expressive, because limited to what rational expressions can do.

Common Lisp specifies different kinds of *lambda lists*, used in different contexts. Ordinary lambda lists, defining the regular function call syntax, are slightly different from macro lambda list, or from the ones used in `destructuring-bind`. Lambda list syntax can be tricky or confusing at times, so `rte` can be used as a type checker, both for validation and warning about common mistakes.

The aforementioned `destructuring-bind` macro allows some form of pattern matching on lists, provided that their structure is known. When the said list can have different forms, it is necessary to programmatically check for the list structure, and issue different calls to `destructuring-bind` accordingly. By using a combination of `typecase` on `rte` types and `destructuring-bind`, it is possible to provide a new construct, which we call `destructuring-case`, doing both the list structure check and the destructuring at the same time.

1 Newton, J. E., Demaille, A., and Verna, D. (2016). Type-checking of heterogeneous sequences in Common Lisp. In *ELS'16, 9th European Lisp Symposium*, pages 13–20, AGH University of Science and Technology, Krakow, Poland.

2 Newton, J. E. and Verna, D. (2018a). Recognizing hetergeneous sequences by rational type expression. In *Meta'18, Meta-Programming Techniques and Reflection Workshop*, Boston, MA, USA.

5.1.2 Implementation

The implementation of `rte` uses well known techniques founded in rational language theory, with some peculiarities due to the fact that we are working on types.

In order to determine whether a given sequence matches a particular regular type expression, we conceptually execute a Deterministic Finite Automaton (DFA) with the sequence as input. Thus, we must convert the regular type expression to a DFA, which needs only be done once, and can often be done at compile time (the `rte` type is often available statically; only the actual sequences usually appear at run-time).

There are various known techniques for converting rational expressions into DFAs, notably the *Rational Language Derivative* approach of Brzozowski (1964) and Owens et al. (2009), and that of Xing (2004). We chose the former because it allows more expressive rational expressions.

In order to work on Common Lisp types, a number of specificities need to be taken into account. The set of sequences of Common Lisp objects is not a rational language. The mapping of a sequence of objects to a sequence of types is not unique, which, in particular, led us to develop an algorithm for converting a set of Lisp types into an equivalent set of disjoint types. These problems and the workarounds are described in Newton et al. (2016).

Constructing the DFA can be extremely expensive (Hromkovič, 2002), so it is important to be able to do it at compile-time. Once a DFA is generated, there are basically two approaches for execution: we can either use a general function, accepting any DFA as input and running it, or generate specialized code for every single DFA we have. The latter approach is chosen for performance reasons. The advantage of a specialized function is that the number of states encountered at execution is equal or less than the number of elements in the target sequence. Thus, the time complexity is linear in the number of elements in the sequence, and is independent of the number of states in the DFA (which can be very large). Again, because most of the time, the `rte` types are available at compile-time, we effectively end up with only an additional $\mathcal{O}(n)$ cost to the run-time.

5.2 Future Work

The rational type expression formalism is purely theoretical, hence potentially applicable to any language offering support for heterogeneous sequences. As such, we would like to study its applicability to languages such as Python, Ruby, Lua (Ierusalimschy, 2013), Julia, or Javascript. Because type information is available at run-time in dynamic languages, converting a (declarative) regular type expression into the corresponding (programmatic) type-checker function is not expected to cause any problem. It is not obvious, however, whether we can reach the same performance as we do in Lisp. Indeed, our DFA execution functions make heavy use of very low-level, imperative constructs, such as `block` and `go`, allowing for very efficient native code.

Another aspect of this research has a much less obvious outcome, however, namely, the insertion of `rte` types into the native type system of the concerned language. In Common Lisp, `rte` types are defined using the `satisfies` type specifier, essentially allowing the type-checker to call any user-defined procedure. How to mimic this functionality, or even just whether it is feasible in other languages remains to be seen.

In the traditional object-oriented approach, methods are specialized to the types (or classes, since it is generally equivalent) of their argument(s). In that model, it may thus be possible to specialize a method on `list`'s or `vector`'s, but not on their potentially regular structure. We would like to study the idea of plugging `rte` types into the object-oriented dispatch mechanism to make this possible. If successful, a study on the performance of such a mechanism alone will be conducted. Also, it would then be interesting to explore its integration with even more generalised dispatch mechanisms, such as predicate (Ernst et al., 1998; Ucko, 2001) or filtered dispatch (Costanza et al., 2008).

Finally, note that `rte` was originally intended to deal with sequences (in other words, sets with an order relation). Consider however that other kinds of aggregates can be regarded as sequences if we add an order relation to them (for example, classes and structures with the slots ordered by lexical definition). We would like to explore the idea of using `rte` for matching class or structure instances based on the types of the slot values. In such a situation, an instance would turn out to be of two different types: its corresponding class or structure, but also of a specific `rte` type. In fact, doing this establishes some kind of bridge between structural and nominal typing (Pierce, 2002), and we can see potential applications, notably in terms of serialization.

Chapter 6

Type Calculus Performance

One Bachelor internship, one defended Ph.D. •
Four conference papers •

Contemporary Lisp compilers use the native S-Expressions of the language directly to represent types along with ad hoc and imperfect techniques for type calculus (type checking, sub-typing computation, etc.). We propose the use of BDDs as an alternative way to represent types, and compare the respective merits of both approaches. We also propose optimization techniques for several language primitives, along with a re-implementation of `subtypep`*, a predicate of critical importance in type calculus.*

The performance of type-related computation is of critical importance in dynamic languages for several reasons. First of all, a lot of it (notably type checking) happens at run-time instead of compile-time, so it has a direct impact on the efficiency of program execution. On top of that, type calculus is not restricted to the internals of the language; the programmer can also work directly on types, since the information is available at run-time. Finally, providing a facility for static type annotation in an otherwise dynamic language opens a large window to type-calculus optimization. This project deals with various performance aspects of type calculus. It is a direct consequence of the work on rational type expressions (Section 5 on page 19), but it is presented separately because its potential applications are much wider than just rational type expressions.

6.1 Project Description and Current Results

Common Lisp programs which manipulate type specifiers have traditionally used S-Expressions as the programmatic representation of types (Ansi, 1994, Section 4.2.3). Such a choice of an internal data structure offers advantages, such as homoiconicity (McIlroy, 1960; Kay, 1969), making the internal representation human readable (in simple cases), making programmatic manipulation intuitive, and enabling the direct use of Common Lisp primitives, such as `typep` and `subtypep`.

However, this approach presents some challenges and weaknesses that we faced, in particular when working on regular type expressions. As mentioned in Chapter 5 on page 19, one step in the process of constructing a DFA for matching a regular type expression consists of converting a set of overlapping types into a set of disjoint ones, which we call the Maximal Disjoint Type Decomposition (MDTD) Problem. When

working on this specific problem, we ran into two major obstacles. First, working with type specifiers denoted by S-Expressions has a dramatic impact on performance. Next, the `subtypep` primitive, which is at the core of the MDTD algorithm, is in general not implemented efficiently, and worse, doesn't necessarily provide correct or meaningful answers. We hence had to explore alternative representations for type specifiers, and alternative implementations of `subtypep`.

6.1.1 An Alternative Representation for Type Specifiers

In Newton et al. (2017)[1], we propose the use of BDDs (Akers, 1978; Bryant, 1986) as an alternative internal representation for Common Lisp types. BDDs have interesting characteristics such as representational equality (it can be arranged that equivalent expressions or equivalent sub-expressions are represented by the same object). While techniques to implement BDDs with these properties are well documented, their application to the Common Lisp type system presents obstacles which are also analyzed and presented.

In order to evaluate the pros and cons of BDDs as an alternative to S-Expressions for representing type specifiers, we provide two different algorithms for the MDTD problem, using either representation (hence a total of four variants). Performance measurements show that BDDs are a promising path, although not a definitive winner, and not necessarily in all situations.

