MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ȘI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Improving Flexibility and Efficiency in Programming Languages

a Natural Approach

LUCIAN RADU TEODORESCU

A thesis presented for the degree of
Doctor in Philosophy

Scientific advisor: Prof.dr.eng. Rodica Potolea

Thesis committee:

President:	Prof.dr.eng. Liviu Miclea – dean, Faculty of Computer Science and Automation, Technical University of Cluj-Napoca
Members:	Prof.dr.eng. Rodica Potolea – scientific advisor, Technical University of Cluj-Napoca
	Prof.dr.eng. Nicolae Țăpuș - reviewer, University Politehnica of Bucharest
	Prof.dr.eng. Vasile Manta – reviewer, Technical University Gheorghe Asachi of Iași
	Prof.dr.eng. Ioan Salomie – reviewer, Technical University of Cluj-Napoca

Computer Science Department
Technical University of Cluj-Napoca

July 2015

To my wife, Adriana

Acknowledgments

First, I would like to express my gratitude to my thesis supervisor, Rodica Potolea for all the guidance, support, and patience she offered me throughout the research period. This thesis would not have been possible without her constant belief in this idea—even when I had doubts. I would also like to thank my Diploma supervisor, Alin Suciu, who first encouraged me to pursue the idea of creating a programming language, and for all the great advice he gave me during the early years of the language.

Many thanks go to my friend Vlad Dumitrel, who was a constant support in my efforts of creating a programming language. He designed and implemented the first version of the standard library. Moreover, our countless long discussions provided me with the required lucidity for the design and evolution of the language. Also, his efforts of reviewing all the written material and providing feedback was an essential support for writing this thesis.

I would also like to thank my parents Dana and Mircea for all their love and support; during so many years they did everything in their power in order for me to focus on my research.

Finally, I would like to thank my wife Adriana. Even if she does not speak *computerese*, she was always my role model and my inspiration for all the research activities. By the power of example, she taught me what sound judgment means, what passion, hard-work, and excellence are all about. She was always there when I needed somebody.

Abstract

Programming languages lay the foundation for the computer science research field. A major breakthrough in programming languages typically has a major impact on the world of computing. Hall, Padua, and Pingali argue that the biggest accomplishment in computer science is the widespread adoption of high-level languages [HPP09].

There are three main directions that define the landscape of programming languages: efficiency, flexibility, and naturalness. Efficiency refers to the ability to generate machine code that executes as fast as possible with minimal resource usage. Flexibility refers to the versatility of programming languages, and to the possibility of using them to solve problems from various domains, using various techniques. Lastly, naturalness refers to the ease of writing, understanding, and reasoning about a code written in a particular language.

A major problem that the field of programming languages is facing is that these directions are often in opposition to one another. For example, the more focused a language is on efficiency, the less natural it tends to be, and vice-versa. The ultimate programming language would be one that succeeds to fully align these three directions.

This thesis proposes a new programming language, Sparrow, that strives to increase efficiency, flexibility, and naturalness. Our approach starts with these directions as language principles, and works out the design of the language in order to ensure that they are fully satisfied. To substantiate our claims, we have implemented a compiler for the language and provided a series of case studies that show that Sparrow is indeed an efficient, flexible, and natural language.

One of the most important features of the language is hyper-metaprogramming, a form of static metaprogramming that allows the programmer to execute arbitrarily complex computations at compile-time. This feature can act as an extension to the compiler, allowing the programmer to interact with the language, and enrich it with new capabilities.

The language is carefully designed to maintain the same efficiency level as C++. The compiler does not generate code that introduces efficiency penalties, without the programmer using a feature that actually requires that code to be generated. Our experiments show that Sparrow has the same level of efficiency as C++. Moreover, we propose a new technique for moving computations from run-time to compile-time, that can greatly improve the speed of programs. The technique relies on hyper-metaprogramming to execute the computations at compile-time. Our experimental results show speedups of up to 80 in the case of minimal perfect hashing, and over 12 in the case of regular expressions.

Running computations at compile-time in Sparrow enjoys the same expressive power as run-time computations. One can run complex algorithms, read and write files from disk, interact with the operating system, and even influence the compilation processes. Moreover, the execution speed of algorithms at compile-time is almost the same as for run-time algorithms in an interpreted environment.

During the creation of the language, we put a lot of effort into making sure that the language is natural, even when writing highly efficient code. Our flexible operator system and our generic programming feature are two of the main ways of ensuring that the code is easy to write and understand. In all of our efficiency case studies we show that our code is more natural than similar codes written in other programming languages. For example, we show that the same loop-recognition algorithm is easier to write in Sparrow than in C^{++}, Go, Java, and Scala, while also being faster. In the case of moving computations to compile-time, we can obtain speedups without the user noticing any changes in the interface that needs to be called.

Probably the best showcase of Sparrow flexibility is allowing the user to embed other programming languages in Sparrow. No changes to the compiler are required; adding a new language can be done simply as a library feature. We exemplify this process by embedding a subset of Prolog into Sparrow. Expressing domain-specific problems in a familiar dedicated language can also be a boost in naturalness. We believe that the programmer should be allowed to express computations in the programming paradigm that is most appropriate for the problem being solved and for the programmer himself.

This thesis demonstrates that it is possible to create a language that aligns efficiency, flexibility, and naturalness. It is not necessary for them to exclude one another; on the contrary, they can function together in synergy to create a powerful programming experience.

Contents

Who will love a little Sparrow?
Who's traveled far and cries for rest?

Simon & Garfunkel

Chapter 1
Introduction

The human language is not only the most complex tool for communication, as the neo-positivist movement would like us to believe, but also a social and historical power that shapes our minds and contributes in a definitive way in configuring our sense of reality. Wilhelm von Humboldt (1767-1835), the father of Comparative Linguistics and of the Philosophy of Language, accomplished an epistemological revolution when he pointed out that different languages (French, English, etc.) organize the human mind and its representation of reality in different ways. For example, the mental image that an English man has when thinking of/referring to/speaking about a *"snowdrop"* is different from the image created by the analogous French word *"perce-neige"*, which would be literally translated into *perforate the snow*. The first image is a static, stable image—a drop of snow—, while the second implies an action, focusing on the idea of movement.

Recent linguistic research fostered Humboldt's perspective. The father of Integral Linguistics, Eugeniu Coșeriu, insisted upon the triadic dimension of the language [Cos00]. He argues that any language has three intertwined and inter-related levels: the universal level (the level of reference/designation), the historical level (of signification), and last but not least, the individual level (the level of meaning). If the first two levels are somewhat reminiscent of a *product*, because they already exist, prior to any *historical* speaker[1], the third is comparable to a creative energy that perpetuates, and at the same time enhances the historical languages.

One of the basic consequences of such a paradigmatic shift is the fact that

[1] true if we realize that dictionaries and books precede man's coming into the world, considering that every birth takes place within a language

knowing and being able to use more languages enriches and increases the number of meanings one could get from living in a single world. On the other hand, one must acknowledge that sometimes challenging a language, in one respect or another, from one perspective or another, is not a sign of disrespect but rather a natural impulse of creativity.

The same goes for programming languages. In spite of the popular narrow perception according to which they are merely series of symbols and rules, programming languages actually mold the reality of the computer science domain (the programmer's expectations, behaviors, abilities, resulting products, etc.).

For example, to create a paycheck system, an object-oriented programming language would model it as a set of inter-related objects with *messages* sent between them; still, the same system would be implemented in a functional programming language as a set of pure functions that will always create new instances of the data for each processing.

Translating Coșeriu's triadic dimension of the language in computer science terms, we can associate the universal level with the actual computations that a machine would perform (machine executable code). The historical level of the language would correspond to the existing practice of programming in that language, with all the corresponding conventions, patterns, and idioms. Finally, the individual level of a program corresponds to how a programmer would read, understand, and relate to that specific program.

This thesis is all about creating a new programming language—the Sparrow programming language. The three main principles of the language, envisioned at the start of the development, are: efficiency, flexibility, and naturalness. Efficiency governs the connection between the programs in the language and the machine code—the universal level. The more natural the programming language is, the easier it is for the programmer to learn the language and relate to it. Its flexibility is responsible for allowing the programmers to express their creativity in the language, constantly changing and evolving the language.

1.1 The idea for Sparrow

The author has worked as a professional C++ programmer since 2002, a few years before Sparrow was born. Compared to other programming language, C++ has a suitable combination of low-level abstractions and high-level abstractions. In the same language, one can write both very efficient code, and very complex and abstract code. Moreover, one can use complex abstractions to write elegant yet efficient code. A few examples in this respect can be found in [SLL01; GIL; SL98; Vel98].

Inspired by Alexandrescu's book [Ale01], the first research ambitions of the author aimed towards using metaprogramming and generic programming techniques in C++ to create highly-efficient, high-level, and reusable libraries.

The author started to create his own metaprogramming library, similar to the Boost MPL library [GA; AG04].

At this point the author started to realize that metaprogramming is difficult in C++. Although C++ templates are Turing-complete [Vel03], in practice performing computations at compile-time is usually not feasible. The compilation processes become very slow [AG04; Dub13], computations with strings and floating-point numbers are not possible, and the generated errors are long and hard to understand [Sie05; GJL+07; VJ02]. The template metaprogramming model is difficult to use in practice [Por10]; after all, C++ templates were not designed for metaprogramming [Str07]. Coupled with the complexity present in other C++ language features, we reached the conclusion that C++ is too complex.

By the end of 2005, we started thinking of a programming language similar to C++, but less complicated, while preserving the same basic functionalities. This was the point when the first idea for Sparrow was born. What started out as *a language like C++, with simpler metaprogramming*, soon evolved into a language that aimed to integrate efficiency, flexibility, and naturalness. Of course, C++ would be the inspiration for this new language, mainly due to its efficiency.

Throughout the years, the idea of Sparrow changed significantly. However, the three main principles remained intact. Moreover, the high interest in static metaprogramming also remained constant. What was merely an intuition in the early years, later became more and more certain: static metaprogramming can be the language feature that *glues* together efficiency, flexibility, and naturalness.

Most of the changes were driven by a constant effort to simplify the language as much as possible. Following Steele's ideas [Ste99] we desired for Sparrow to be a small language that can grow through extensions implemented as library features. A *minimal* language would imply an increase in naturalness, as the users would have to learn less to use the programming languages. Also, by providing the mechanism to build complex abstractions, the language becomes more flexible.

The reader can see this simplification pattern often throughout the thesis. For example, simplifying the rules for operators makes it possible to allow custom operators, and also to call functions using operator syntax; as chapters 6 to 8 show, this allows writing efficient code in a compact and simple manner. As another example, by using the same syntax for regular function parameters and generic parameters, we make all the rules that apply to functions to also apply to generics; for example, we can define operators on types (e.g., `@Int` is a call to a generic operator `@`), and we can instantiate generic classes using postfix operator calls (e.g., `Int Vector` instantiates the `Vector` generic with the `Int` parameter).

Probably the most fruitful simplification was to reduce static metaprogramming from a complex set of rules to the simple idea of being able to execute arbitrary computations at compile-time; the user would use the same syntax and the same programming style to create metaprograms, as well as regular run-time programs. This form of static metaprogramming is called *hyper-metaprogramming*.

In addition to allowing the writing of arbitrarily complex metaprograms to be run at compile-time, hyper-metaprogramming is also used as the fundamental feature in the language. Many language features that can be used at run-time are in fact based on hyper-metaprogramming. For example, adding a reference to a type is an operation on a type; this is defined as a compile-time function in the standard library. Moreover, generic programming—on which most of the Sparrow standard library is based—relies on hyper-metaprogramming to evaluate and handle generic parameters.

Using hyper-metaprogramming, one can *inject* into Sparrow programs written in other programming languages. Although this may sound very complicated to do, it is actually relatively simple in Sparrow. One would need to implement a parser, a semantic analyzer, and a code generator as a metaprogram in Sparrow. The source programs can be translated directly into Sparrow Abstract Syntax Trees (ASTs), and the compiler will generate optimized code for it. Chapter 11 describes how one can implement Prolog as an embedded language inside Sparrow.

Overall, we show that these design principles allow us to build an efficient, flexible, and natural programming language. We hope to convince the reader that the approach that we have taken here follows naturally from the design principles.

Objectives

The main objective of this thesis is to devise an approach of building programming languages that are able to integrate flexibility, efficiency and naturalness. In order to substantiate this objective, we endeavored to build such a programming language. Our approach needs to start from the basis of creating programming languages, questioning existing compiler practices in order to provide a new method of constructing programming languages.

As secondary objectives, we wanted to go beyond the state of the art of programming languages methods in all three directions: flexibility, efficiency and naturalness. An improvement made in any of these directions should not compromise the other two. The programming language that we develop needs to be more flexible than traditional programming languages. The language must also be efficient, at least as efficient as C and C++; our goal was also to find some methods of outperforming these languages. Lastly, we wanted to create a natural programming language that allows the user to easily write programs; again, we set a goal in exceeding the existing practice in terms of naturalness.

As another objective, we aim to improve the state of the art methods for compile-time metaprogramming (also called static metaprogramming), especially in imperative programming languages. We set a goal to simplify these methods, and to put static metaprogramming at the core of the language.

1.2 Claims and evaluation

The following list highlights the most important claims of this thesis, along with the methods to substantiate them:

1. The Sparrow programming language can be built from its design principles, with a proper compiler, and can be used for regular programming activities.
    - chapter 4 presents the design of the language, starting from design principles, and concluding with the actual mechanism used to build the language
    - chapter 5 presents a possible translation scheme for the language, describing how an actual compiler should be built according to the required principles
    - the rest of the chapters present a variety of problems that can be solved using the Sparrow programming language
    - appendix B presents the full source-code of a loop recognition algorithm in Sparrow
2. Sparrow is efficient.
    - chapter 8 shows that Sparrow is in the same class of efficiency as C++
    - chapter 9 presents how Sparrow can move entire computations to compile-time, thus achieving impressive speedups—almost 80 times faster for minimal-perfect-hashing and over 12 times faster for regular expressions
3. Sparrow is flexible.
    - chapter 6 presents how a simple set of rules allow an operators system in the language that is more flexible than other languages (e.g., C++, Scala, Haskell); the user can create custom operators, use function names as operators, and change the precedence and associativity of the operators
    - chapter 7 presents the generics feature of Sparrow; again, starting with a simple set of rules, Sparrow is able to generate a complex generic programming system matching and sometimes exceeding the capabilities of programming languages like C++, Java, Haskell, or ML
    - chapter 9 shows how hyper-metaprogramming can be used to move computations to compile-time, a feature that is not available in most languages

- chapter 11 shows how one can embed new programming languages inside Sparrow, and how one can interact with the compiler internals; the same chapter shows how Sparrow can add new programming paradigms to the language

4. Sparrow is natural (easy to use)
    - chapter 3 gives a short tutorial of the Sparrow language; there is no feature that is inherently hard to use in practice
    - chapter 7 presents the generic programming feature from Sparrow, one of the most used features; although it is more powerful than C++, it is easier to use
    - at the end of chapter 8 we present a complex benchmark, comparing C++, Go, Java, Scala, and Sparrow; we show that Sparrow allows writing less code than the other programming languages, while also generating the fastest binary
    - chapter 9 shows that moving computations to compile-time can be done while maintaining the same interface; this interface completely hides the fact that we are moving computations to compile-time
    - chapter 11 shows that other languages (DSELs) can be embedded inside Sparrow, allowing the final programmer to write code in the paradigm that most suits the problem to be solved
5. Hyper-metaprogramming can be implemented, and is useful
    - chapter 4 presents the design of hyper-metaprogramming and provides the means of implementing it in a programming language
    - chapter 9 shows how one can use hyper-metaprogramming to move computations to compile-time
    - chapter 7 shows how hyper-metaprogramming is used to implement **`if`** clauses and Sparrow concepts
6. Hyper-metaprogramming is feasible in practice; there are no major limitations, as in the case of the C++ template system
    - chapter 10 analyzes in depth the costs and implications of running programs at compile-time; it shows that executing programs at compile-time can be almost as efficient as executing the same program at run-time (using an interpreter)
    - chapters 9 to 11 presents examples in which hyper-metaprogramming is used to perform complex computations at compile-time (reading from files, running complex algorithms, parsing and type-checking Prolog programs, generating code, etc.)
7. One can move computations to compile-time to speed up programs
    - chapter 9 shows how one can use hyper-metaprogramming to move computations to compile-time, providing two real-world examples; we obtained impressive speedups
8. One can design languages with high degrees of efficiency, flexibility, and naturalness; hyper-metaprogramming can be used to *glue* these qualities

together
- the Sparrow programming language is designed to fulfill these goals (see chapter 4)
- the case studies chapters show that these three principles can work together, and that hyper-metaprogramming can facilitate them

1.3 Road map

Chapter 2 presents the state of the art in the field of programming languages, and the most relevant work with respect to our thesis. The rest of the thesis is divided three parts.

The first part of the thesis describes the Sparrow programming language. We start with a brief introduction to Sparrow (chapter 3) to make the reader familiar with the syntax and the main abstractions of the language. Chapter 4 presents the design of Sparrow; it outlines the principles that govern the language, introduces hyper-metaprogramming as the most important Sparrow feature, and finally it formalizes the mechanism of constructing the language. Chapter 5 provides a sketch of the translation to LLVM intermediate representation. Although semi-formal, this translation serves as the language definition. It also represents an implementation description of a Sparrow compiler.

The second part of the thesis consists of six case studies that analyze the most important language features and capabilities, showing how Sparrow can fulfill its goals. This part provides a qualitative analysis of Sparrow, comparing it to other programming languages.

Chapter 6 presents the operators system in Sparrow. It showcases how a simple set of rules can create a rich palette of interactions with the language; the operators system is also very important for language naturalness, as it allows expressing computations in a simple fashion, one that is easy to read and understand.

Chapter 7 presents Sparrow generics. This is one of the most highly used features from Sparrow. According to our design principles, we show again how a simple set of rules can create a flexible and natural feature. We show how Sparrow generics are easier to use than generics from C++, yet more powerful. We also discuss how *concepts* [GSJR06; SS12a] are implemented in Sparrow; once more, the rules for the concepts are simpler than in C++[2], while being able to be more flexible.

In chapter 8 we perform an analysis of the general efficiency of the Sparrow programming language. We show how one can easily encode problems using ranges[3], and yet produce efficient binaries. While the Sparrow encoding of the problem is smaller than in C++ (in some cases even smaller than the corresponding English description), its performance is equivalent to that of the

[2]at the time of writing this thesis, the concepts feature has yet to be included in the C++ standard; it is planned for inclusion in the 2017 version of the standard

C++ program. In the last part of the chapter we feature a more complex study. This compares the implementation of a loop recognition algorithm in Sparrow with equivalent implementations in C++, Go, Java, and Scala. The Sparrow implementation is the smallest implementation, while also being the fastest.

Chapter 9 presents a novel technique based on partial evaluation [JGS93] and hyper-metaprogramming, that moves computations from run-time to compile-time. If a program contains some computations that do not depend on run-time variables, then these computations can be executed at compile-time. Using this technique the programs can be sped up significantly.

As computations are executed at compile-time, hyper-metaprogramming can increase the compilation time. Chapter 10 provides an in depth analysis of the execution costs of hyper-metaprogramming. We show that it is more expressive than template metaprogramming in C++, and also that metaprograms execute much faster compared to the C++ versions. Moreover, we show that the execution of algorithms at compile-time is almost as efficient as executing the same algorithms at run-time, in an interpreted environment.

Chapter 11 shows how hyper-metaprogramming can be used to embed other languages inside Sparrow; we use a subset of Prolog as the language to be embedded. The presented mechanism allows any user to extend the language and the compiler from a library level. One can add support for new programming languages and new programming paradigms, without changing the Sparrow compiler.

The third part of the thesis presents the final notes on the Sparrow programming language. Chapter 12 presents a tentative future work, and chapter 13 concludes the dissertation.

[3] a library abstraction that refers to the general concept of a sequence of elements of the same type; ranges rely mostly on generic programming

Every new beginning comes from some other beginning's end.

Seneca

CHAPTER 2 STATE OF THE ART AND RELATED WORK

We present in this section some of the most important work that has influenced our thesis, and provide a brief description of the state of the art in the programming languages domain. Constructing a new general-purpose programming language touches on such a broad variety of research topics, that an exhaustive identification of the sources and influences would be, if not impossible, at least overwhelming for the reader. Therefore, we consciously endeavor to devote the chapter specifically to those works that contributed significantly to the creation of our programming language.

2.1 Programming languages

A Programming language is a set of lexical, syntactical, and semantic conventions used for describing computations, so that both people and machines understand them. Such computations are called a program. All humans and all the machines that are provided with a program need to have the same understanding of the program. There is a vast literature describing the theory and practice of programming languages. Some of the textbooks describing general concepts of programming languages are [Sco09; GJLB00; Ses12; Seb12; Bru02; Fin96; Rey09; Pie02]. A particularly interesting approach is taken by Biancuzzi and Warden in their *Masterminds of programming* book [BW09], as they interview the creators of major programming languages—the book provides many valuable insights into what it is like to design a programming language.

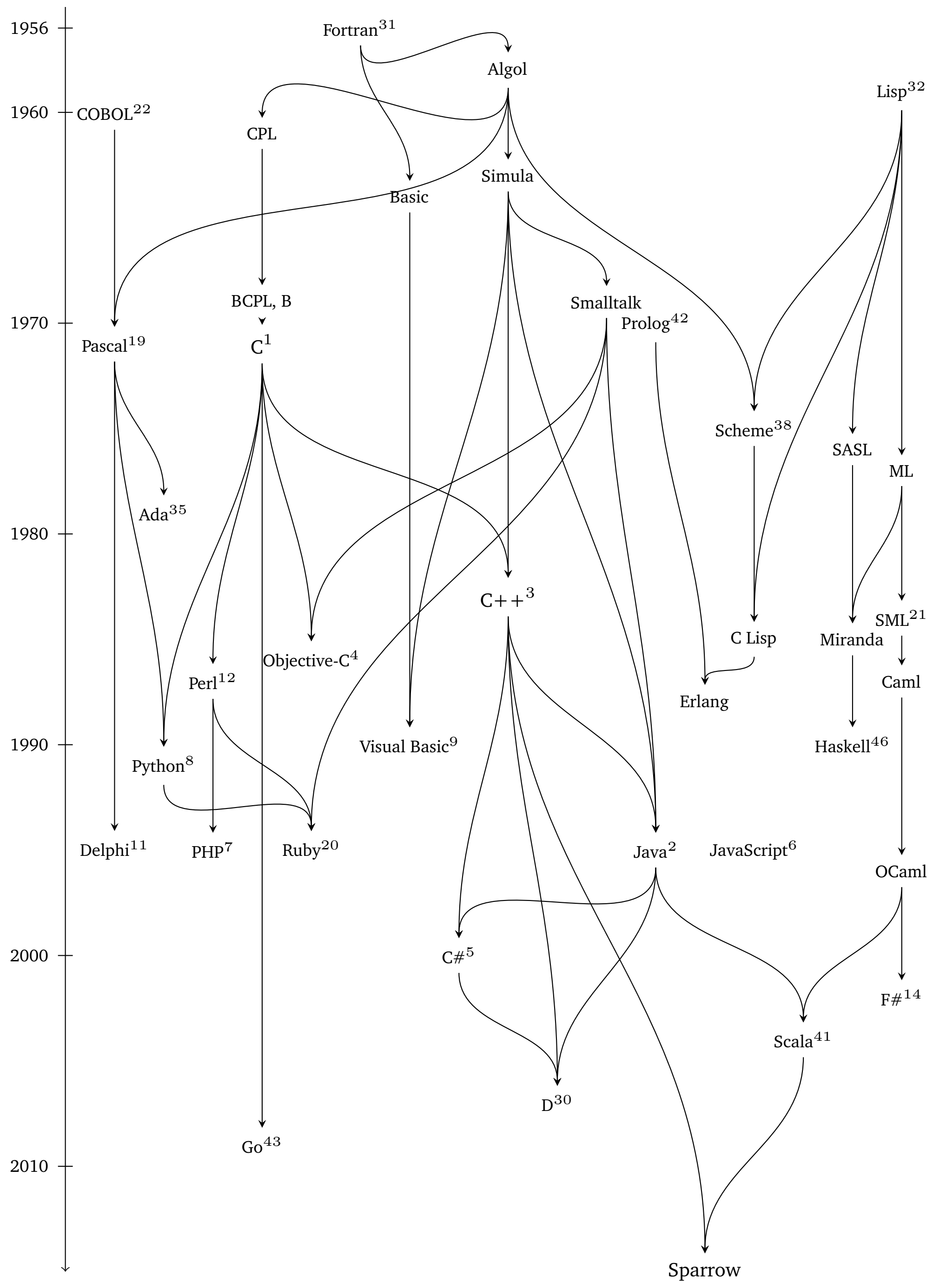


Figure 2.1: The genealogy of programming languages

A history of programming languages can be found in the *History of Programming Languages* series [Wex78; BG96; RH07]. Figure 2.1 presents a genealogy of some of the most important programming languages. For each programming language we presented the main influences, and also the ranking in the TIOBE programming languages index for February 2015 [Sof15]. The Sparrow programming language is also shown in the figure, so that the reader may easily identify the origins of the language.

From fig. 2.1 the user can see that there are two main archetypal programming languages: Fortran and Lisp. The vast majority of imperative programming languages are derived from Fortran and its successor, Algol. On the other hand, most functional languages are derived from Lisp. Despite being the oldest programming language, Fortran is still used today, and according to TIOBE index, it is ranked 31st, higher than other modern languages like Go (43rd), Scala (41st), and Haskell (46th). Lisp is one position lower in the same index.

Although C is one of the oldest programming languages, it is ranked number one in the TIOBE index. Along with C++ and Fortran, C is considered one of the most efficient languages. For all three languages one can predict what instructions are generated by the compiler for a given program. This is not true for functional languages (e.g., Haskell, F# or Scheme) or for interpreted languages (e.g., Perl, Python, Ruby). In functional languages, the emphasis is on type correctness and on a notion of *abstract mathematical beauty*. Interpreted languages try to focus on allowing the user to write programs quickly, and not on the correctness aspects of the programs.

In this thesis we often reference the C++ programming language—Sparrow is heavily influenced by its design. At this point there are 4 main versions of the C++ standard. C++ was standardized for the first time in 1998 [Cpp98]; this is the base standard for the language. In 2003 a minor standard revision was approved [Cpp03]; this did not introduce major features in the language. The next major release of the standard (known as C++ 0x) came out in 2011 [Cpp11]; this release added many features to the standard, like auto type inference, lambda functions, rvalue references, etc. At the end of 2014 a minor release of the standard was approved [Cpp14]; this release fixes some issues with the previous standard and adds some smaller language features (e.g., relaxed constexpr restrictions, generic lambdas, variable templates, etc.)

Other programming languages actively referred to in the thesis are C [C11; KRE88], D [Dla; Ale10], Java [GJS+14], $\mathcal{G}$ [Sie05], Scala [Ode11], Haskell [Mar10], ML [MTHM97], F# [SHL+12], and Prolog [DM88]. The C and D languages are efficient programming languages. Java is an object-oriented programming language, widely used in practice, which does not necessarily emphasize efficiency; the main idea in Java is to model everything as a set of stateful objects. $\mathcal{G}$ is an academic language created by Jeremy Siek to demonstrate the benefits of concepts for a language with an emphasis on generic programming. Scala, Haskell, ML, and F# are exponents of functional programming languages. Sparrow operators are mainly inspired from Scala (see chapter 6). From the list of functional programming languages, we also refer to some of the derivatives while evaluating Sparrow features for metaprogramming and DSEL abilities: MetaML [ST97], Template Haskell [JS02], MetaOCaml [CTHL03], and Racket [Rac; Fla12]. Lastly, the Prolog programming language is representative for the logic programming paradigm; we use it as an example of a language, different enough from Sparrow, that we want to integrate as a DSEL into Sparrow (see chapter 11).