1 Newton, J. E., Verna, D., and Colange, M. (2017). Programmatic manipulation of Common Lisp type specifiers. In *ELS'17, 10th European Lisp Symposium*, pages 28–35, Vrije Universiteit Brussel, Belgium.

6.1.2 An Alternative Implementation of `subtypep`

As mentioned earlier, our implementations of the MDTD problem make extensive use of `subtypep`, a Common Lisp predicate function for introspecting sub-typing relationships.

Every invocation of `(subtypep A B)` either returns the values `(T T)` when `A` is a sub-type of `B`, `(NIL T)` when not, or `(NIL NIL)` when the predicate could not (or failed to) answer the question. The latter can happen when the type specifier `(satisfies P)` (representing the type $\{x|P(x)\}$ is involved. For example, given two arbitrary predicate functions `F` and `G`, there is no way `subtypep` can answer the question `(subtypep '(satisfies F) '(satisfies G))`. However, some implementations abuse the permission to return `(NIL NIL)`. For example, in SBCL, `(subtypep 'boolean 'keyword)` returns `(NIL NIL)`, thus violating the standard which indeed requires a correct answer for all primitive types. The definition of the `keyword` type is in fact responsible for this failure (at least in that implementation, it involves the `satisfies` type specifier).

The inaccuracy of `subtypep` has a direct impact on our work with BDDs as an alternative representation for type specifiers. Indeed, as BDDs can be quite large, the predicate is used extensively to reduce the BDDs to the so-called Reduced Ordered Binary Decision Diagram (ROBDD) form (Gröpl et al., 1998). Using the ROBDD form ensures that there are no duplicate sub-trees in the BDD, and may affect the performance of the DFA's execution function. The unreliability of `subtypep` leads to many lost BDD reductions and therefore to the generation of sub-optimal code.

In Valais et al. (2019)[2], we present a new implementation of `subtypep`, based on initial work by Baker (1992). In this paper, Baker provides guidelines to obtain an implementation that is supposedly more accurate than, and as efficient as the average one. He does not, however, provide any serious implementation and his guidelines

2 Valais, L., Newton, J. E., and Verna, D. (2019). Implementing baker's `subtypep` decision procedure. In *ELS'19, 12th European Lisp Symposium*, pages 12–19, Genova, Italy.

are sometimes obscure. Along with our new implementation, we try to clarify several parts of his description and fill in some of its gaps or omissions. We also argue in favor of or against some of his choices, and present our alternative solutions. We further provide some proofs that might be missing in his article, and some early efficiency results.

6.1.3 `typecase` Optimization

A direct consequence of our work with BDDs and a more accurate version of `subtypep` is in potential optimizations of the `typecase` macro expansion, a user-level facility that we also use extensively in our work on regular type expressions. The `typecase` macro is a multi-branch conditional checking the type of its argument rather than its value. The conditional type specifiers may go from simple type names such as `fixnum`, to more expressive types involving the `satisfies` predicate, or even logical combinations of types.

Naive `typecase` implementations may lead to sub-optimal code for at least three reasons. When different conditional branches overlap, the same type checks could end up being performed multiple times (and at run-time). The specification *suggests* but does not require that the compiler issue a warning if a branch is unreachable (being completely shadowed by earlier ones). Therefore, implementations are free to leak unreachable code. Finally, if it can be determined that the provided branches perform a full type coverage, then the last one can be turned into a default branch, hence removing one (run-time) type check completely.

In Newton and Verna (2018b)[3], we contrast two approaches to optimizing `typecase` accordingly. The first one is based on heuristics intended to estimate the run-time performance of certain type checks. The second one makes use of ROBDDs as an alternate representation of type specifiers. Both approaches allow us to identify unreachable code, test for exhaustiveness of the clauses, and eliminate redundant checks. One critical aspect of the heuristic approach is to be able to re-order the different branches without breaking the semantics of the original code, something which may not be trivial in the presence of side-effects. Doing this reordering requires working on disjoint types, and is thus deeply connected to the MDTD problem mentioned earlier.

Both approaches have pros and cons. We demonstrate that the first approach is sensitive to the number of branches, and is not always able to remove redundant type checks. On the other hand, the second approach won't miss redundant checks, but may leave unnecessary ones behind.

6.1.4 `destructuring-case` Optimization

In the general area of conditional branching on types, related optimization techniques may also be applied to the `destructuring-case` macro that we introduced in Section 5 on page 19, as an application example of regular type expressions. Recall that `destructuring-case` pattern-matches structurally different lambda-lists, and then performs destructuring on the appropriate one. A risk of inefficiency associated with a naive implementation of `destructuring-case` is that the candidate expression being examined may be traversed multiple times, once for each clause whose format fails to match, and finally once for the successful one.

In Newton and Verna (2019a)[4], we provide an implementation which encodes

3 Newton, J. E. and Verna, D. (2018b). Strategies for typecase optimization. In *ELS'18, 11th European Lisp Symposium*, pages 23–31, Marbella, Spain.

4 Newton, J. E. and Verna, D. (2019a). Finite automata theory based optimization of conditional variable binding. In *ELS'19, 12th European Lisp Symposium*, pages 26–33, Genova, Italy.

the lambda-lists into `rte` types, uses DFAs to compile the type checking code, but also merges all the DFAs together, so as to share states into a single automaton. The resulting execution function has the property to avoid multiple traversals of the candidate expression.

6.2 Future Work

Because the BDD approach is not clearly better than the S-Expression one, we need to work on heuristics for predicting in which situation one approach is better than the other. Ideally, the two approaches should co-exist and be selected appropriately. It is known that algorithms using BDDs tend to trade space for speed. Our implementation is not much optimized yet, apart from ideas taken from Andersen (1999). Castagna (2016) suggests a lazy version of the BDD data structure which may reduce the memory footprint. We could also probably benefit from the expertise of the CUDD developers in BDD optimization (Somenzi, 2016).

Our re-implementation of `subtypep` is still under active development. It currently targets SBCL only, and focuses almost entirely on result accuracy. It supports primitive types, user-defined types (via `deftype`, classes, and structures), `member` / `eql` type specifiers, and ranges (*e.g.* `(integer * 12)`). The plan is, of course, to complete the implementation. After that, we also intend to optimize it, even though encouraging results on efficiency have already been observed, without any particular work on that aspect.

For future extensions to this research we would like to experiment with extending the `subtypep` implementation to allow application level extensions, and thereby examine run-time performance when using `rte` based declarations within function definitions.

As far as the `typecase` optimization is concerned, the heuristics used by the first approach are still simplistic. It is thus desirable to find more accurate ones, notably taking into account how computationally intensive certain type specifiers are to manipulate. While `typecase` is a user-level macro, it is also used by the compilers themselves. There are some limitations to a portable implementation of it, notably the lack of a standard expander for user-defined types, so this, as well, is an interesting topic for future research.

As far as the `destructuring-case` optimization is concerned, there is still work to be done in improving our handling of DFAs. The simplification algorithm we use to eliminate equivalent states is not optimal. There is also the question of DFA merging by synchronized cross-product. Currently, the merged DFAs are sub-optimal, so an open question is whether to improve the input DFAs *before* doing the cross-product, only simplify the resulting cross-product, or doing both.

Because all dynamic languages reify types at run-time and offer at least introspective access to them, it would be interesting to study the efficiency of type calculus in those languages (*e.g.* Python, Ruby, Lua, Julia, and Javascript), and maybe try to apply our original ideas to them, notably the use of BDDs, and alternative implementations for the equivalent of `subtypep` and other type-related primitives.

Chapter 7

Worst-Case BDD Study

One defended Ph.D. •
One journal paper •

BDDs are a very simple data structure, yet useful in a very wide range of applications. Although simple to implement, their dynamic behavior, notably in terms of number of nodes, shape, and memory footprint is neither obvious, nor intuitive. This project contributes new theoretical, statistical, and experimental analysis of the dynamic behavior of BDDs, in typical and worst-case scenarios. This knowledge helps making more pertinent decisions about their usefulness in specific applications.