Programming languages can be categorized according to the programming paradigms that they employ [ALSU06; GJLB00; Seb12; BW09]. The major recognized programming paradigms include: object-oriented programming, logical programming, and functional programming. The former is part of imperative programming, whereas the latter two are part of declarative programming.

In declarative programming the user specifies what the machine needs to compute, but not how to compute it. In contrast, in imperative programming, the user needs to specify the actions that the machine needs to perform in order to arrive at the desired result.

Object-oriented programming (OOP) is based on the concept of "objects", which are data structures with associated functionality (methods), grouped together in single entities. The objects interact with one another to create complex behavior. Object-oriented programming is widely used in the industry, and it is believed that using OOP is the best way of mitigating the complexities of software projects. In Stroustrup's words: "too many people think that object-oriented is simply a synonym for good.". Examples of programming languages that adopt this paradigm as the main programming paradigm are: C++, Java, C#, D, etc. For wide industry adoption, we used it as the main paradigm for our programming language.

The logic programming paradigm is based on formal logic [DM88]; it relies on the use of *facts* and *rules* to describe the problem to be modeled. The main advantage of modeling problems in this manner is that the user is able to request specific inferred facts by doing *queries*. Although some classes of problems can be solved using the logical paradigm very elegantly, this paradigm is not widely used in practice.

Functional programming emphasizes functions as the main method of abstracting computations. Ideally, the functions are pure, meaning that they produce the same results for the same input arguments, without having any side effects. Languages likes Haskell, ML, and F# are examples of languages that adopt the functional programming paradigm. Even though functional programming is not as widely used as object-oriented programming, it has recently seen a growth in popularity.

From a formal point of view, any program that can be expressed within the OOP paradigm can also be expressed in the functional paradigm, but using slightly different abstractions. The main principles from object-oriented programming (encapsulation, information hiding, decoupling, polymorphism, dynamic dispatch) have correspondents in the functional world, and ultimately they can all be encoded by using functions.

2.2 Compilers and tools

Although not directly visible in this thesis, one of the major activities involved in the construction of the Sparrow programming language was building the compiler and the tools for the language. Some fundamental books describing compiler technologies are: [ALSU06; GJLB00; Mak09; CT11; Par10; App97; Ses12; Wir05]. An insightful overview of compiler technology with predictions of what the future might bring in the compiler research community can be found in [HPP09].

Building the compiler typically involves a variety of activities [ALSU06]: lexical analysis, syntactical analysis, semantical analysis, intermediate code generation, machine-independent optimization, code generation, and machine-dependent optimization. For most of these activities there are specialized tools that provide state-of-the-art implementations.

LLVM [LLVM; Lat02; Lat11] is such a compiler framework, providing a multitude of components that are helpful in building real-world compilers. It contains modern optimizers, code generators for a large number of platforms, frontends for C, C^{++}, and Objective-C, an interpreter and Just In Time compiler, a debugger, an many others. The Sparrow compiler uses LLVM as a backend. The full optimization and code generation chain is implemented with the help of LLVM. We also use it as an interpreter (more precisely, JIT execution engine) to execute computations at compile-time, powering the hypermetaprogramming feature of Sparrow.

Other alternatives to LLVM are GCC [Gcc] and Phoenix [Pho]. GCC is an open-source full compiler stack for languages such as C, C^{++}, Fortran, Java, and many others. Phoenix is a compiler framework focused on the backend side of the compilers: program analysis, code generation and optimization.

Most of the compiler books cover the lexical and syntactical analysis of the compilers. We recommend [HMU07; ALSU06] as excellent textbooks on parsing techniques. There are two main categories of parsers: top-down (LL) and bottom-up (LR). For some parsing frameworks the lexical analysis is included in the syntactical analysis (e.g., every character is a token). See [HMU07; ALSU06] for a more ample discussion on parsing techniques.

For our compiler we use the Flex tool [Fle] for lexical analysis and Bison [Bis; DS93] for syntactical analysis (bottom-up, LALR parser). These two tools are known to work well together [Lev09]. Bison is a derivative of Yacc [Joh75], and Flex is a derivative of Lex [LS75].

A popular alternative to the Bison and Flex tools is the ANTLR framework [Par; Par10]. This is representative of the top-down parsing family, using the LL(*) algorithm [PF11].

2.3 Generic programming

Generic programming is the programming paradigm in which basic algorithms and data structures are *lifted* to a more general form that typically does not depend on the concrete implementation types. Instead, the concrete types are passed as parameters to the these generic algorithms and data structures [MS89; Sie05].Generic programming allows the same code to be used in more than one context, and moreover it allows generating efficient code for each particular use.

Although generic programming was pioneered in ML [Mil78], it was heavily acknowledged with the creation of the STL (Standard Template Library) [SL94; MS89] and its inclusion in the C++ standard. Actually, generic programming was influenced by high-order abstractions typically found in functional programming languages [KMA88]. The C++ experience shows us that generic programming opens the way to new optimizations, while maintaining the same programming primitives.

A good description of generic programming, of different design choices of implementing it, and with a solid bibliographical work can be found in Siek's thesis [Sie05].

There are several books that describe STL and the generic programming model of C++: [Jos12; Aus99; Mey01; PLMS00; Ale01; Str13]. Alexander Stepanov, the creator of STL, takes a more mathematical approach in his books [SR14; SM09]. Generic programming in C++ are implemented on top of the templates feature [VJ02; Str94].

Early work with the C++ template system proved that it can be used in high-performance libraries: [Vel95b; Vel95a; Vel98; VG98; Vel99; LM97; SL98; BDQ98; FGK+98; AJR+01; LL02; GJK+05]. In most of these case studies, C++ template system employs the partial specialization technique [BJE88; JGS93; Jon96] coupled with an ad-hoc polymorphism [Str67; CW85; Sie05] to generate specialized code that is very efficient for the actual usage. Because of efficiency concerns Sparrow uses the same type of polymorphism while implementing generic programming.

Other important work on generic programming in the C++ community include [RJ05; GJL+03; GJL+07].

After the introduction of the 1998 standard, the C++ research community switched its attention to implementing concepts. In the C++ world concepts are predicate on types [Jon92]; they represent a method of grouping together a set of types with similar characteristics, and specialize the algorithms for different groups of types. Aiming at introducing the concepts into the upcoming version of the C++ standard an intense effort was carried out between 2000 and 2009: [SL00; JWL03; SL05; DS06; GSJR06; JGW+06; Got06; JMS07; TJ07; GWJ08; TJ10]. In 2009 the concepts proposal was voted out of the C++ standard; Jeremy Siek gives a detailed report on this *effort* for bringing concepts into C++, and why the decision was made to rule concepts out of the standard:

[Sie12].

A second period of attempting introducing concepts into C++ started with Andrew Stutton, under Stroustrup's guidance, who proposed a *lighter* approach to concepts: [SS12a; SS12b; SSR13].

The concepts effort is important for the C++ community. First, it adds parametric polymorphism to C++ [JWL03; CW85]. Second, it allows algorithms to be specialized on concepts, making the algorithm efficient for a variety of distinct classes of problems, without compromising the generality of the algorithm or changing the interface of the algorithm.

In C++, for each usage of a generic, the compiler will a separate *instance* of the generic, compiled separately [VJ02]; the compiler takes advantage of this while optimizing, and it tends to generate more efficient code than for a non-specialized algorithm. In languages likes Java, C# or ML, the compiler produces a single set of instructions for a generic, sacrificing efficiency in favor of compilation speed.

In the world of functional programming, parametric polymorphism [Mil78; Car96] is the method of choice for implementing generic programming. On top of that, Haskell provides *type classes* to allow a form of ad-hoc polymorphism in the language [WB89; HHPW96]. Haskell's type classes are somehow similar to C++'s concepts [Sie05]. In ML, the equivalent of concepts would be *signatures* [MTHM97]. Comparative studies between programming languages with respect to generic programming can be found in [GJL+03; GJL+07; Sie05].

Recently, in the Haskell community, work has started towards datatype-generic programming; this is a form of abstraction that allows defining functions that can operate on a large set of datatypes. Some of the most important papers in this area are: [LP03; LPJ05; Gib03; GWd07; GP09; HL07; JLPR08; MaDJL10].

2.4 Metaprogramming

Metaprogramming is the ability of programs to manipulate programs [She01]. The program that does the manipulation is called the *metaprogram*, and the manipulated program is called the *target-program* or *object-program*. The language in which the metaprogram is expressed is called the *metalanguage*, whereas the language of the target-program is called the *target-language* or *object-language*. If the two languages are the same we have a homogeneous metaprogramming system; in other words, we say that the language supports metaprogramming. If the languages are not the same, we have a heterogeneous metaprogramming system.

Depending on the moment of the program transformation we have two types of metaprogramming: if the transformation is performed during compilation we have *compile-time metaprogramming*; if it is performed during program execution we have *run-time metaprogramming*. Generally, static metaprogramming is more appropriate for static languages, whereas dynamic metaprogramming suits dynamic languages better. Static metaprogramming tends to be more efficient in the execution of the target program, as the metaprogram does not introduce costs at run-time. Example of programming languages with dynamic metaprogramming support are Lisp and MetaML [ST97]. Examples of programming languages that support compile-time metaprogramming are C++, D, and Racket [Fla12].

Metaprogramming techniques originated in the early days of programming. One can argue that simply using a compiler is an act of metaprogramming: the metaprogram is the compiler, the target-program is the program to be compiled, and the object-program is the binary resulting out of the compilation process. The Sparrow programming language uses homogeneous static metaprogramming, and therefore we focus on static metaprogramming in this thesis.

In C++, the template system [VJ02; AG04] is the primary instrument for metaprogramming; templates can be used to perform computations at compile-time [Vel03; AG04; Por10; Dub13], to help in generating code [Vel95b; VG98; CE00; Ale01; AG04], and to specialize algorithms for better efficiency [Vel95a]. For insight into how C++ templates can be used for high-performance libraries, the reader should consult [Vel95b; Vel95a; Vel98; VG98; Vel99; LM97; SL98; BDQ98; FGK+98; AJR+01; LL02; GJK+05].

Unruh discovered by accident that one can use the template system to perform computations at compile-time; he wrote a program to compute prime numbers and print the results as error messages [Unr94]. Later, Veldhuizen showed that C++ templates are Turing complete [Vel03]—in theory they can execute any algorithm. However, in practice there are limitations to the computations that the template system can perform [Dub13; AG04; VJ02]. Moreover, judging by the method in which computations are encoded in the program, one can say that this kind of metaprogramming in C++ is cumbersome. The syntax is verbose and inappropriate, and furthermore, although C++ is mostly an imperative language, compile-time metaprogramming is performed in a functional style [Por10; AG04]. The reason behind it is that templates were not intended to encode computations—they were introduced in C++ for representing generic library components [Str07].

The 2011 version of the C++ standard [Cpp11] introduced the *generalized constant expressions* feature (`constexpr`) to improve the way in which compile-time computations can be performed [DS10]. One can have composite objects as compile-time constants and call functions using these objects. The D programming language also features a mechanism that allows the programmer to perform more complex computations at compile-time [Dla; Ale10]. However, both approaches have restrictions in terms of syntax and semantics on what the compile-time code can do; for example, they are not allowed to allocate dynamic memory at compile-time. This thesis makes a point of presenting a compile-time metaprogramming system in an imperative language that allows performing arbitrarily complex computations.

Template Haskell [JS02], MetaML [ST97], MetaOCaml [CTHL03] are several functional programming languages[1] that employ staged compilation and have good support for static metaprogramming. Although these languages can perform computations at compile-time, they focus on manipulating and transforming expressions and other code constructs. The Racket programming language is a functional language derived from Lisp that supports compile-time metaprogramming [Rac; Fla12]. Although Racket supports dynamic metaprogramming, through proper use of `require`, the user can enforce execution of certain algorithms at compile-time. With this feature Racket allows writing metaprograms in the same way as one would write them for run-time execution—very similar to our hyper-metaprogramming system.

Domain-specific languages (DSLs) have been used in various forms since the early days of programming languages. Approaches range from APIs for a particular domain, to overloading operators to support domain-specific abstractions, and parsing a language with a different syntax. In some cases, external tools provide the support for adding DSLs (e.g., Lex and Yacc practically bring new DSLs to aid in the creation of compilers). Vasudevan et al. [VT11] and Czarnecki et al. [COST04] provide two brief comparisons between languages/tools with support for DSLs.

Ignoring external tools, there are two main categories of host languages with respect to Domain-specific embedded languages (DSELs): one in which the DSEL must follow the same lexical and syntactical rules of the host language, and one in which the DSEL can have arbitrary syntax. C^{++} [CE00; COST04; Vel95a] and Scala [RO12; SGB13] are two examples of languages in the first category. Later in the thesis we will showcase how one can implement in Sparrow DSELs that can have arbitrary syntax.

Languages can also be categorized according to the moment when the parsing of the DSEL takes place: run-time or compile-time. Some notable systems that interpret DSELs at run-time are MetaML [ST97], MetaOCaml [CTHL03], Mint [WRI+10], and Scala [RO12]. Languages like Template Haskell [JS02], Racket [Fla12], and Converge [Tra08] are prominent examples of languages that support DSELs with arbitrary syntax, implemented at compile-time. Template Haskell and Racket are functional languages that let the user specify methods of interpreting the content of the embedded code, similar to our approach. In Converge, the parsing of the embedded code is performed using a provided parser toolkit, and the code is generated with the help of quasiquotation.

Our work on embedding new languages in Sparrow is similar in many ways to the approaches used in Template Haskell [JS02] and Racket [Fla12]. How-

[1]derived from Haskell, ML, and OCaml, respectively

ever, these are functional languages, whereas we are targeting an imperative language.

Throughout our thesis we show how Sparrow employs partial evaluation [BJE88; JGS93; Jon96; Fut99; Glu09] to generate efficient code. The work on C++ templates (e.g., [Vel95a; Vel99]) shows how the ad-hoc polymorphism of C++ templates is able to generate efficient code; Sparrow generics are designed to take advantage of this. On the other hand, this thesis will show (see chapter 9) how partial evaluation can be used to move computations from run-time to compile-time, thus increasing the efficiency of the generated program. Some recent examples in which partial evaluation can improve the efficiency of the generated code can be found in: [RO12; RSA+13; SC11; SGB13]

PART I

The Sparrow programming language

I have so much I want to tell you,
and nowhere to begin.

J.D. Salinger

CHAPTER 3 BRIEF INTRODUCTION TO SPARROW

This chapter provides a short introduction into Sparrow programming language. We expose the basic features that allow users to write traditional programs. The aim is to introduce the basic concepts that users will encounter while programming in Sparrow, without diving into more complex features. This way, throughout this paper the reader will easily recognize most Sparrow language constructs.

3.1 Hello, world!

In following computer science tradition, the first example program that is given in a language is the *Hello, world!* program. In Sparrow, a program that displays the text `Hello, world` to the console can be written as:

```
fun sprMain {
    cout << "Hello, world!" << endl;
}
```

The `sprMain` function will be called when the program starts. Optionally, it can receive a range of strings representing the command line arguments[1].

Similarly to a C++ program, to display something to the console Sparrow uses the insertion operator to put the string into the `cout` object. This object represents the standard console output. Adding an `endl` expression at the end causes the program to display a new-line character.

[1]more on ranges in section 3.9

3.2 Values and expressions

The `cout` construct can be used to display to the console various values, as shown below:

```
cout << 1 << endl;
cout << -1 << endl;
cout << 3.141592 << endl;
cout << "square root of 2 is " << 1.41421356237 << endl;
```

The values can also be used in arithmetic expressions:

```
cout << (2+2) << endl;
cout << (12-4) << endl;
cout << (2*3.141592 / 180.0) << endl;
cout << "square root of 2 is " << Math.sqrt(2) << endl;
```

Here the parentheses are optional, but are added for better readability.

The standard arithmetic operations (`+`, `-`, `*` and `/`) can be used for numbers (integers, floating point). The standard operators can be overloaded, and moreover, non-standard operators can be defined by the user. More information on operators is available in chapter 6.

The `Math.sqrt(2)` is a call to a function defined in the Sparrow standard library that computes the square root of its argument. In Sparrow, functions can be called by following the conventions of most programming languages: *funName(args)*.

A line such as `cout << (2+2) << endl;` is an expression statement. `2+2` is a subexpression, `cout` and `endl` are operands, and `<<` is an operator. This is an expression that, instead of producing an useful value, will display something to the console.

3.3 Variables

An important concept of imperative programming languages is the variable. A variable is an *alias* to a memory location in which we store values. Here are some examples of defining variables in Sparrow:

```
var n = 10;
var m: Int = 15;
var f: Float = 0.3;
var greet = "Hello, world!";
var k: Long;
var p1, p2, p3: String;
```

When declaring a variable one needs to supply the name of the new variable (or variables), an optional type, and an optional initial value. It is not possible to have a variable definition that lacks both the type and the initial value. If the variable receives an initial value without a type, the type of the value will be taken from the initializer. If the variable does not have an initial value, the variable will be *default constructed* (more precisely, the default constructor will be called for the variable); this assures that the variable is initialized.

After definition, the variables can be used in any place in which a value can be used:

```
cout << "Our greeting: " << greet << endl;
cout << m-n << endl;
cout << f*n << endl;
```

Moreover, the value of a variable can be changed at any time by calling the assignment operator (=):

```
m = 20;
k = m-n;
p1 = "Our greeting: ";
p2 = "Hello";
p3 = ", world!";
```

Like C++, Sparrow provides a series of operators on integer values. For example:

```
k = ++m;     // m will become 21, and k will become 21 too
k = m++;     // m will become 22, and k will get the old value: 21
k -= n;      // subtract n from k
n /= 2;      // divide n by two
```

3.4 Basic types

In Sparrow, any value, variable, or expression needs to have a well defined type. A type determines the way a value can be encoded in the system's memory and what operations are valid for that variable.

The standard library defines a series of integer types: `Byte`, `UByte`, `Short`, `UShort`, `Int`, `UInt`, `Long` and `ULong` of sizes 8, 16, 32 and 64 bits, signed and unsigned. Two additional integer types are defined to contain at least as many bits as a pointer: `SizeType` and `DiffType`; the first one is unsigned and the second one is a signed type. To represent floating point numbers, the language defines the types `Float` (32 bits) and `Double` (64 bits).

To represent booleans the language defines the `Bool` type. To represent characters we have the `Char` type. In Sparrow, strings use UTF-8 encoding, so setting the `Char` to 8 bits is an obvious choice.

In the most basic form, strings can be represented as a `StringRef` type. This just refers to the string, but does not hold ownership of the string data. It is analogous to the `const char*` type from C/C++, but more type-safe. To use a string with ownership of data, one can use the `String` type. String literals have the type `StringRef`, but there is an implicit conversion between a `StringRef` and `String`.

Certain implicit conversions (called *type coercions*) can be made between these types. An integer type can always be converted into an integer type of a larger size. An unsigned type can be converted into a signed type of the same size, and vice-versa. Any integer type can be implicitly converted into a floating point type.

Here are some implicit conversion examples:

```
var b: Byte = 1;
var ub: UByte = 2;
var i: Int = 3;
var ui: UInt = 4;
var l: Long = 5;
var f: Float = 3.14f;
var d: Double = 3.14159265359;

i = b;       // OK: Byte -> Int
i = ub;      // OK: UByte -> Int
ui = i;      // OK: Int -> UInt
i = ui;      // OK: UInt -> Int
// b = i;    // ERROR: cannot convert wider to narrower type
f = l;       // OK: Long -> Float
d = b;       // OK: Byte -> Double
```

Note that some of these conversions can loose precision (e.g., large integers to floating points), and sometimes can even dramatically change the actual value (negative number to unsigned or large number to signed). The user must be careful when performing such conversions. As the benefits provided by these conversions are typically more significant than the drawbacks, they are allowed.

3.5 References

Sparrow supports references as a method of referring to a memory location. Although Sparrow references resemble C++ references more closely, one can think of them as being pointers.

A reference can be declared in Sparrow using the `@` operator applied to a type, as shown in the following example:

```
var i: Int = 1;
var ri: @Int = i;        // We need to initialize the reference

cout << ri << endl;      // prints 1
i = 22;                  // also changes ri
cout << ri << endl;      // prints 22
```

```
ri = 33;                 // also changes i
cout << i << endl;       // prints 33
```

As can be seen in this example, having a reference (`ri`) to a particular value associated with a regular variable (`i`), any change to the original variable will be reflected in the reference variable, and vice-versa. Otherwise, the reference variable can be used just like a regular variable.

3.6 Control structures

Like most imperative programming languages, Sparrow supports control structures like **`if`**, **`while`**, and **`for`**. The structure for **`if`** statements is identical to the one in C++:

```
if ( n % 2 == 0 )
    cout << n << " is even" << endl;
else
    cout << n << " is odd" << endl;
```

The **`while`** statement is a combination of C++'s **`while`** and **`for`** statements. In the most basic form, the **`while`** statement executes a command (or a list of commands) as long as a condition is true. For example, the following code computes the length of the Collatz sequence[2] that starts with a given number `n`:

```
var len = 1;
while ( n > 1 ) {
    ++len;
    if ( n % 2 == 0 )
        n /= 2;
    else
        n = n*3 + 1;
}
```

In Sparrow, we allow adding a *step* action to a **`while`** statement. For example, summing the squares of all natural numbers up to a certain value can be written like this:

```
var res = 0;
while ( n > 0 ; --n )
    res += n*n;
```

[2]a Collatz sequence is a sequence of numbers that, starting with a given number we continuously apply a special transformation to the given number to produce more values, until we reach value 1. The used transformation is the following: if the number is even, divide it by two; otherwise multiply it by three and add 1. For example, the Collatz sequence that starts with the number 13 is: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. This sequence has the length 10.

Note that this form is somewhat similar to the **for** instruction from C++. The main difference is that the initialization statement needs to be placed before the **while** statement, and not inside it.

The **for** structure from Sparrow is similar to range-based for loops from C++ and other languages. Instead of providing an initialization statement, a condition expression, and a step statement, the user needs to provide a range that can produce values. Here is one example:

```
var numbers: Int Vector = getNumbers();
for ( val = numbers.all )
    cout << val << endl;
```

The `numbers` object is a vector of integers. Like all the standard containers it exposes a method named `all`, which returns a range that we can use to iterate over the values in the vector. In our example, `val` is a variable introduced with the **for** structure, and will have the type `Int`.

To iterate over a set of numbers (e.g., from 1 to `n`), one can use the following code:

```
for ( x = 1..n )
    cout << x << endl;
```

Here, `..` is an operator that generates a range that will yield values between 1 and `n` (open range). To indicate a closed range, one needs to use three dots (`...`) instead of two. One can also iterate with a given step:

```
for ( x = 1...n ../ 2 ) // odd numbers in range [1, n]
    cout << x << endl;
```

Here, `..`, `...`, and `../` are all infix operators that act on numbers. We will see that ranges represent an important concept in Sparrow, and there are many other range constructors.

Sparrow does not support the `goto` statement.

3.7 Function definitions

Functions are the main method of representing computations. The following example presents a function definition in Sparrow that can compute the `n`'th Fibonacci number:

```
fun fib(n: Int): Int {
    if ( n <= 1 )
        return 1;
    else
        return fib(n-1) + fib(n-2);  // recursive; not optimal
}
```

If the function does not return a value, it can return the `Void` type, or it can omit the return type completely:

```
fun greet1(name:String):Void {cout << "Hello, " << name << endl;}
fun greet2(name:String)      {cout << "Hello, " << name << endl;}
```

If the function does not have any parameters, the parameter list can be omitted:

```
fun geetTheWorld             {cout << "Hello, world!" << endl;}
```

Sometimes, when a function is simple enough the user can define the function with an alternative syntax that puts emphasis on the returned value rather than the actual instructions involved. Here is one example:

```
fun sum(x, y: Int) = x+y;
```

This has exactly the same semantics as writing the function in the classic way:

```
fun sum(x, y: Int): Int { return x+y; }
```

The functions defined so far can be used as follows:

```
greet1("Alice");    // prints "Hello, Alice"
greet2("Bob");      // prints "Hello, Bob"
greetTheWorld();    // prints "Hello, world!"
greetTheWorld;      // parenthesis can be omitted here
cout << sum(2, 4) << endl;  // prints 6
```

The reader should note that if a function does not take any parameters the parentheses can be omitted when calling the function. This provides an interesting property of the language in that we can use function and variable names interchangeably, without changing the code that uses the variable/function.

So far, we have shown function definitions that operate on concrete data types. In addition to those, the Sparrow programming language allows definitions of *generic* functions that have parameters of *concept* types. For example, the previously defined `sum` function can work on all numeric types, not just on values of type `Int`. The following function definition is able to work on all numeric types:

```
fun sum(x, y: Numeric) = x+y;
```

The `Numeric` name refers to a *concept* defined in the standard library that accepts any numeric type (e.g., `Int`, `ULong`, `Double`). More on generics and concepts can be found in chapter 7.

There is a special concept in Sparrow called `AnyType` that is compatible with any type. Here is an example of a function that prints to the console the value given as parameter:

```
fun writeLn(x: AnyType)     { cout << x << endl; }

writeLn(10);                        // prints an Int value
writeLn(3.14);                      // prints a Double value
writeLn("Pretty cool, huh?");       // prints a StringRef value
```

In cases where all parameters are `AnyType`, the parentheses and the type specifications can be omitted:

```
fun writeLn x { cout << x << endl; }
fun sum x, y = x+y;
```

As can be seen from the definition of the `sum` function, this form can be very compact.

A function is not just a definition that can be invoked directly; we can store a function in an variable. This allows us to separate the binding of the function name from the actual function call. Here is one example, in which we pass a function as a parameter to another function:

```
 1  fun applyFun(n: Int, f: AnyType) {
 2      for ( x = 0..n )
 3          cout << f(x) << ' ' << endl;
 4  }
 5  fun mul2(x: Int) = 2*x;
 6  fun sqr(x: Int) = x*x;
 7
 8  var f = \mul2;       // type: FunctionPtr(Int, Int)
 9  applyFun(10, f);     // 0 2 4 6 8 10 12 14 16 18
10  f = \sqr;
11  applyFun(10, f);     // 0 1 4 9 16 25 36 49 64 81
```

At line 8 we create a variable and initialize it with a reference to the `mul2` function. To take the reference of a function, Sparrow uses the backslash operator. The type of this function will be `FunctionPtr(Int, Int)` (a function that returns an `Int` and takes one `Int` as parameter). This variable can then be passed as the second argument to the `applyFun` function. Note that instead of using `AnyType` for the second parameter we could have used `FunctionPtr(Int, Int)`; still, we find that retrieving an `AnyType` is not only simpler, but also more generic.

We could have passed the function references directly to the function call, but we wanted to show that we can store function references in variables, just like any other values.

There is an easier method of achieving the same result, without defining the `mul2` and `pow` function prior to the call to `applyFun`. Instead, we could have used lambda functions (or anonymous functions):

```
applyFun(10, (fun x = 2*x));    // 0 2 4 6 8 10 12 14 16 18
applyFun(10, (fun x = x*x));    // 0 1 4 9 16 25 36 49 64 81
```

The form `(fun ...)` does the following: creates a function-like structure that can be called with the written computation, and then instantiates this functor to produce an object that can be called under the specified conditions. Note that this object is of an unspecified type, and cannot be placed inside a `FunctionPtr(Int, Int)`.