Out of a concern for performance, the development of rational type expressions (Section 5 on page 19) led us to envision the use of BDDs as an alternative data structure for reifying Common Lisp type specifiers and reason about them. The preliminary performance experiments contrasting BDDs with S-Expressions (the native type representation) were neither very conclusive, nor easy to interpret. It was thus deemed important to get a better perspective on the behavior and performance of BDDs, independently from any concrete use.

7.1 Project Description and Current Results

Binary Decision Diagrams are a data structure useful for representing Boolean expressions, integrated circuit design, type inferencers, model checkers, and many other applications. Decision diagrams have been defined in various forms in currently available literature. Colange (2013) provides a succinct historical perspective, including BDDs (Bryant, 1986), Multi-Valued Decision Diagrams (Srinivasan, 2002), Interval Decision Diagrams (Strehl and Thiele, 1998), Multi-Terminal BDDs (Clarke et al., 1997), Edge-Valued Decision Diagrams (Lai and Sastry, 1992), and Zero-Suppressed Binary Decision Diagrams (Minato, 1993).

The particular variant we investigate in this project is the ROBDD. When we use the term ROBDD, we mean, as the name implies, that the BDD has been *reduced* (R) and its variables *ordered* (O) in specific ways. It is worth noting that there is variation in the terminology used by different authors. For example, Knuth (2009) and Bryant (2018) both use the unadorned term BDD for what we call ROBDD. Even though the ROBDD is a lightweight and very simple data structure, some of its

behavior regarding the amount of necessary memory allocation may not be obvious in practice.

In Newton and Verna (2019b)[1], we convey an intuition of the expected sizes and shapes of ROBDDs from several perspectives. We explore (experimentally, statistically, and theoretically) the typical and worst-case ROBDD sizes in terms of number of nodes. We define the "residual compression ratio" as the ratio between the number of nodes in the reduced form and the un-reduced one (hence, the smaller the ratio, the better).

1 Newton, J. E. and Verna, D. (2019b). A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. *ACM Transactions on Computational Logic*, 20(1).

First, we provide an analysis of the explicit space requirements of ROBDDs. This analysis includes an exhaustive characterization of the sizes of ROBDDs of up to four Boolean variables, and an experimental random-sampling approach to provide an intuition of size requirements for ROBDDs of more variables. We additionally provide a rigorous prediction for the worst-case size of ROBDDs of n variables. We use this size to predict the residual compression the ROBDD provides. While the size itself grows unbounded as a function of n, the residual compression ratio shrinks asymptotically to zero. That is, ROBDDs become arbitrarily more efficient for a sufficiently large number of Boolean variables.

In order to perform our experiments, we designed an algorithm for generating a worst-case ROBDD for a given number of variables. This algorithm may be useful to projects deciding whether the ROBDD is the appropriate data structure to use, and in building worst-case examples to test their code.

While our theoretical results are not surprising, as they are in keeping with previous literature, we believe our method contributes to the current body of research by our experimental and statistical treatment of ROBDD sizes. Our approach for this development is different from what we have found in current literature, in that while it is mathematically rigorous, and at the same time, highly based on intuitions gained from experimentation.

7.2 Future Work

There are several shortcomings to our intuitive evaluation of statistical variations in ROBDD sizes. For example, our random-sampling measurements led us to believe that when the number of variables grows, the difference between the average and worst-case ROBDD sizes becomes negligible, an observation which seems related to the *Shannon Effect* (Gröpl et al., 1998). We would like to continue this investigation to better justify this guess.

The number of samples we took in our statistical experiments were constrained by the computation time at our disposal. To get an idea of the figures involved, computing approximately 3000 samples of 10-variable ROBDDs takes around 50 hours. We would like to extend our platform to work in a multi-threaded environment, thus exploiting more cluster nodes for shorter periods of time. It may also be possible to exploit other Common Lisp features such as dynamic extent objects, or weak hash tables to better manage the memory footprint of our computations, thus achieving more ROBDDs computed per unit of time.

Our current implementation for constructing many ROBDDs while preserving structural identity is to memoize them in a hash table. This hash table can become extremely large, even if its lifetime is short. We have characterized the worst-case size of an ROBDD as a function of the number of Boolean variables. This characterization

ignores the transient size of the hash table, so one might argue that our size estimations are misleading in practice. We would like to continue our experimentation and analysis to provide ways of measuring or estimating the hash table size, and potential ways of decreasing the incurred burden, perhaps getting inspiration from CUDD and the cache management system described by Brace et al. (1990). For example, we suspect that most of the hash table entries are in fact never re-used. We would like to experiment with weak hash tables as well: once all internal and external references to a particular hash table entry have been abandoned, that hash table entry can be removed, thus potentially also freeing the child nodes. Measuring the effectiveness of weak hash tables is ongoing research.

Minato (1993) suggests that using the BDD variant called 0-Sup-BDD is well suited for sparse Boolean equations. We see potential applications for 0-Sup-BDDs in type calculations, especially when types are viewed as sets, as is the case in Common Lisp. In such a situation, the number of types is large, but each type constraint equation scantly concerns few types. We would like to experiment with 0-Sup-BDD-based implementations of our algorithms, and contrast the performance results with those found thus far.

It is known that algorithms using BDDs tend to trade space for speed. A question naturally arises: can we implement a fully functional BDD which never stores calculated values. The memory footprint of such an implementation would *potentially* be smaller, while incremental operations would become slower. It is not clear whether the overall performance would be better or worse. Castagna (2016) suggests a lazy, more memory-friendly version of the BDD data structure, which would have a positive effect on our BDD based algorithms. This approach suggests dispensing with the excessive heap allocation necessary to implement Andersen's approach (Andersen, 1999). Moreover, our implementation (based on the Andersen model) contains additional debugging features which increase memory usage. We would like to investigate which of these two approaches gives better performance, or allows us to solve certain problems. It seems desirable to attain heuristics to describe situations in which one or the other optimization approach is preferable.

Even though both Andersen (1999) and Minato (1993) claim the necessity to enforce structural identity, it is not clear whether, in our case, the run-time cost associated with this memory burden, outweighs the advantage gained by structural identity. Furthermore, the approach used by Castagna (2016) seems to favor laziness over caching, lending credence to our suspicion.

Somenzi (2016) uses a data structure called `DdNode` to implement different flavors of BDDs, including Algebraic and Zero-Suppressed ones. We have already acknowledged the need to experiment with other BDD flavors to efficiently represent run-time decisions on types, for example, to perform simplification of type-related logic at compile-time (Newton et al., 2017; Newton and Verna, 2018b). The work of Lozhkin and Shiganov (2010) and Shannon (1949) may give insight into how much improvement is possible, and hence whether it is worth dedicating compilation time to it.

Chapter 8

Dynamic Object Orientation

Two journal papers, one conference paper •

All sorts of interesting aspects to object orientation arise in the context of a dynamically typed language, in interaction with other paradigms such as the functional one, when both introspection and intercession are available, and when the object layer is built out of a Mop*. While the project dealing with context-oriented optimization (Chapter 4 on page 15) focused on a particular* use *of dynamic object orientation, this project seeks to make the very paradigm continue to evolve.*

There is no such thing as an "object-oriented" paradigm, as Nierstrasz (2010) complained about, in an enjoyably sardonic fashion. What mainstream industrial languages call "object-oriented" today is hardly what Alan Kay had in mind (Kay, 2003) when he designed Smalltalk (Kay, 1993) and coined the term. The classical approach tends to view object orientation as hierarchies of inheritable classes for state, and polymorphic methods for behavior. On the other hand, languages such as Self (Ungar and Smith, 1987) and JavaScript rather use prototypes and delegation (Ungar et al., 91; Lieberman, 1986). Even within one family, such as the class-based one, a diversity of approaches is to be found. For example, inheritance can be extremely limited, as in Julia (which, to be honest, does not really claim to be fully object-oriented). The dynamic dispatch can be implemented in the form of message-passing as in Smalltalk, or via multi-methods (Castagna et al., 1992; Castagna, 1996) as in Clos, *etc.*

8.1 Project Description and Current Results

Facing this "object-oriented jungle", we are interested in the following questions. Can we contribute to putting some order in it, and how? What are the specificities of the object-oriented paradigm when mixed with the other ones we are interested in? Is this paradigm as expressive as it will ever get?