This expression can become a *closure* if it refers to variables declared in the scope in which it is used. In Sparrow, one needs to explicitly declare all the variables that are used by the closure. For example, if we generate the first lambda function to parameterize the factor we are multiplying with, we can write:

```
var k = 2;
applyFun(10, (fun.{k} x = k*x));     // 0 2 4 6 8 10 12 14 16 18
k = 3;
applyFun(10, (fun.{k} x = k*x));     // 0 3 6 9 12 15 18 21 24 27
```

3.8 Operators

Like in most programming languages, there are three types of operators in Sparrow, depending on the placement of the operator relative to its argument(s): prefix, infix and postfix. Prefix and postfix operators are unary, whereas infix operators are binary. An operator can be either a set of symbols or a regular function name. Moreover, Sparrow does not limit the names of operators formed by symbols to a fixed set (like C++ for example).

Here are some basic examples of operators in Sparrow:

```
-10                // '-' is a prefix operator
--k;               // prefix operator
k++;               // postfix operator
a + b * c;         // '+' and '*' are infix operators
a + -b * c;        // '+' and '*' are infix operators, '-' is prefix
```

Defining an operator is very similar to defining a function. Here is an example of defining an operator to raise a number to an integer power:

```
fun **(x: Double, p: Int): Double {
    var res = 1.0;
    for ( i = 0..p )
        res *= x;
    return res;
}

cout << (3 ** 2) << endl;        // 9.0
cout << (3.2 ** 2) << endl;      // 10.24
```

Defining unary operators is as easy as defining infix operators; we just need to specify whether we need prefix or postfix operators:

```
fun pre_**(x: @Int): @Int   { x = x*x; return x; }
fun post_**(x: @Int): Int   { var old = x; x = x*x; return old; }

var a, b = 5;
cout << **a << endl;     // writes 25, a becomes 25
cout << (b**) << endl;   // writes 5, b becomes 25
```

If the `pre_` and `post_` prefixes are missing, then the operators can be used as both prefix and postfix operators.

Note that for prefix operators the parentheses are not needed, whereas for the postfix operators they are. In both cases, the first occurrence of << needs to be an infix operator; after it we can have a *primary expression* or a prefixed primary expression. In the first case, it is clear to the compiler that we have a prefix operator (because `**` cannot be an operand), while in the second case the compiler will treat `b` as the second operand of <<, and `**` as the next infix operator.

In general, in an expression that is separated by spaces, the terms on the even positions are infix operators and the terms on the odd positions are operands. This rule changes if an operand has one or more prefix operators applied to it, in which case we collapse the prefix operators first. If the expression has an even number of terms, the last term is a postfix call. Here is an example:

```
a + - - b * c !!!
```

In this expression, `- - b` will be treated as one operand, + and `*` as infix operators, while `!!!` will be treated as a postfix operator.

Beside operators that are formed by symbols, Sparrow allows operators to have alphanumeric names. For example, the `pow` and `sqr` functions previously defined can be used in the following way:

```
2 pow 3;            // 8
2 sqr;              // 4
`sqr` 3;            // 9
2 pow 3 sqr;        // 64 = (2^3)^2
```

Note that, in order to distinguish prefix name operators from name operands, Sparrow requires the placement of these prefix operator names in backquotes.

This is an important feature of Sparrow that allows writing concise programs, without losing performance. More on operators can be found in chapter 6.

3.9 Ranges

In Sparrow, a range is a collection (not necessarily finite) of elements that can be iterated through in an well-defined order. From an implementation point of view, ranges need to support three operations: *is the range empty?*, *get the current value*, and *move to the next value*. With these three operations one can extract all the values from the range.

We have already seen that `1..n` is a range. This is a range which will produce `Int` values starting from `1` and ending with `n` (without actually yielding `n`).

All the standard containers provide methods for accessing their values through ranges. In addition, Sparrow provides methods of generating ranges. Here are some examples:

```
repeat(13);              // infinite range with value 13
repeat(13, 5);           // 13 repeated 5 times
generate( (fun = 13) ); // infinite range with value 13
generate(\getNextRand); // infinite range with values of getNextRand
generate1(2, (fun x = x*x));    // 2, 4, 16, 256, ...
```

In addition to those, there are many functions that apply transformations to existing ranges. The most common is the `map` operation. It applies a functor to the given range to produce a new range:

```
1..10 map (fun x=x*x);  // 1, 4, 9, 16, 25, 36, 49, 64, 81
1..10 map (fun x=x/2);  // 0, 1, 1, 2, 2, 3, 3, 4, 4
1..5 map (fun x=repeat(2*x, x)); // range of ranges:
                                 // (2), (4,4), (6,6,6), (8,8,8,8),
                                 // (10,10,10,10,10)
```

Another important range operation is `filter`. It skips elements in the input range if they do not satisfy a predicate:

```
1..10 filter (fun x = x%2==1);  // 1, 3, 5, 7, 9
1..10 filter (fun x = x<5);     // 1, 2, 3, 4
```

Here are some examples of other range functions:

```
(1..) take 5;                   // 1, 2, 3, 4, 5
(1..) takeWhile (fun x=x<=5);   // 1, 2, 3, 4, 5
(1...3) ++ (10...12)            // 1, 2, 3, 10, 11, 12
(1...3 cycle) take 8;           // 1, 2, 3, 1, 2, 3, 1, 2
```

To illustrate the power of ranges we would like to solve the following problem: *the sum of the first 10 Fibonacci numbers that are greater than a given number*. Here is the Sparrow solution using ranges:

```
var res = (1..) map \fib filter (fun.{n} x = x>n) take 10 sum
```

Our solution is simple, and yet efficient. Starting from the range of natural numbers, we map them to Fibonacci numbers, we take only the values that are greater than the given `n`, we get the first 10 such elements, and finally we sum them.

3.10 Classes

Sparrow supports the Object-Oriented-Programming (OOP) programming paradigm. As a fundamental concept, a class is a programming language construct that defines a structure for program data and associates behavior (methods) to that data; it groups them together in the same construct. It defines the memory layout of the data, it specifies the constraints on the data, and allows the definition of basic manipulation methods. In Sparrow, defining classes is the only way of creating data types. Even the standard types are defined as classes.

In Sparrow, a class is defined in the following manner:

```
class MyItem {
    var id: Int;
    var name: String;
    var description: String;
    private var isLent: Bool;
    private var borrower: String;

    fun ctor(id: Int, name, description: String) {
        this.id ctor id;
        this.name ctor name;
        this.description ctor description;
        this.isLent ctor false;
    }
    fun borrow(toWhom: String) {
        borrower = toWhom;
        isLent = true;
    }
    fun restore {
        borrower = "";
        isLent = false;
    }
    fun isAvailable = !isLent;
    fun borrowerName = borrower;
}
```

To use a class, one needs to create an instance of the class (also called an *object*), which can be done by simply declaring a variable of the class type. The following code shows how our `MyItem` class can be used:

```
var item = MyItem(1, "pen", "a nice, blue color pen");
item.borrow("Alice");
if ( !item.isAvaiable )
    cout << " Item" << item.name
         << " is lent to " << item.borrowerName << endl;
```

Accessing data and calling methods from the class is done using the `.` syntax. If a member function does not take any arguments, it can be used just like a variable.

There are two special types of methods that can be placed in a class: constructors and destructors. Constructors are called `ctor`, whereas destructors are identified by the name `dtor`. A constructor is responsible for bringing an object into a valid, consistent state. The destructor is responsible for releasing the resources of an object (mainly memory) before the object is destroyed.

In our example we defined a constructor that creates an object with the given id, name, and description. A default constructor is one that takes no parameters, and initializes the object with some default state. By default, if no default constructor is provided, the language will generate one. The same applies for a copy constructor (a constructor that can create an object by copying the data from another object of the same type).

A destructor does not have any parameters. Like in the case of constructors, if the user doesn't supply a destructor, the language will automatically create one, by calling the destructors for all the data members.

Sparrow follows the C++ tradition and expects manual memory management[3]; it does not provide garbage collection. In such a language, constructors and destructors pay a very important role.

Just like functions, classes can be generics as well. While a regular class does not take any parameters, any class that has parameters is a generic. Here is an example of a class generic:

```
class Pair(t1, t2: Type) {
    var first: t1;
    var second: t2;

    fun ctor(f: t1, s: t2) {
        first ctor f;
        second ctor s;
    }
}
```

In this example we defined a class parameterized by two types[4]. To be able to use this generic, one needs to *instantiate* it; this is the process that transforms a class generic into a class. Class instantiation is just like function application:

```
var p1: Pair(Int, Double);            // call default constructor
var p2 = Pair(Int, Double)(1, 3.14);  // call our constructor
p1.first = 10;
p1.second = 2.34;
cout << "(" << p2.first << ", " << p2.second << ")" << endl;
```

[3]if we count smart pointers, one can say that the user needs to do semi-automatic memory management

[4]as a class is a compile-time entity, any generic parameters need to be available at compile-time; for example, a class cannot have an `Int` parameters, but can have an `Int ct` parameter

3.11 Standard library

Sparrow provides a minimal standard library that provides the user with the basic abstractions for writing programs. The Sparrow standard library was influenced to some degree by the C++ standard library.

One of the most important abstractions in a standard library are the containers. Sparrow provides the following general-purpose containers:

- `Vector` - a dynamic size array, holds the elements contiguously in memory
- `List` - a generic double-linked list that allows constant time insertion and removal in any place of the container
- `Set` - associative containers that ensures that objects are uniquely stored in the set; the search, insertion, and removal operations have average constant-time complexity; implemented using hash tables
- `Map` - provides a mapping from a set of keys to a set of values; the search, insertion, and removal operations have average constant-time complexity; implemented using hash tables

The containers are implemented as generic classes. Any container has a method called `all` that returns a range of the elements in the container. They all provide methods for accessing elements, inserting, and removing elements from the container. Example:

```
var v: Vector(Int) = 0..100;   // vector of integers
for ( x = v.all )              // iterate over all elements
    cout << x << endl;
v(0) = 12;                     // change the first element
v.pushBack(42);                // append at the end of the vector
v.subrange(5, 10) sort;        // sort 10 el. starting at index 5
v.insertBefore(0..10, v.all); // insert 10 numbers at start
v.remove(v.subrange(3, 12));   // remove 12 elements
```

Following STL principles [Str13; Jos12] the algorithms that operate on data are separated from the containers holding the data. Unlike C++ which uses iterators as a bridge between containers and algorithms, Sparrow uses ranges. Here is a short example:

```
var v: Int Vector = 0..100;     // use postfix operator notation
var l: Int List = 0..100;
replace(v.all, 10, 110);
replace(l.all, 10, 110);
(v.all find 50) size;          // range starting with 50 -> size
(l.all find 50) size;
v.all map \fib sum;
l.all filter (fun x = x%10<5) maxElement;
v.all copy l.all;      // copy list elements into vector elements
```

Beside containers, ranges, and algorithms, the Sparrow standard library also provides utilities, such as: strings, pointers (raw, scoped, shared), memory allocation, pairs, optionals, bitsets, math functions, etc.

3.12 Summary

The Sparrow programming language is a simple language, but at the same time it provides the most common primitive to build complex abstractions. It provides most of the important features from languages like C++. However, the syntax and the semantics of the Sparrow language make it simpler than the C++.

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature.

C.A.R. Hoare

CHAPTER 4

DESIGN

Growing a language... a hard thing to do [Ste99]. When designing a general purpose programming language one usually aims for a certain degree of complexity. Simpler languages (e.g., Lisp) provide a limited set of primitives, but are very flexible. More complex languages (e.g, Python, Perl) provide a multitude of high-level features that target various application domains, providing the tools required for the users to build complex applications. A small language that can be easily extended is usually preferable, but only if the language can provide the users with all the features required for building real-world applications. From the language designer's point of view, the main problem with a small language is creating a mechanism for extending the language in order to support high-level features.

This chapter describes the design of the Sparrow programming language, a language that strives to be as simple as possible, while still providing high-level abstractions as library features. We first describe the language's main principles—efficiency, flexibility, and naturalness—as they dictate the general direction for Sparrow's development. We discuss how these principles are apparently contradictory, and how we can employ metaprogramming to integrate them into the same language. We then move to describing hyper-metaprogramming, the feature that allows the user to execute any complex algorithm at compile-time, thus extending the compiler. As we will see throughout the thesis, this is the most important feature of Sparrow, as it allows increasing efficiency, flexibility, and naturalness, at the same time. Finally we describe the reduction mechanism. It dictates how the compiler is organized, and how we can implement more complex features on top of simpler features.

4.1 Language principles

4.1.1 The three main principles

The Sparrow programming language is designed according to three main principles: efficiency, flexibility, and naturalness.

Efficiency

Sparrow aims at producing nearly optimal code for the target machine. It aims for the same efficiency class as C++ and C. This way, Sparrow should allow the programmer to express low-level constructs, and moreover, should allow understanding the transformations that the code (expressed at high-level in Sparrow) undergoes in order to be translated into machine code. The users should be able to control the efficiency of every piece of Sparrow code.

It should be noted that every programming language aims to some degree of efficiency. After all, the resulting programs have to run on real-world machines to solve real-world problems. Even languages that, by design, focus primarily on convenience (or naturalness), have efficiency considerations. For example, in functional languages users are encouraged to apply tail-recursion when alternative forms would be simpler. As another example, in logic programming languages the *backtracking cut* has the sole purpose of optimizing backtracking calls; a pure logical programming language would not need this functionality.

Looking at the programming language rankings of February 2015 generated by TIOBE [Sof15], we find that the first and third most popular languages are C and C++[1]. As these two languages have efficiency as their main concern, one can infer that efficiency is an important topic in the field of programming languages.

Because of their popularity, but also because they are proven to be languages that generate efficient code [BG], we will often compare the two with Sparrow throughout the thesis. Considering that C++ has a higher abstraction level than C and that a C program is also a valid C++ program most of the time, we will refer to C++ more often than C. Therefore, we aim for the same efficiency level as C++.

Flexibility

Flexibility is one of the most subjective concepts in programming languages. We do not have any particular measurement that can indicate the level of flexibility of a programming language. We define flexibility as the variety of problems that a language can solve (in a relatively *clean* manner) using a fixed

[1]this index changes monthly; however, looking at the long term history, since 1990, the C and C++ languages have always been among the first four positions

set of language primitives. A possible way of encoding this into a formula could be:

$$Flexibility = \frac{ProgramVariety}{LanguageComplexity}$$

When discussing the variety of problems that a language can solve we must include the distinction between general-purpose languages (that can solve a large variety of problems) and domain-specific languages (that are tailored to solve the problems in one particular domain) [CE00]. *ProgramVariety* is higher for general-purpose language. However, a flexible language should have mechanisms for creating domain-specific abstractions and operating with them in an easy manner. A language that allows the user to create Domain Specific Embedded Languages (DSELs) [CE00] should be counted as a language with increased *ProgramVariety*.

Allowing the programmer to encode solutions using different methods and programming paradigms (without using language features in unusual or unintuitive ways) also counts as increased *ProgramVariety*.

On the other hand, language complexity has a negative effect on its flexibility. The more complex the language is, the more *rigid* it is, making it harder for the user to solve problems.

For example, one can say that Scheme is a flexible language [AOP+98]. It provides a small number of primitives[2], and is able to encode solutions to very complex problems with the same syntax and programming style (albeit possibly hard to read). Scheme even allows a very complex form of metaprogramming that few other languages allow, and it does that without any syntactic or semantic changes to the core language.

On the other hand, a language like C++, although it allows writing a large variety of programs in different styles and paradigms, it contains a large number of language primitives. The latest language standard [Cpp14] consists of more than 1300 pages describing a highly-complex set of rules. Scott Meyers' *Type Deduction and Why You Care* presentation at CppCon 2014 [Mey14b] is a good showcase for the language's complexity, with respect to simple features like automatic type deduction[3]. Discussing about C++ templates, Vandevoorde and Josuttis affirm [VJ02]:

> Although templates have been part of C++ for well over a decade [...], they still lead to misunderstanding, misuse, and controversy.

[2]essentially everything is a list, functions are also data, and the language associates semantics to a few special forms (e.g., define, lambda, if, etc.); everything else can be built on top of these

[3]the presentation contains material from his Effective Modern C++ book [Mey14a], presented with an emphasis on the complexity of the C++ rules

In Sparrow we aim to have a relatively small number of language primitives, constructed in a coherent manner; they need to be easily composable to allow generating more complex structures.

One of the best exposition of this principle is provided by Guy L. Steele Jr., in his inspiring talk *Growing a language*: [Ste99]

> I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow—but I need, too, to leave some choices so that other persons can make those choices at a later time.
>
> [...]
>
> This leads me to claim that, from now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.

Naturalness

Naturalness refers here to the *ease* of writing programs in a programming language. We consider intermediate-to-advanced programmers, the ones who write complex and efficient code, as the target audience for our language. For example, we say that it is natural to write imperative code with variable assignments, `while` structures, and `goto` instructions in a language like C, but it is not natural to express these in languages like Prolog or Haskell (cumbersome, although possible).

As naturalness is also a concept without a precise meaning, and difficult to measure, we use the following formula for defining the important properties of naturalness:

$$\begin{aligned}Naturalness = {} & ProgramLength + Syntax + CoherentConcepts \\ & + HighLevelConcepts + AppropriateParadigm\end{aligned}$$

It is common wisdom that the shorter the programs the easier it is for the programmer to understand and reason about the code[4]. As opposed to the other terms, this one can easily be measured.

Syntax is a somewhat subjective matter. Different programmers find different syntax styles to be easier to grasp and use. However, due to the popularity of C-based languages, we consider a syntax closer to C++, Java, or Scala to be more natural than a Scheme-like syntax.

Brooks [BJ95] divides the difficulties of software development into essential difficulties and accidental difficulties. Essential difficulties are inherent to

[4]this is true only to some degree; one can find plenty examples of obfuscated code that is smaller than a traditional solution to the same problem

software development, while the accidental ones are not, as they arise from minor problems regarding the methodology of developing software and the tools used. He argues that complexity[5] is an essential property of software development, therefore one cannot find a silver bullet for it. The only way to mitigate complexity is to conquer it step by step, to find methods that only decrease it by a small amount. To solve the accidental difficulties, Sparrow aims to have a clear and concise syntax, and to also be as coherent as possible from the semantic point of view.

Following on Brooks' work, one may argue that people cannot deal properly with inherent complexity as long as their primary tools are the **`if`** and the **`for`**. From this perspective, people need to think in higher-level terms and must not reinvent the wheel every time. For example, instead of writing the same old sorting algorithm over and over, the programmer should say something like "*I need this container to be sorted at this point*". He/she should not care what kind of container needs to be sorted (if it is a random-access array, a singly or doubly linked list, or a red-black tree), the exact moment of the sorting (maybe at that point the container is already sorted), nor which algorithm is used for sorting. Similarly, the programmer should think in terms of: "*I want a grammar, here are the production rules, the compiler should figure out the details*" or "*this is a database, save all my objects here, let the compiler find the best way of connecting to the database*". Being able to build complex program constructs to match the user's high-level concepts, the language must provide a set of coherent basic primitives that can be combined to create more and more complex structures, without sacrificing conciseness.

For expressing programs as close as possible to the way in which people think, the programming language should also allow the programmer to use different methods and paradigms for solving problems. For example, a program may have a parsing module implemented using a yacc-like grammar specification [Joh75] or an EBNF grammar specification [Wir77], an inference module implemented using logic programming, and the rest of the application implemented using an object-oriented or functional approach.

4.1.2 Integrating these principles

These three language principles appear to be highly desirable if taken in isolation. The problem is that they are often found to be in opposition to each other.

The best example is the opposition between efficiency and naturalness. It is common wisdom that the more efficient a language is, the less natural it tends to be. For example, assembly languages or the C languages are widely considered hard to use. Conversely, trying to increase naturalness typically

[5]Brooks refers here to the term complexity as the property of software systems to be complicated, to be composed from different parts that are not of the same kind.

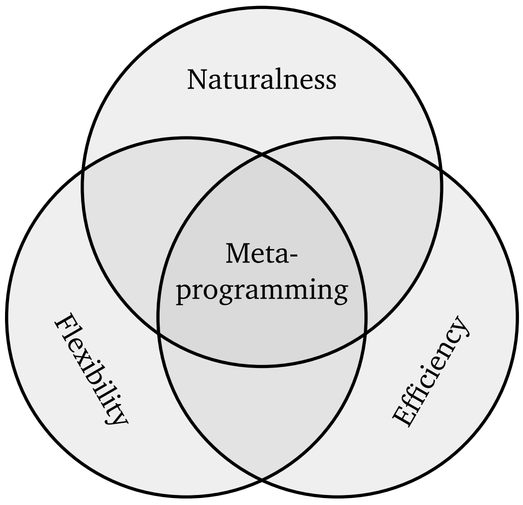


Figure 4.1: Metaprogramming and the three goals of the Sparrow programming language

detracts from efficiency. Efficiency implies working with low-level abstractions, while naturalness implies working with high-level abstractions. In programming languages where efficiency is important, the user often deals with stack variables and manually managed heap allocations; in languages that aim for naturalness, usually everything is on the heap and garbage collection takes care of freeing the memory.

Flexibility can also be in contrast with the other two goals. Typically, by trying to improve flexibility, we can degrade naturalness or we can lose efficiency.

How can we integrate these three language principles? We think the answer is *static metaprogramming*. It can act as a *glue* between the three apparently opposing principles, as depicted in fig. 4.1. Let us briefly describe how metaprogramming[6] can help in increasing the efficiency, flexibility, and naturalness of programming languages, and moreover how can it help to combine these three goals.

Efficiency

- Partial specialization
- Moving computations from run-time to compile-time
- Better organization of resources at compile-time (the compiler detects the usage of the resources, and tries to plan for better access to the resources)

[6]to avoid cluttering the text, we sometimes omit the word *static*; if the word *metaprogramming* is not qualified we intend to say *static metaprogramming*

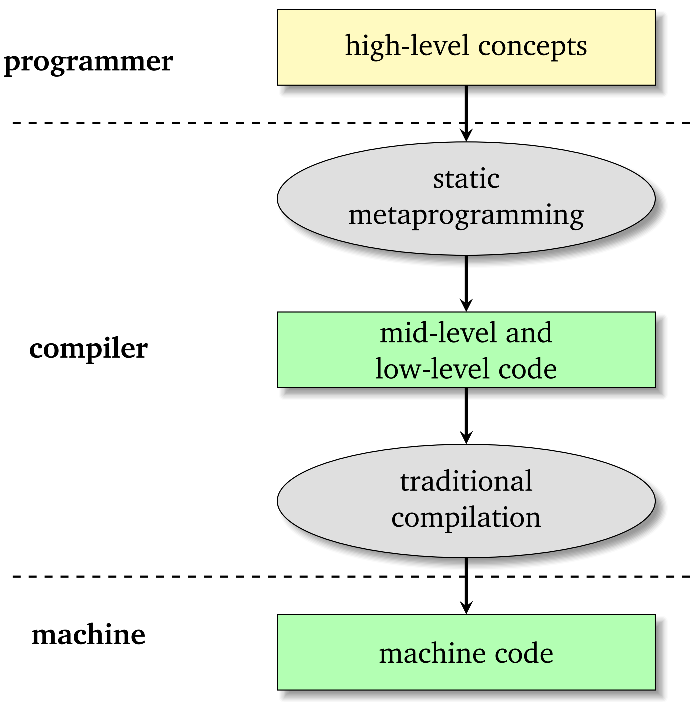


Figure 4.2: Translating high-level concepts into machine code

- Program transformations and code optimizations at compile-time (the optimization routines that were traditionally performed in later stages of the compilers can now operate over high-level structures of the language)

Flexibility

- Allow code transformations
- Extend the language and the compiler
- Allow building new programming paradigms into the language

Naturalness

- Add new programming paradigms better suited to solve certain problems
- Allow the creation of higher-level concepts (the compiler will take care of the details and produce efficient code—see fig. 4.2)
- Allow the compiler to take care of the interconnections between various components

The C++ experience shows us that some of these items can be achieved with the use of templates [VJ02; CE00]; however, more advanced items (e.g., program transformation) cannot. Our metaprogramming functionality must be improved significantly if we are to tackle all of these problems.

To showcase how metaprogramming can *glue* together these three goals, we will discuss the case for program transformation. Having a program expressed in high-level terms (or in a programming paradigm that is not directly supported by the language), metaprogramming can be used to transform the problem into mid-level and low-level constructs (typically found in C++ programs) that can be further transformed by the compiler into machine code. This process is depicted in fig. 4.2. A more capable metaprogramming system allows for a larger variety of high-level concepts, and for more efficient code to be produced.

Simply allowing program transformation in general (without increasing language complexity) is an improvement in flexibility over traditional programming languages, as it can increase the variety of programs accepted by the language.

We can apply this transformation ability to high-level concepts, that are easier to grasp and operate with from a user perspective. The naturalness will increase if the high-level concepts are closer to the programmer's way of thinking, and if they are expressed in an appropriate paradigm.

A well-crafted program transformation, while providing the user with an easy-to-use interface, can at the same time generate efficient code. The creator of the program transformation can theoretically implement the best transformation algorithm possible, yielding an optimal generated code (much better than what a regular programmer would write). Moreover, if the transformation is built on top of efficient lower-level primitives, the resulting code tends to be efficient, even if the transformation itself is not optimal.

This is how a powerful metaprogramming system can make the language accept a large variety of programs, expressed in high-level terms closer to the programmer's way of thinking, and translate it into machine code. If the high-level abstractions can be modeled using functions, then generic programming is enough to model the transformation to efficient code. Otherwise, the metaprogramming component must do Abstract Syntax Tree (AST) manipulations.

To conclude, programming transformations can extend the scope of a programming language in new directions, can provide its users with an easy-to-use interface, and at the same time generate optimal code.

4.2 Hyper-metaprogramming

4.2.1 Context and definition

Although there are several languages with support for metaprogramming, this is not yet a mainstream feature. It is usually seen as an esoteric feature, and often suffers in terms of applicability, convenience, and performance [She01; Dub13; Por10].

Such is the case with template metaprogramming in C++ [Cpp11; AG04; VJ02; Por10; Dub13]. Initially designed for generic library components [Str07], templates were eventually shown to be Turing complete [Vel03]; theoretically they allow a variety of computations to be performed at compile-time. Nevertheless, the compile-time computational system from C++ turned out to be difficult to apply in practice, for several reasons: [Dub13; AG04; Por10]

- performing computations at compile-time has a high impact on compilation time
- even simple computations at compile-time require the programmer to write a large amount of code, with complex structures and cumbersome syntax
- the metaprograms are typically written in a purely functional style, while the rest of the program is typically written in an imperative style
- C++ templates were simply not designed to be used to perform computations at compile-time

In our opinion the biggest problem with C++ metaprogramming is that writing metaprograms in C++ is fundamentally different from writing traditional programs. To reduce this kind of disparity we introduce *hyper-metaprogramming* as an extension of Turing complete static metaprogramming.

We say that a programming language supports hyper-metaprogramming if the following conditions are met:

- the language supports Turing-complete static metaprogramming
- metaprograms can be written using the same abstractions and constructs as traditional programs
- the syntax of metaprograms is identical to the syntax of regular programs, excepting the syntax elements where the programmer specifies whether computations need to be used at compile-time or at run-time
- for a given construct, the compile-time semantics should resemble the semantics corresponding to run-time execution; there can only be differences related to the different execution environments between the compile-time and run-time worlds
- a programmer can write data structures and algorithms that work both at compile-time and run-time without duplicating the code

In a nutshell, the restrictions that hyper-metaprogramming imposes on the programming language allow programmers to write code that is executed at compile-time in the same way they would write code for run-time. As any algorithm and data structure can be executed at compile-time, one can build and execute arbitrarily complex programs at compile-time.

The Sparrow programming language is designed to support hyper-metaprogramming.