In Verna (2008)[1] and Verna (2010b)[2] we follow the footsteps of Norvig (1996) and Bruce et al. (1995), and address, from the perspective of paradigm mixture, two classical object orientation problems: design patterns (Gamma et al., 1994; Buschmann et al., 1996) and binary methods. Those problems are interesting because they underline, either the deficiencies of the classical object-oriented approach (in the case of

1 Verna, D. (2008). Binary methods programming: the CLOS perspective. *Journal of Universal Computer Science*, 14(20):3389–3411.

2 Verna, D. (2010b). Revisiting the visitor: the just do it pattern. *Journal of Universal Computer Science*, 16(2):246–271.

binary methods), or the influence of the general expressivity of the target language (in the case of design patterns).

Some of the points we make are well known already, and are merely restated in a new and hopefully interesting setting. In particular, we emphasize why multiple-dispatch on methods external to classes increases the Separation of Concerns (SOC) by clearly separating inheritance from polymorphism. We also stress how dynamically typed object orientation makes it possible to lighten the object-oriented design of the application by getting rid of traits often associated with the object paradigm, whereas in fact being merely static typing idiosyncrasies (*e.g.* abstract classes). We also contrast the use of objects and methods, as opposed to (first-class) functions and lexical closures, to implement stateful behavior.

Some other points we make, however, are probably much less frequent in the existing literature, probably because not many languages offer the necessary paradigms to raise them. In fact, most of them involve reflexivity (both intercession and introspection), and the existence of a MOP.

The first point consists of bridging the gap between the functional and object-oriented paradigms. In Verna (2010b) for example, we observe that the visitor pattern is essentially a form of mapping (originally a functional concept), only *structural*, since it involves traversing the components of an aggregate object rather than a sequence of values. Nevertheless, the introspective capabilities of the object system make it possible to implement a universal structural mapper, not requiring any preliminary knowledge of its argument type. On top of that, we demonstrate that it is also possible to fully integrate it with the built-in facilities of the language. The result is in fact a unified "traversal" facility, acting like a transparent visitor pattern on objects, and like a regular functional `map` on sequences.

The second point we never cease to make is to fight against the general idea that dynamic languages are unsafe or too permissive (an idea often propagated by the advocates of static typing). In Verna (2008) for example, we do not stop at a simple implementation of binary methods which would make the concept merely available. On the contrary, we also show how the MOP allows us to design and implement a very secured version of it. In particular, we show how to establish two different levels of protection. The first one enforces a correct *usage* of binary methods, by automatically checking that any call to a binary function is made with two arguments of the exact same type. The second one enforces a correct *implementation* of binary methods, by automatically checking that all existing methods in fact specialize on two arguments of the same type, and also that no necessary method implementation is missing (what we call "binary completeness"). Note that these safeguards behave exactly like a strong yet dynamic typing system: it becomes impossible to misuse or misimplement a binary function, but the safeguards are triggered as late as possible (something that dynamic typing detractors may still call "unsafe"). In doing so, we do have a completely secure system, while preserving dynamic flexibility, such as the ability to add or remove binary methods at run-time.

In addition to these considerations on paradigm mixture, the question of extending the expressivity of dynamic object orientation even more was also raised more recently. Several attempts at generalizing object-oriented concepts already exist, notably regarding polymorphic dispatch (Ernst et al., 1998; Ucko, 2001). Recently, we started to work on a new generalization of polymorphic dispatch, based on a facility which is already available in CLOS, although in a somewhat embryonic and ill-defined state.

In traditional object-oriented languages, the dynamic dispatch algorithm is simplistic and hardwired: a polymorphic call triggers the most specific method only. Any other method call must be programmed explicitly. CLOS generalizes the dispatch algorithm through the concept of *method combinations*: when several methods are applicable, it is possible to select several of them, decide in which order they will be called, and how to combine their results. Method combinations are programmable and are specified in a declarative way, which greatly improves the SOC. Indeed, when in need for a non-conventional method call chain, the chaining code can be expressed outside of the methods themselves.

Although a powerful abstraction, method combinations are unfortunately underspecified, and the specification sources (either the Common Lisp standard, or the CLOS MOP) are even sometimes contradictory. The first step of this research was hence to clean up the concept, which we did in in Verna (2018b)[3]. More specifically, we pointed out the caveats of their current specification, and exhibited the consequences, notably showing inconsistencies in otherwise conformant implementations. We also proposed a new MOP for a cleaned up version of the concept, called *method combinators*, along with a prototype implementation.

3 Verna, D. (2018b). Method combinators. In *ELS'18, 11th European Lisp Symposium*, pages 32–41, Marbella, Spain.

As a direct consequence of this work, one first generalization of dispatch was achieved. Method combinations are normally attached to a generic function in a way that makes it very impractical to change them (most of the time, it is simpler to create a brand new generic function). With our new implementation, on the other hand, it becomes trivial to call a generic function with *alternative* combinators, even a different one for every call. In some sense, the way applicable methods are combined becomes completely separate from the generic function itself. This, however, is only the first step in a bigger agenda.

8.2 Future Work

Specifically on method combinators, there is still some work left to do. They are currently provided as a prototype and a proof of concept only. Their API still needs to be stabilized, and their implementation made as portable as possible. Portability is expected to pose some difficulties, as it involves modifying the internals of the MOP itself. To be more precise, it *is* possible to implement them in a completely portable fashion, but the impact on performance is prohibitive. The second difficulty is that our proposed MOP for method combinators not only provides a new API, but also contains improved or fixed versions of "official" functions specified in the MOP standard (Kiczales et al., 1991). Although the MOP itself is not part of the ANSI specification of the language, updating it is not expected to be easily accepted by existing vendors.

As mentioned earlier, our work on method combinators is the first step in a bigger plan. The plan is in fact to increase yet again the degree of SOC in the dynamic dispatch mechanism. Generic functions clearly separate behavior (polymorphism) from structure (inheritance). Predicate or filtered dispatch, along with multi-methods, generalize the method selection process. A fully functional concept of method combinators such as the one we propose (and especially the ability to specify a combinator at function call time) helps separating the method(s) *execution* process from the method(s) *selection* one.

In this context, we believe there is no more reason for methods to belong to

a *single* generic function at a time (as required by the Common Lisp standard). Consequently, we propose the concept of "standalone methods". With such a design, we believe to have pushed the SOC on step further in the dynamic dispatch facility. Method combinators exist as global objects, so do methods and generic functions, which simply become mutable sets of shareable methods. Note that this idea has not been explored *at all* yet, but we are eager to study the implications in terms of expressivity, the potential limitations, and perhaps more importantly, how it can interact with filtered or predicate dispatch.

Let us end this section with two ideas regarding future projects around dynamic object orientation. These two ideas do not fit anywhere else, neither have they been investigated at all yet, but it is our intention to explore them sooner or later, so we thought we might as well put them down here.

The first one deals with the segregation between the class-based and the prototype-based approach. In the current object-oriented landscape, it seems that the segregation holds, although some level of co-existence is possible, especially in dynamic languages (prototype systems are known to have been implemented on top of class-based ones, for example). There also seems to be a lot of confusion between the two approaches, notably when it comes to the implementation of the concepts. For instance, Python claims to be a class-based language, but the way object properties are looked up definitely sounds like a prototype-based approach more than a class-based one. To paraphrase Castagna (1995), we suspect that the class *vs.* prototypes debate is in fact a "conflict without a cause", and we would like to explore potential ways to re-unite the two approaches, hopefully in a formal way, with a calculus of some sort.