Hyper-metaprogramming is defined independently of the programming language, with Sparrow being just an example of a language that implements this concept. We found that, besides Sparrow, languages like Racket and Tem-

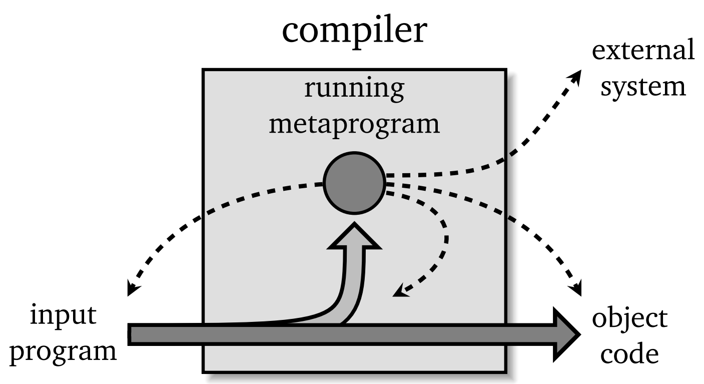


Figure 4.3: Hyper-metaprogramming possible interactions

plate Haskell also implement hyper-metaprogramming. Still, to our knowledge, Sparrow is the only imperative language to support it.

For hyper-metaprogramming to be useful, one needs to get the results of the compile-time computations and use them at run-time. These results can be simple data (numbers, strings), complex data (vectors, maps, etc.), or even code to be executed at run-time. Note that hyper-metaprogramming does not pose any restrictions on how the programming language will perform this movement of data or code from compile-time to run-time. However, it is safe to assume that a language that implements hyper-metaprogramming also has some strategies for generating run-time values and code from compile-time information.

The Sparrow compiler knows how to translate values of basic types from compile-time to run-time, it has a default behavior for values of user-defined types, and moreover the user has means to implement its custom transformations. Basic instructions are provided to be able to generate code from compile-time data. More information can be found in chapter 10.

By using hyper-metaprogramming in a compiler, we can interact with the following elements during the compilation (fig. 4.3):

- the input program itself
- the generated object code
- compilation procedures
- any other external system

Therefore we can control with hyper-metaprogramming the entire compilation process and its environment. We can perform at compile-time computations that are traditionally performed at run-time, we can implement additional compiler features (like preprocessors or code optimizers), and we can even implement the compilation of a different language (implementing a full compiler as a metaprogram), adding a high degree of flexibility to the compiler.

```
1  fun[autoCt] fact(n: Int): Int {
2      if ( n == 0 ) return 1;
3      else return n*fact(n-1);
4  }
5  fun testFact() {
6      var n = 5;
7      writeLn(fact(n));
8      writeLn(fact(5));
9  }
```

Listing 4.1: Sparrow code to compute factorial both at run-time and compile-time

```
1  define void @testFact() {
2      %n = alloca i32
3      call void @Int_ctor_Int(i32* %n, i32 5)
4      %1 = load i32* %n
5      %2 = call i32 @fact(i32 %1)
6      call void @writeLn(i32 %2)
7      call void @writeLn(i32 120)
8      call void @Int_dtor(i32* %n)
9      ret void
10 }
```

Listing 4.2: LLVM translation for the Sparrow code to compute factorial both at run-time and compile-time

4.2.2 A simple example

Let us consider the Sparrow program that computes factorials of integer numbers depicted in listing 4.1

When we call (at run-time) the function `testFact`, it will print `120` twice on the console, as one would expect[7]. But there is a significant difference between the computations that take place at lines 7 and 8. Let us examine the LLVM code produced by the Sparrow compiler for this program—see listing 4.2

The `writeLn` function calls on lines 6 and 7 correspond to the two function calls to `writeLn` from the Sparrow program. Note that for the first function call we compute the factorial by calling the `fact` function (line 5). In the second call to `writeLn` the reader can see that the compiler inserts the integer constant `120` directly. This is a direct effect of the compile-time evaluation.

The most important part here is the way the `fact` function was defined: the `[autoCt]` modifier was used. This tells the compiler that this function should be used both for run-time and compile-time, and the choice should be made based on the arguments. If all the arguments of the function are

[7]we assume there is a function `writeLn` that can print an integer to the console

available at compile-time, then the compile-time version is called. Otherwise the run-time version is called.

The compiler is able to execute at compile-time the second invocation of the `fact` function and will act as if the user had written the `120` constant directly in the code.

As can be seen from this small example, hyper-metaprogramming can be easily used to execute code at compile-time.

4.3 Reduction mechanism and the general compiler flow

4.3.1 Architecture overview

A typical compiler is composed of two parts, a *frontend* and a *backend*, with the following processing phases: parsing, semantic analysis and intermediate code generation for the frontend, and optimizations and machine code generation for the backend [Mak09; ALSU06; GJLB00; CT11]. The Sparrow compiler uses Bison and Flex [Lev09] to parse the input source code. The result of this parsing is a complex Abstract Syntax Tree (AST) that is then *reduced* to a simpler version of AST, and then transformed to an intermediate representation that the backend can recognize. For the backend, the current compiler implementation uses the LLVM framework [Lat02; LLVM] as it provides a solid set of optimization phases and can generate code for a wide range of target machines. Figure 4.4 presents the general flow in the Sparrow compiler.

A Sparrow program consists of both run-time (RT) and compile-time (CT) constructs; it can also have constructs that run both at compile-time and at run-time. Therefore we have three evaluation modes: RT only, CT only, and both RT and CT. To support these modes, the compiler creates in the backend two LLVM modules, one for holding run-time code and one for compile-time code. The compiler should ensure the placement of each code construct in the right LLVM module (section 4.3.3).

To implement hyper-metaprogramming, in addition to implementing Turing-complete compile-time metaprogramming, the language ensures that the compile-time computations are performed in the same way as traditional run-time computations. Both the syntax and the semantics should be (almost) the same. Therefore the compiler must implement this duality as the basic *modus operandi*. The compiler should guarantee that any complex language feature is usable both at run-time and at compile-time, and moreover that the semantics are the same. For this, the compiler implements a *reduction mechanism*: it attempts to reduce complex constructs to a series of simpler constructs that are guaranteed to work similarly for both compile-time and run-time.

The compiler defines a basic set of AST (Abstract Syntax Tree) node kinds for which the compile-time and run-time behavior is easily implemented, and

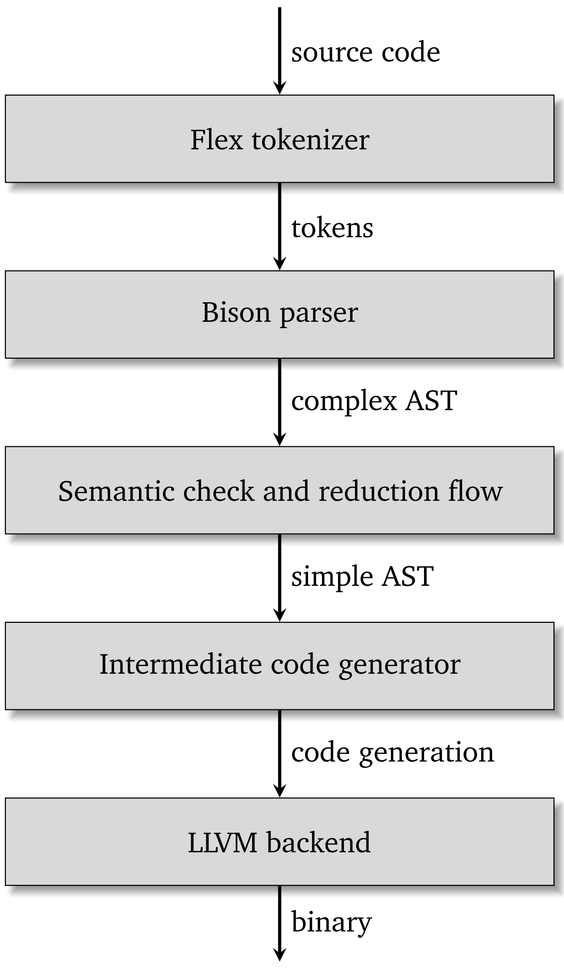


Figure 4.4: General compiler flow

the semantics are simple and easy to follow. Subsequently, the actual AST nodes corresponding to complex language constructs are expanded into these basic nodes that the compiler handles easily. There is a perfect equivalence between the complex AST nodes and their corresponding expansion in terms of basic nodes, and therefore the compiler will use the basic nodes for further processing. This mechanism is important in hyper-metaprogramming as it simplifies the compile-time and run-time interactions, confining them to a small core of the compiler.

The compile-time code needs to be interpreted by the compiler during the actual compilation. The compile-time LLVM module is also interpreted as the program compilation unfolds. Consequently, the compiler has to support two operations on the compile-time module: adding new declarations (classes, functions, variables) and evaluating expressions. To evaluate an expression the compiler wraps it in an LLVM function and then calls the LLVM interpreter on that function; the result is put in a special node that can hold arbitrary objects of different types.

4.3.2 Program structure

All programs can be viewed as abstract syntax trees if we ignore the lexing and parsing processes. Therefore, we will describe the language and the compiler for the language in terms of the AST.

Let $\mathcal{N}$ be the set of all possible nodes in the abstract syntax tree. The nodes depend on their sub-nodes, so when we consider a node, we also consider its sub-nodes as integral parts. We denote by $sub(n)$ the list of all the sub-nodes of node n $(sub(n) \subset \mathcal{N})$. Also let $\overline{sub}(n) = sub(n) \cup \{n\}$.

In our representation the entire program will be a node $n_{program} \in \mathcal{N}$.

Formally, for a program p to be well-formed, the following two functions must be defined for all sub-nodes:

$$type : \overline{sub}(p) \to \mathcal{T}$$
$$translate : \overline{sub}(p) \to \mathcal{B}$$

Here, $\mathcal{T}$ is the set of all the types supported by the language (section 4.3.3). A node has a valid type if and only if it passes all the semantic checks dictated by the language semantics.

The $\mathcal{B}$ set defines the possible intermediate representation (IR) instructions of the backend. Having a correspondence from each node of a program to a backend construct means that we are able to translate the program into the IR. A typical compiler backend can produce actual machine code from this representation.

If we are able to define these two functions for any node in a program p, then, assuming there are no compile-time computations involved, the program is represented by the machine code $translate(p)$.

4.3.3 Evaluation modes and types

The evaluation mode is one of the key concepts that are used in the Sparrow compiler to implement hyper-metaprogramming. The evaluation modes are defined by the following set:

$$\mathcal{M} = \{\texttt{rt}, \texttt{ct}, \texttt{rtct}\}$$

Such an evaluation mode can be determined for each node (through the type of the node). A node that has an associated `rt` mode will only be used at run-time. Similarly a node with `ct` evaluation mode is only used at compile-time. A node with an associated `rtct` evaluation mode can have meaning for both run-time and compile-time.

The evaluation mode of a node is reflected in the type of the node.

We also define the following binary operator:

$$\mapsto: \mathcal{M} \times \mathcal{M} \rightarrow \texttt{bool}$$

that holds only for the following:

$$\begin{array}{ll} \texttt{rt} \mapsto \texttt{rt} & \texttt{rtct} \mapsto \texttt{rtct} \\ \texttt{ct} \mapsto \texttt{ct} & \texttt{rtct} \mapsto \texttt{rt} \\ & \texttt{rtct} \mapsto \texttt{ct} \end{array}$$

For any other pairs, $\mapsto$ returns `false`.

All AST nodes must have a well defined type in order for the program to be well defined. The middleware defines 7 type categories: `Void`, `Data`, `Reference`, `LValue`, `Array`, `Concept`, and `Function`. Formally, the set of types $\mathcal{T}$ can be defined recursively as:

If $m \in \mathcal{M}$ then V_m is a type in $\mathcal{T}$ (`Void` types).

Let $\mathcal{T}_D$ be the set of all non-`Void` types: $\mathcal{T}_D = \mathcal{T} - V_m, \forall m \in \mathcal{M}$

If $m \in \mathcal{M}$ and C is a class declaration with evaluation mode m_C, with $m_C \mapsto m$ then $D[C]_m$ is a type in $\mathcal{T}$. (`Data` types)

If $m \in \mathcal{M}$ and C is a concept declaration with evaluation mode m_C, with $m_C \mapsto m$ then $P[C]_m$ is a type in $\mathcal{T}$. (`Concept` types)

If $t \in \mathcal{T}_D$ and $n \in \mathbb{N}$ then $R[t]$, $LV[t]$ and $A[t, n]$ are also types in $\mathcal{T}$ (`Reference`, `LValue` and `Array` types).

If $m \in \mathcal{M}$, $r \in \mathcal{T}$ and $p_1, ..., p_n \in \mathcal{T}_D, \forall n \in \mathbb{N}$ with $r \mapsto m$ and $p_i \mapsto m, \forall i \in \mathbb{N}$ then $F[r, p_1, ..., p_n]_m$ is also a type in $\mathcal{T}$. (`Function` types)

This is a basic set of types that can be used to lay foundation for an entire language. Other types can be defined on top of these to semantically check more complex language features, but the compiler requires that the final program (after reduction, see section 4.3.5) only use the types defined here[8].

The astute reader may have noticed that there are no traditional built-in types. Indeed, Sparrow treats basic numeric types and the string type as `Data` types. They are all introduced by class declarations, but with some special properties (native names).