The second prospective idea we want to explore some time in the future is related to the tight integration between types and classes that practically all class-based object-oriented languages suffer from. The sub-typing / sub-classing equivalence is an old and well-known deficiency of the traditional approach (America, 1987; Liskov and Wing, 1994). To the best of our knowledge, the only two programming languages to have ever attempted to separate types from classes are POOL (America, 1991) and Cecil (Chambers, 92; Dean et al., 1996), maybe also with its successor Diesel. We are very interested in exploring this idea in the context of CLOS, all the more than the re-implementation of `subtypep` that we are working on (*cf.* Chapter 6 on page 23) may provide us with the hooks that we need into the Common Lisp type system.

Chapter 9

Exception-Oriented Programming

No publication yet •

In this project, we want to explore the opportunities offered by extended exception handling mechanisms, such as the ones of Common Lisp and Dylan. Most of the time, languages with explicit support for exceptions focus on error management and recovery. The extended exception handling systems we are interested in not only provide more expressive error management, but can also be bent to address other problems, unrelated to error management or even exceptions.

With the notable exceptions of Go and Rust, many contemporary high-level programming languages offer some support for structured exception handling (Goodenough, 1975). This is true of both statically typed languages such as C++ or Java, and dynamic ones such as Python, Ruby, or Julia.

Except for small variations, some of them inherent to the very nature of the concerned languages (*e.g.* statically *vs.* dynamically typed), support for exception handling is almost always the same, based on the classical try-catch/throw duet (`try-except/raise` in Python, `rescue/raise` in Ruby). A `try` block executes code that may `throw` an exception, disrupting the normal flow of control, yet it is possible to `catch` and handle them gracefully instead of just aborting the execution.

More expressive exception handling mechanisms *do* exist, however, and although they are not very common, we believe that they still have some unexplored expressivity potential.

9.1 Project Description and Current Results

The benefits of language-level support for exception handling are well known today. Without such support, one ends up using specific return values to indicate an error, which is not always possible, *cf.* the "semi-predicate" problem (Norvig, 1992, p. 127). One can otherwise resort to global variables, such as `errno`, or even reserve return values for indicating state, and passing in-out arguments by reference to the callee, to be filled in with results. All such idioms are commonly encountered in the POSIX (2018) system calls API for example (note that C doesn't have any exception handling mechanism, although it is possible to build one on top of `setjmp/longjmp`).

On the other hand, language-level support for exception handling has many advantages. First of all, the more declarative shape of code dealing with reified exceptions

increases the SOC: the code for normal execution becomes separate from that dealing with "abnormal" behavior. The second advantage of native exception handling is that exceptions do not have to be caught and processed exactly where they occur. For example, errors may be raised at a low level in the code, while being treated only at a higher application level. The third advantage is that, when exceptions are reified, it is possible to provide a rich error ontology, with at least a hierarchy of built-in ones, upon which the programmer can act. Moreover, and this is the fourth advantage, first-class exceptions, which is the case when they are implemented in an object-oriented fashion, are usually extensible. Programmers can hence define their own sub-hierarchies, for example by subclassing `std::exception` in C++, `Exception` in Java or Python, and `StandardError` in Ruby (in Julia, one would also derive from `Exception`, but creating user-level exception hierarchies is not possible because all concrete types are final).

The classical try-catch/throw model suffers from a number of limitations, rarely acknowledged by the community (probably because it is so widespread nowadays that we simply take it for granted).

First of all, there is a frequent confusion between "exception" and "error", as noticeable in the official Java tutorials (Oracle, 2019) or Ruby documentation (Ruby, 2019), where the two terms are used interchangeably. This means that even if that is not strictly the case (for example, Ruby has an exception class for Unix signals), there is a clear bias towards considering exception handling mostly as error management. This bias is unfortunate because we think that a properly designed exception management model can also be useful for handling non-problematic, if exceptional, behavior. The exploration of this idea is one first aspect of this project. In fact, we also think that this idea can even go further, in the sense that exceptions do not even need to be *exceptional* (in the sense of not occurring frequently). In other words, applying language-level support for exceptions to normal and even frequent behavior is worth revisiting.

A second limitation of the classical model is its 2D-only SOC. As mentioned earlier, language-level support for exception handling increases the SOC, by clearly separating the code that may throw exceptions from the code that will handle them. Unfortunately, there is one more step towards orthogonality that the classical model misses. In this model, the `catch` part actually does two different things: catching the exception, and handling it locally. Again, this lack of orthogonality is unfortunate for two reasons. First of all, the place where an exception is caught should not necessarily be the place where it is handled. Secondly, apart from the catching location, there could even be *several* places where the exception could be handled, maybe in different ways. By making this additional distinction, that is, by indicating catching places and handling places separately, we achieve one more degree of SOC in our exception handling model.

The third limitation, arguably a corollary to the second one, is related to stack management. In the traditional model, when an exception is caught, the stack is effectively unwound to that particular location. Such systems are often said to have *termination semantics* (Shalit et al., 1996). Unwinding the stack means that a large part of the dynamic execution context is lost. This context could nonetheless contain useful information for further handling of the caught exception. On the other hand, if we preserve the entirety of the stack when an exception is thrown and caught (*a.k.a. calling semantics*), we can potentially establish handlers anywhere between those two points, hence "propagating" the exception back down the stack. That is why this

limitation is in fact the most important of the three, as lifting it will make it much easier to find interesting applications of a 3D separation of concerns.

The Common Lisp *condition system* (Seibel, 2005, chapter 19) is probably the most prominent advocate of a stack-preserving, 3D SOC exception management system (the one in Dylan was modelled after it). Although nothing is published yet, we have started to explore the idea of using it for applications unrelated to error (or even just exception) management, and in accordance with our initial intuition, we already have some preliminary results. In particular, we have demonstrated that it can be used, in a somewhat unexpected fashion, to emulate a (still) rudimentary form of coroutines, a very old and useful paradigm which suffered from a general disinterest before surfacing again in languages such as Lua or Go. As Common Lisp lacks native coroutines support, the idea is worth exploring. In some sense, by handling coroutine return values as exceptions, we use an error management system for the exact opposite of its original goal: to handle events which are neither errors, nor exceptional.

9.2 Future Work

To the best of our knowledge, the idea of a condition-based coroutine support is novel. How far we can really go with it remains unclear, however. What we currently have is more or less the equivalent of CLU iterators (Liskov et al., 77): our pseudo-coroutines are not first-class objects, and they can be invoked only one at a time. On the other hand, we have not used the Common Lisp condition system to its full power yet, and we already have a number of paths to explore. Here are the most important ones.

Our prototype currently uses a single condition, called `yield`, to emulate the return value of a coroutine. Symmetric coroutines have the ability to decide where they want to transfer control to, which we could reproduce by installing different simultaneous handlers for a whole hierarchy of subconditions. A coroutine can then decide where it wants to yield by signalling a specific sub-class of `yield`. Note that it has been demonstrated that full asymmetric and symmetric coroutines can be expressed in terms of each other (de Moura and Ierusalimschy, 2009).

One feature of the Common Lisp condition system that we haven't used yet is the ability for a handler to *decline* handling a condition. A handler is considered having handled a condition when it performs a non-local transfer of control (such as invoking a restart). However, if a handler returns normally, it is said to have declined, and the search for another handler continues. Although somewhat of a "dirty trick", nothing prevents a handler from performing side-effects before declining (such a handler would in fact only *pretend* it declined). Now suppose that several such handlers are active at the same time. What we obtain is essentially a coroutine, *broadcasting* its yielded values to several callers simultaneously. Although this is definitely not a normal property of actual coroutines, we can already see a potential benefit, notably in terms of composability.

Another feature that we haven't explored yet is the ability to invoke a restart with arguments. Yielding and restarting with arguments establishes a full two-way communication between coroutines and their callers, and could potentially be used for more complex dispatch schemes than what we have today.

Finally, our current prototype, along with its accompanying examples, uses the `restart-case` macro only. Just like `handler-case`, this macro imposes some stack

unwinding: when a restart is invoked, the result of its function is automatically used as the return value of the whole `restart-case`. On the other hand, a lower level construct, `restart-bind` (analogous to `handler-bind`) does not behave like that: the function of a restart will return normally. The implications of using `restart-bind` to our condition-based coroutine implementation remain to be explored.

At some point, we will obviously want to publish our results, and contrast them with existing literature. In particular, as far as coroutine emulation is concerned, we need to contrast our ideas with well know techniques such as using continuations or threads with channels and mailboxes, both in terms of expressivity and performance. Note that this work is likely to be of interest only in the particular case of Common Lisp, as continuations are not very well supported either. Coroutine emulation is much easier with native continuations, such as in Scheme or Racket. Previous work on continuation support through exceptions (Sekiguchi et al., 2001) may also help us and needs to be investigated.

We previously mentioned that the exception handling system in Dylan was modelled after that of Common Lisp. The intent, however, was not only to copy it, but to *improve* it. In particular, and contrary to Common Lisp, the Dylan restarts are themselves implemented in terms of conditions. We currently don't know the implications of this in terms of expressivity.

Finally, coroutine emulation is just the first, somewhat unusual, application of the condition system that occurred to us. It remains to be seen whether a more general concept of "Exception-Oriented Programming" is worth pursuing.

Chapter 10

Conclusion

In this report, we have exposed our original motivations, our general academic research strategy, and we have given a brief overview of height more-or-less distinct research projects, all but one related to the performance and expressivity of (dynamic (programming paradigms)). We hope that the reader was teased enough to read the complete literature we produced on the aspects that he or she has found interesting. In the case the reader is a potential Ph.D. candidate, we are looking forward to future collaboration on any part of the material presented in this report.

Several aspects of our work are either slightly less academic, or more marginal, and hence were not deemed worthy of a full chapter. We still feel compelled to mention them here.

In addition to the one successfully defended Ph.D. that we mentioned several times already, another one lasted a year, but was interrupted due to the student resigning. The purpose of this Ph.D. was the development of a new system for statistical analysis of financial flows, with interactivity, efficiency, and extensibility in mind. That Ph.D. was set up in partnership with a private financial investment company, which is still interested in reviving our aborted collaboration.

Our work has sometimes "diverged" towards horizons broader than the sole field of programming paradigms, in the form of more transversal research, and published as essays rather than technical papers. We already mentioned one such essay (Verna, 2018a). The reader interested in, or at least a bit curious about the field of biology may also enjoy reading our thoughts on the biological aspects of software evolution, as exposed in Verna (2010a)[1] and Verna (2011a)[2].

In line with the third axis of our academic strategy (contributing back to our main community), but in terms of pure engineering rather than academic research, let us quickly mention a number of activities that still led to some form of publication. In particular, we have 3 so-called "CDRs", that is, proposals for modifications, clarifications, or extensions to the Common Lisp standard (Verna, 2011b,c, 2012b). Quickref, a global documentation repository of generated reference manuals for Lisp libraries is also a notable achievement (Verna, 2019b,a).

Finally, we wish to express our satisfaction in view of the success of the European Lisp Symposium, which we helped create 13 years ago, and has become since then the major academic event for the community, with an ACM status.

1 Verna, D. (2010a). Classes, styles, conflicts: the biological realm of LaTeX. In Beeton, B. and Berry, K., editors, *TUG'10, 31st TeX Users Group Conference*, volume 31, pages 162–172. TeX Users Group.

2 Verna, D. (2011a). Biological realms in computer science. In *Onward!'11: the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software Proceedings*, pages 167–176. ACM.

Appendix A

Personal Bibliography

Newton, J. E., Demaille, A., and Verna, D. (2016). Type-checking of heterogeneous sequences in Common Lisp. In *ELS'16, 9th European Lisp Symposium*, pages 13–20, AGH University of Science and Technology, Krakow, Poland.

Newton, J. E. and Verna, D. (2018a). Recognizing hetergeneous sequences by rational type expression. In *Meta'18, Meta-Programming Techniques and Reflection Workshop*, Boston, MA, USA.

Newton, J. E. and Verna, D. (2018b). Strategies for typecase optimization. In *ELS'18, 11th European Lisp Symposium*, pages 23–31, Marbella, Spain.

Newton, J. E. and Verna, D. (2019a). Finite automata theory based optimization of conditional variable binding. In *ELS'19, 12th European Lisp Symposium*, pages 26–33, Genova, Italy.

Newton, J. E. and Verna, D. (2019b). A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. *ACM Transactions on Computational Logic*, 20(1).

Newton, J. E., Verna, D., and Colange, M. (2017). Programmatic manipulation of Common Lisp type specifiers. In *ELS'17, 10th European Lisp Symposium*, pages 28–35, Vrije Universiteit Brussel, Belgium.

Senta, L., Chedeau, C., and Verna, D. (2012). Generic image processing with Climb. In *ELS'12, 5th European Lisp Symposium*, Zadar, Croatia.

Valais, L., Newton, J. E., and Verna, D. (2019). Implementing baker's `subtypep` decision procedure. In *ELS'19, 12th European Lisp Symposium*, pages 12–19, Genova, Italy.

Verna, D. (2006). Beating C in scientific computing applications. In *ELW'06, 3rd European Lisp Workshop*, Nantes, France.

Verna, D. (2008). Binary methods programming: the CLOS perspective. *Journal of Universal Computer Science*, 14(20):3389–3411.

Verna, D. (2009). CLOS efficiency: Instantiation. In *ILC'09 International Lisp Conference*, pages 76–90, MIT, Cambridge, Massachusetts, USA. ALU (Association of Lisp Users).

Verna, D. (2010a). Classes, styles, conflicts: the biological realm of LaTeX. In Beeton, B. and Berry, K., editors, *TUG'10, 31st TeX Users Group Conference*, volume 31, pages 162–172. TeX Users Group.

Verna, D. (2010b). Revisiting the visitor: the just do it pattern. *Journal of Universal Computer Science*, 16(2):246–271.

Verna, D. (2011a). Biological realms in computer science. In *Onward!'11: the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software Proceedings*, pages 167–176. ACM.

Verna, D. (2011b). Clarification proposal for CLHS 22.3. Common Document Repository #7.

Verna, D. (2011c). File-local variables. Common Document Repository #9.

Verna, D. (2012a). Extensible languages: Blurring the distinction between DSLs and GPLs. In Mernik, M., editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 1. IGI Global.

Verna, D. (2012b). Standard output streams default behavior in terminal sessions. Common Document Repository #11.

Verna, D. (2012c). Star TeX: the next generation. In Beeton, B. and Berry, K., editors, *TUG'12, 33rd TeX Users Group Conference*, volume 33. TeX Users Group.

Verna, D. (2013). The incredible tale of the author who didn't want to do the publisher's job. In Beeton, B. and Berry, K., editors, *TUG'13, 34th TeX Users Group Conference*, volume 34. TeX Users Group.

Verna, D. (2018a). Lisp, Jazz, Aikido. *The Art, Science and Engineering of Programming Journal*, 2(3).

Verna, D. (2018b). Method combinators. In *ELS'18, 11th European Lisp Symposium*, pages 32–41, Marbella, Spain.

Verna, D. (2019a). Parallelizing Quickref. In *ELS'19, 12th European Lisp Symposium*, pages 89–96, Genova, Italy.

Verna, D. (2019b). Quickref: Common Lisp reference documentation as a stress test for Texinfo. In Beeton, B. and Berry, K., editors, *TUG'19, 40th TeX Users Group Conference*, volume 40, pages 119–125. TeX Users Group, TeX Users Group.

Verna, D. (2020). Dynamic programming paradigms: Performance and expressivity. Habilitation Report. Sorbonne Université.

Verna, D. and Ripault, F. (2015). Context-oriented image processing. In *COP'15, Context-Oriented Programming Workshop*.

Appendix B

Third-Party Bibliography

Akers, S. B. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516.

America, P. (1987). Inheritance and subtyping in a parallel object-oriented language. In Bézivin, J., Hullot, J., Cointe, P., and Lieberman, H., editors, *European Conference on Object Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242, Paris, France. Springer.

America, P. (1991). Designing an object-oriented programming language with behavioural subtyping. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages*, pages 60–90, Berlin, Heidelberg. Springer Berlin Heidelberg.

Andersen, H. R. (1999). An introduction to binary decision diagrams. Technical report, Course Notes on the WWW.

Baker, H. G. (1992). A decision procedure for Common Lisp's SUBTYPEP predicate. In *Lisp and Symbolic Computation*.

Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. (1988). Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142.

Bolz, C. F., Cuni, A., and Fijalkowski, M. (2009). Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOLPS*. ACM Press.

Boreczky, J. and Rowe, L. A. (1994). Building Common Lisp applications with reasonable performance. `http://bmrc.berkeley.edu/research/publications/1993/125/Lisp.html`.

Brace, K. S., Rudell, R. L., and Bryant, R. E. (1990). Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45.

Bravenboer, M., Kalleberg, K., Vermaas, R., and Visser, E. (2006). Stratego/XT 0.16: Components for transformation systems. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 95–99. ACM SGPLAN.

Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T., and Pierce, B. C. (1995). On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691.

Bryant, R. E. (2018). Chain reduction for binary and zero-suppressed decision diagrams. In Beyer, D. and Huisman, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–98, Cham. Springer International Publishing.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *J. ACM*, 11(4):481–494.

Buschmann, F., Meunier, R., Rohnert, H., P.Sommerlad, and Stal, M. (1996). *Pattern-Oriented Software Architecture*. Wiley.

Castagna, G. (1995). Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447.

Castagna, G. (1996). *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser.

Castagna, G. (2016). Covariance and contravariance: a fresh look at an old issue. Technical report, CNRS.

Castagna, G., Ghelli, G., and Longo, G. (1992). A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, 5(1):182–192.

Chambers, C. (92). Object-oriented multi-methods in cecil. In *European Conference on Object Oriented Programming*, pages 33–56.

Clarke, E. M., McMillan, K. L., Zhao, X., Fujita, M., and Yang, J. (1997). Spectral transforms for large boolean functions with applications to technology mapping. *Formal Methods and Systems Design*, 10(2–3):137–148.

Colange, M. (2013). *Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems*. Thèse de doctorat, Laboratoire d'Informatique de Paris VI, Université Pierre-et-Marie-Curie, France.

Costanza, P., Herzeel, C., Vallejos, J., and D'Hondt, T. (2008). Filtered dispatch. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, page 4.

Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS'05, pages 1–10, New York, NY, USA. ACM.

Costanza, P., Hirschfeld, R., and Meuter, W. D. (2006). Efficient layer activation for switching context-dependent behavior. In *JMLC'06: Proceedings of the Joint Modular Languages Conference*, pages 84–103. Springer.

Czarnecki, K., O'Donnell, J., Striegnitz, J., and Taha, W. (2004). DSL implementation in MetaOCaml, Template Haskell, and C++. In Lengauer, C., Batory, D., Consel, C., and Odersky, M., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin / Heidelberg.

de Moura, A. L. and Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6:1–6:31.

Dean, J., DeFouw, G., Grove, D., Litvinov, V., and Chambers, C. (1996). Vortex: an optimizing compiler for object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 83–100.

DeMichiel, L. G. and Gabriel, R. P. (1987). The common lisp object system: An overview. In *European Conference on Object Oriented Programming*, pages 151–170.

Denert, E., Ernst, G., and Wetzel, H. (1975). Graphex68 graphical language features in algol 68. *Computers & Graphics*, 1(2-3):195–202.

Elliott, C. (1999). An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25:291–308.

Ernst, M. D., Kaplan, C. S., and Chambers, C. (1998). Predicate dispatching: A unified theory of dispatch. In *European Conference on Object Oriented Programming*, pages 186–211, Brussels, Belgium.

Fateman, R. J., Broughan, K. A., Willcock, D. K., and Rettig, D. (1995). Fast floating-point processing in Common Lisp. *ACM Transactions on Mathematical Software*, 21(1):26–62. Downloadable version at `http://openmap.bbn.com/~kanderso/performance/postscript/lispfloat.ps`.

Fortran (1977). American National Standard x3.9-1978.

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`.

Fowler, M. (2010). *Domain Specific Languages*. Addison Wesley.

Gabriel, R. P. (1985). *Performance and Evaluation of Lisp Systems*. MIT Press.

Gabriel, R. P., White, J. L., and Bobrow, D. G. (1991). CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Géraud, T. and Levillain, R. (2008). Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus.

Ghosh, D. (2010). *DSLs in Action*. Manning Publications.

Ghosh, D. (2011). DSL for the uninitiated – domain-specific languages bridge the semantic gap in programming. *ACM Queue*, 9(6).

Goodenough, J. B. (1975). Structured exception handling. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'75, pages 204–224, New York, NY, USA. ACM.

Google (2018). The Go language specification. https://golang.org/ref/spec.

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., and Smith, D. (2019). *The Java Language Specification*. Oracle America, Inc.

Graham, P. (1993). *On Lisp*. Prentice Hall.

Gröpl, C., Prömel, H. J., and Srivastav, A. (1998). Size and structure of random ordered binary decision diagrams. In *STACS 98*, pages 238–248. Springer Berlin Heidelberg.

Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology*, 7(3):125–151.

Hofer, C. and Ostermann, K. (2007). On the relation of aspects and monads. In *Proceedings of the 6th Workshop on Foundations of Aspect-oriented Languages*, FOAL '07, pages 27–33, New York, NY, USA. ACM.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition edition.

Hromkovič, U. (2002). Descriptional complexity of finite automata: Concepts and open problems. *Journal of Automata, Languages, and Combinatorics*, 7(4):519–531.

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ierusalimschy, R. (2013). *Programming in Lua*. lua.org, Université Pontificale Catholique de Rio de Janeiro, 3rd edition edition.

ISO/IEC (2011). C international Standard. ISO/IEC 9899:2011.

ISO/IEC (2017). C++ international standard. ISO/IEC 14882:2017 (E).

Kamin, S. N. (1998). Research on domain-specific embedded languages and program generators. In *Electronic Notes in Theoretical Computer Science*, volume 14. Elsevier.

Karpinski, S. and Bezanson, J. (2019). *The Julia Language*. The Julia Project.

Kay, A. (2003). Dr. Alan Kay on the meaning of "Object-Oriented programming". Email to Stefan Ram.

Kay, A. C. (1969). *The Reactive Engine*. PhD thesis, University of Hamburg.

Kay, A. C. (1993). The early history of smalltalk. In *Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 69–95, New York, NY, USA.

Keene, S. E. (1989). *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *European Conference on Object Oriented Programming*. SpringerVerlag.

Kiczales, G. J., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

Kiselyov, O. (2014). The design and implementation of BER MetaOCaml. In Codish, M. and Sumii, E., editors, *Functional and Logic Programming*, pages 86–102. Springer International Publishing.

Kiselyov, O. and chieh Shan, C. (2009). Embedded probabilistic programming. In Taha, W., editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, pages 360–384. Springer Berlin / Heidelberg.

Klabnik, S. and Nichols, C. (2018). *The Rust Programming Language*. No Starch Press.

Knuth, D. E. (2009). *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition.

Lai, Y. T. and Sastry, S. (1992). Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC'92, pages 608–613, Los Alamitos, CA, USA. IEEE Computer Society Press.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9:157–166.

Levillain, R., Géraud, T., and Najman, L. (2010). Why and how to design a generic and efficient image processing framework: The case of the Milena library. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pages 1941–1944, Hong Kong.

Lieberman, H. (1986). Using prototypical objects to implement shared behavior in object oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, Portland, OR, USA.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. (77). Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576.

Liskov, B. and Wing, J. (1994). A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems*.

Lozhkin, S. A. and Shiganov, A. E. (2010). High accuracy asymptotic bounds on the BDD size and weight of the hardest functions. *Fundamenta Informaticae*, 104(3):239–253.

Maes, P. (1987). Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*. ACM.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3:184–195. Online version at http://www-formal.stanford.edu/jmc/recursive.html.

McIlroy, D. M. (1960). Macro instruction extensions of compiler languages. *Communications of the ACM*, 3:214–220.

Mendhekar, A., Mendhekar, A., Kiczales, G., Kiczales, G., Lamping, J., and Lamping, J. (1997). RG: A case-study for aspect oriented programming. Technical report, Xerox Parc.

Minato, S. (1993). Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, DAC'93, pages 272–277, New York, NY, USA. ACM.

Neuss, N. (2003). On using Common Lisp for scientific computing. In *CISC Conference, LNCSE*. Springer-Verlag. Downloadable version at iwr.uni-heidelberg.de/groups/techsim/people/neuss/publications.html.

Nierstrasz, O. (2010). 10 things i hate about object oriented programming. ECOOP Conference Dinner Speech.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 3rd edition.

Norvig, P. (1996). Tutorial on design patterns in dynamic programming. In *Object World Conference*.

Odersky, M., Spoon, L., and Venners, B. (2010). *Programming in Scala*. Artima.

Oliveira, B. (2009). The different aspects of monads and mixins. Draft Paper. Last Update: 04/03/2009. Submitted to ICFP 2009.

Oracle (2019). Java tutorials: Exceptions. https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html.

Owens, S., Reppy, J., and Turon, A. (2009). Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190.

Paepcke, A. (1993). User-level language crafting – introducing the CLOS metaobject protocol. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press. Downloadable version at http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

Pagan, F. (1979). Algol 68 as a meta-language for denotational semantics. *The Computer Journal*, 22(1):63–66.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

POSIX (2018). International Standard: Portable Operating Systems Interface. ISO/IEC 9945:2018.

Quam, L. H. (2005). Performance beyond expectations. In *International Lisp Conference*, pages 305–315, Stanford University, Stanford, CA. The Association of Lisp Users. Downloadable version at ai.sri.com/~quam/Public/papers/ILC2005/.

Reid, J. (1996). Remark on "fast floating-point processing in Common Lisp". In *ACM Transactions on Mathematical Software*, volume 22, pages 496–497. ACM Press.

Ruby (2019). Ruby documentation: Class Exception. `https://docs.ruby-lang.org/en/2.7.0/Exception.html`.

Seibel, P. (2005). *Practical Common Lisp*. Apress, Berkeley, CA, USA. Online version at `gigamonkeys.com/book/`.

Sekiguchi, T., Sakamoto, T., and Yonezawa, A. (2001). *Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling*, chapter 14, pages 217–233. Springer Berlin Heidelberg, Berlin, Heidelberg.

Shalit, A., Moon, D., and Starbuck, O. (1996). *The Dylan Reference Manual*. Apple Press, Addison-Wesley.

Shannon, C. E. (1949). The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98.

Siek, J. and Taha, W. (2007). Gradual typing for objects. In Ernst, E., editor, *European Conference on Object Oriented Programming*, pages 2–27, Berlin, Heidelberg. Springer.

Siek, J. G. and Taha, W. (2006). Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92.

Smaragdakis, Y. and Batory, D. (2001). Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs.

Smith, B. C. (1984). Reflection and semantics in Lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM.

Sniedovich, M. (2010). *Dynamic Programming: Foundations and Principles*. Chapman & Hall/CRC Pure and Applied Mathematics. CRC Press, 2nd edition edition.

Somenzi, F. (2016). CUDD: BDD package. University of Colorado.

Srinivasan, A. (2002). Algorithms for discrete function manipulation. In *IEEE International Conference on Computer-Aided Design, 1990. Digest of Technical Papers.*

Strehl, K. and Thiele, L. (1998). Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD'98, pages 686–692, New York, NY, USA. ACM.

Taha, W. and Sheard, T. (1997). Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '97, pages 203–217, New York, NY, USA. ACM.

Tarver, M. (2015). *The Book of Shen*. FastPrint Publishing, 3rd edition edition.

Ansi (1994). American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999).

Thomas, D., Fowler, C., and Hunt, A. (2009). *Programming Ruby*. Pragmatic Bookshelf.

Tobin-Hochstadt, S. and Felleisen, M. (2006). Interlanguage migration: from scripts to programs. In *Conference proceedings on Object-oriented programming systems, languages and applications*.

Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., and Felleisen, M. (2011). Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Tratt, L. (2005). The converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College, London, UK.

Tratt, L. (2008). Domain-specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems*, 30:31:1–31:40.

Ucko, A. M. (2001). Predicate dispatching in the Common Lisp Object System. Technical Report 2001-006, MIT, Cambridge, MA, USA.

Ungar, D., Chambers, C., Chang, B.-W., and Hölzle, U. (91). Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3).

Ungar, D. and Smith, R. B. (1987). Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–241, Orlando, FL, USA.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35:26–36.

van Rossum, G. (2012). *The Python Language Reference*. Python Software Foundation.

Vasudevan, N. and Tratt, L. (2011). Comparative study of DSL tools. *Electronic Notes in Theoretical Computer Science*, 264:103–121.

Warren, H. S. (2002). *Hacker's Delight*. Addison Wesley Professional. `http://www.hackersdelight.org`.

Weitz, E. (2015). *Common Lisp Recipes: a Problem-Solution Approach*. Apress.

Wyk, E. V., Bodin, D., Krishnan, L., and Gao, J. (2008). Silver: an extensible attribute grammar system. In *Electronic Notes in Theoretical Computer Science*, pages 103–116. Elsevier.

Xing, G. (2004). Minimized Thompson NFA. *International Journal of Computational Mathematics*, 81(9):1097–1106.

Acknowledgments

First of all, I am very grateful to the members of my jury, Marco Antoniotti, Ralph Möller, and Gérard Assayag for their participation, along with Robert Strandh, Nicolas Neuß and Manuel Serrano, for having also accepted to review this report (special mention to Robert Strandh for his extensive annotations).

I wish to thank all the colleagues I had the chance to meet and work with over the years, and all the students that endured me as a supervisor. They are all too numerous to list nominatively, but they have my utmost gratitude.

Finally, let me give a special mention to Jim "*I refuse to accept*" Newton, my first successful Ph.D. student. It has been a pleasure working with him, and his work has been a key factor in triggering my habilitation process. It's not everyday that you get to supervise a student who is actually older than you (let alone coming from Mississippi).

One day, Thierry "*théo*" Géraud founded the EPITA Research and Development Laboratory (LRDE). One year later, he put my CV on top of a pile in the EPITA director's office. Since then, both the school and the lab have provided the conditions (human, logistical, and financial), that made all this research possible, something that I felt I had to mention as well.

Being able to blossom in your work is not *only* a luxury. It does not depend only on luck, on the persons you meet, or on the working conditions you get. It also depends on *introspection*, that is, on your ability to know yourself, and seek harmony between who you are and what you do. Because of that, I have to end this section by paying a tribute to John McCarthy, inventor of Lisp, O Sensei Morihei Ueshiba, founder of Aikido, and all the extraordinary musicians that made Jazz the music it is today.