Given a well defined type $t \in \mathcal{T}$, we can always find an associated evalua-

[8]actually, `Concept` type is specific only to the Sparrow frontend and should not be present after reduction

tion mode $m \in \mathcal{M}$ defined by the function:

$$\begin{aligned}
evalMode &: \mathcal{T} \to \mathcal{M} \\
evalMode(V_m) &= m \\
evalMode(D[C]_m) &= m \\
evalMode(P[C]_m) &= m \\
evalMode(R[t]) &= evalMode(t) \\
evalMode(LV[t]) &= evalMode(t) \\
evalMode(A[t, n]) &= evalMode(t) \\
evalMode(F[r, p_1, ..., p_n]_m) &= m
\end{aligned}$$

If each AST node has a well-defined type, then we can define the $evalMode$ function for nodes:

$$\begin{aligned}
evalMode &: \mathcal{N} \to \mathcal{M} \\
evalMode(n) &= evalMode(type(n))
\end{aligned}$$

4.3.4 The basic nodes

Let $\mathcal{K}$ be the set of all possible node kinds and let $\mathcal{N}_k$ be the set of all nodes of kind $k \in \mathcal{K}$. Examples of node kinds are: `Class`, `Function`, `If`, `FunCall`, etc. Instead of defining compilation rules for all the node kinds that the Sparrow programming language supports (or may support in the future), we define the compilation rules only for a subset of these node kinds, $\mathcal{K}_0$; for all other node kinds we expand them into nodes with kinds in $\mathcal{K}_0$ and the use the compilation rules defined for $\mathcal{K}_0$ (this will be covered in detail in section 4.3.5).

Let us denote as $\mathcal{N}_\perp$ the set of all nodes with kinds in $\mathcal{K}_0$; formally $\mathcal{N}_\perp = \{n | n \in \mathcal{N}_k, k \in \mathcal{K}_0\}$. We will call these *basic nodes*. In the current version of Sparrow the basic nodes are:

- General: `Nop`, `NodeList`, `LocalSpace`, `BackendCode`, `ChangeMode`, `GlobalConstructAct`, `GlobalDestructAct`, `ScopeDestructAct`, `TempDestructAct`, `TypeNode`
- Declarations: `Class`, `Function`, `Var`
- Expressions: `CtValue`, `VarRef`, `FieldRef`, `MemLoad`, `MemStore`, `FunCall`, `FunRef`, `Bitcast`, `Null`, `StackAlloc`, `Conditional`
- Statements: `If`, `While`, `Break`, `Continue`, `Return`

These nodes are designed to be as simple as possible; they do not cover complex functionality present in high-level programming languages. For example, they do not include name lookup and type conversions. However they are designed to provide the means of implementing more complex constructs on top of them. These nodes can easily act as the foundation of a language like C.

We assume that for these nodes we can define the following functions:

$$type_0 : \mathcal{N}_\perp \to \mathcal{T}$$
$$translate_0 : \mathcal{N}_\perp \to \mathcal{B}$$

These two functions correspond to the $type$ and $translate$ functions required for nodes in order to compile the program. In fact, section 4.3.5 defines the $type$ and $translate$ functions as being equal to $type_0$ and $translate_0$ for basic nodes.

4.3.5 Reduction mechanism

We say that two nodes n_1 and n_2 are equivalent if their semantics are exactly the same: $type(n_1) = type(n_2)$ and $translate(n_1) = translate(n_2)$. We denote the equivalence in the following way: $n_1 \equiv n_2$.

Assume that we need to compile $p \in \mathcal{N}$, a perfectly valid Sparrow program, in which we have both basic and non-basic nodes: $\overline{sub}(p) \not\subset \mathcal{N}_\perp$. We need to transform this program p into an equivalent program $p' \equiv p$, for which $\overline{sub}(p') \subset \mathcal{N}_\perp$. As $p' \equiv p$, we have $translate(p') = translate(p)$, thus we can safely use p' instead of p. We will investigate here a method for performing this transformation.

Let us define the following function for *explaining* a node in terms of another node:

$$expl : \mathcal{N} \to \mathcal{N} \cup \{\texttt{none}\}$$

Here, `none` is a placeholder value that indicates that the node is not explained in terms of another node.

If $expl(n_1) = n_2$, where $n_1, n_2 \in \mathcal{N}$, then the compiler will treat n_1 as being equivalent to n_2 ($n_1 \equiv n_2$).

In practice we define a function $expl_k$ for each node kind k. However, for simplicity we assume here that there is only one function generically defined over the node set $\mathcal{N}$.

If the $expr$ function is defined in such a way that it always reduces the given node to a more primitive node, we can also define $\overline{expl} : \mathcal{N} \to \mathcal{N}$ as:

$$\overline{expl}(n) = \begin{cases} \overline{expl}(expl(n)) & \text{if } expl(n) \neq \texttt{none} \\ n & \text{if } expl(n) = \texttt{none} \end{cases}$$

For a node to be valid, we require $\overline{expl}(n)$ to converge. Any program that contains a node n for which $\overline{expl}(n)$ does not converge is not valid.

For basic nodes, the $expl$ function is defined in the following way:

$$expl(n) = \texttt{none}, \forall n \in \mathcal{N}_\perp$$

Therefore $\overline{expl}(n)$ always converges for basic nodes, and moreover:

$$n = \overline{expl}(n), \forall n \in \mathcal{N}_\perp$$

Only basic nodes can be handled directly by the compiler; therefore when creating new nodes one must make sure that they are always explained in terms of basic nodes. As a result, for a program p to be well-formed, it is required that $\overline{expl}(p)$ is convergent and $\overline{expl}(p) \in \mathcal{N}_\perp$. Otherwise the program is ill-formed and should be rejected by the compiler.

Note that for a program p, although it is required that for each node $n \in \overline{sub}(p)$, $\overline{expl}(n)$ be convergent, it is not required that $\overline{expl}(n) \in \mathcal{N}_\perp$. We could have internal nodes that are not explained in terms of basic nodes and are used solely as helper nodes for other types of nodes. These nodes will not be included in the final program.

Now, with the help of the $\overline{expl}$ function we can generalize the $type_0$ and $translate_0$ functions and actually define the $type$ and $translate$ functions that are required for nodes:

$$\begin{aligned} type(n) =& type_0(\overline{expl}(n)) \\ translate(n) =& translate_0(\overline{expl}(n)) \qquad\qquad , \forall n \in \mathcal{N} \end{aligned}$$

Therefore, assuming that the $expl$ functions are well defined for each node kind, we have defined the $type$ and $translate$ function for any node. This yields a method of reducing any valid Sparrow program to a simple representation.

4.3.6 Run-time and compile-time

We described how a program p can be transformed into an IR code by using the $translate()$ function in conjunction with the reduction mechanism. At some point (run-time or compile-time) this code needs be executed. The compiler backend must ensure that the IR code can be executed in a given environment. If p is a program, f is a function in that program, and e is an execution environment, the backend must ensure that there is a way to define a function $exec[f, p, e]()$ that takes the arguments defined by f and returns an object defined by the return type of f. This can be done through an interpreter or by translating the code into machine code; it is not relevant how this is implemented, as long as one can actually execute the specified function of the program.

As Sparrow implements hyper-metaprogramming, the compiler must handle two execution environments: one for compile-time (e^{ct}) and one for run-time (e^{rt}). One is used during compilation, and the other is used after the compilation is finished, when running the resulting run-time program. To this effect, the Sparrow compiler constructs from the initial program p two new programs p^{rt} and p^{ct}, to handle the compile-time and run-time cases differently.

When translating a program p into backend code the compiler enforces some rules regarding code organization. For example, one cannot have a program without any declarations, that consists only of a memory load outside a function. These rules are enforced in the type checking phase, before translation.

The compiler clearly defines the kinds of nodes that can appear at the top-level of a backend module. In the current implementation the top-level node kinds are: `Nop`, `NodeList`, `GlobalConstructAct`, `GlobalDestructAct`, `Class`, `Function`, `Var`. Let us denote these node kinds as $\mathcal{K}_T$ (where $\mathcal{K}_T \subset \mathcal{K}_0$), and as $\mathcal{N}_T$ all the possible nodes of kinds in $\mathcal{K}_T$. Also, the compiler defines the node kinds that can appear in a function body (in-function nodes); these are expressions, statements and the following node kinds: `Nop`, `NodeList`, `LocalSpace`, `ChangeMode`, `Var`, `ScopeDestructAct`, `TempDestructAct`. We denote these node kinds by $\mathcal{K}_F$ ($\mathcal{K}_F \subset \mathcal{K}_0$), and by $\mathcal{N}_F$ all the possible nodes of kinds in $\mathcal{K}_F$.

Let us define how these node kinds are translated and added to the backend in the appropriate program and execution environment.

Let p be the original program that needs to be compiled, and $p' = \overline{expl}(p)$ the program expressed only in basic nodes. Let n be a node with $s_1, ..., s_m$ its direct sub-nodes; we denote the node as $n(s_1, ..., s_m)$ to indicate that the node n depends on its direct sub-nodes $s_1, ..., s_m$. This has the advantage of showing that we can change the node n by changing one of its direct sub-nodes.

We define a function $tr_{rt} : \mathcal{N} \to \mathcal{N}$ to transform the nodes from a program as written in the source code to the nodes that belong to the run-time program p^{rt}. This function is defined as

$$tr_{rt}(n) = \begin{cases} ctEval'(n) & \text{if } evalMode(n) = \texttt{ct} \\ n(tr_{rt}(s_1), ..., tr_{rt}(s_m)) & \text{otherwise} \end{cases}$$

If all the nodes in a program p' are run-time ($\forall n \in \overline{sub}(p'), evalMode(n) \neq \texttt{ct}$), then the run-time program will be the same as the original program: $p^{rt} = p'$. If the program contains some `ct` in-function nodes, the compiler tries to *compile-time evaluate* them:

$$ctEval'(n) = \begin{cases} ctEval(n) & \text{if } n \in \mathcal{N}_F \\ \texttt{Nop} & \text{otherwise} \end{cases}$$

The top-level `ct` nodes are ignored when building p^{rt}.

In order to describe the $ctEval$ function we need to describe the tr_{ct} function first:

$$tr_{ct} : \mathcal{N} \to \mathcal{N}$$

$$tr_{ct}(n) = \begin{cases} n & \text{if } evalMode(n) \mapsto \texttt{ct}, n \in \mathcal{N}_T \\ \texttt{Nop} & \text{otherwise} \end{cases}$$

Essentially, the top-level definitions in a program that are `ct` or `rtct` are added to the compile-time program.

In addition to top-level compile-time nodes, we need to add a mechanism for evaluating `ct` nodes found in run-time functions. In other words, the compiler needs to implement the $ctEval$ function.

The result of $ctEval(n)$, where n is a compile-time node is a `CtValue` expression if $type(n) \in \mathcal{T}_D$ and a `Nop` expression if $type(n) = V_{ct}$. A `CtValue` expression is very similar to the literals in other languages; they are compile-time values of a given type. In Sparrow these constants are not limited to a small set of types, but apply to all the $\mathcal{T}_D$ of the language. A `CtValue` node contains an object (the bytes that form the object) and its type.

To actually compute the value of the `CtValue` node, the compiler needs to evaluate the given expression, possibly calling other functions and using complex algorithms and data structures. For the evaluation to be performed correctly, the compiler creates a wrapper function over the node and then calls the newly created function. For a node n, depending on whether its type is `Void`, the function will be similar to one of the following:

```
fun[ct] ctEvalImpl_n(): type(n) { return n; }
fun[ct] ctEvalImpl_n(): Void { n; }
```

With this function defined the compiler can use the $exec[ctEval, p^{ct}, e^{ct}]()$ mechanism to execute the body of the newly created function. If the function returns a value, then that value will be put into the resulting `CtValue` node. This way, every expression with compile-time evaluation mode is executed at compile-time, and the run-time will only use the result of the expression. This can increase the speed of certain programs (see section 4.2.2).

Our description of the mechanisms involved here is slightly simplified in order to focus only on the important concepts. For example the compiler must interleave compile-time top-level nodes with `ctEval` functions.

To be efficient, the Sparrow language demands that the compiler perform computations lazily. When a nested structure containing several compile-time nodes is encountered, the compiler is required to perform only compile-time evaluation if possible. For example the expression `1*2+3*4` can be evaluated as $ctEval(\texttt{1*2+3*4})$ or as $ctEval(ctEval(\texttt{1*2}) + ctEval(\texttt{3*4}))$. In this case the first type of evaluation is required. The evaluation function will be the following:

```
fun[ct] ctEvalImpl(): Int { return 1*2+3*4; }
```

To summarize, the compiler will split the initial program into two independent programs: one for compile-time and one for run-time. The compile-time program will contain all the top-level nodes with `ct` or `rtct` evaluation mode. Also, the compiler creates an additional function for each compile-time expression that needs to be evaluated. The run-time program will contain all the `rt` and `rtct` declarations, and for each in-function node that is compile-time, the node gets replaced with a `CtValue` or `Nop` node obtained after evaluating (or executing) the compile-time node.

The mechanisms described here provide the means of implementing hyper-metaprogramming. We can process both run-time and compile-time programs of arbitrary complexity, and in run-time programs we can call an evaluation mechanism that evaluates compile-time expressions and replaces them with the resulting values.

4.3.7 Reduction example

Let us take a simple Sparrow construct:

```
var a = 10;
```

This declares a variable of type `Int` and initializes it with the value `10`. The Sparrow compiler creates the following AST for the given declaration:

```
SprVariable(name="a", type=null, init=Literal(Int, 10))
```

Depending on the context, this can be expanded into several structures. In our case let us assume that this is a local variable, which means that the previous declaration is expanded into:

```
NodeList(Var(name="a", type=Int),
         FunCall("Int.ctor", VarRef("a"), CtValue(Int, 10)),
         ScopeDestructAct(FunCall("Int.dtor", VarRef("a")),
         Nop() )
```

This example shows that a simple variable declaration that takes only 8 visible characters is actually not trivial, as there are several operations involved.

First, a `NodeList` is created; this node kind acts as a container for the other nodes. The actual variable is created using the `Var` node. Every time a construct tries to refer to that variable, it must use a `VarRef` node. Apart from this, the compiler needs to create two `FunCall` nodes for calling the constructor and the destructor for the variable, and a `ScopeDestructAct` node to defer the destructor call until the end of the scope. Even though the constructor and destructor are defined as members of the `Int` class, when expanding, the function calls take an additional argument corresponding to the object of the class (*this* argument); in our case, the passed argument is a reference to the variable.

In this small code, another expansion takes place: the literal `10`—initially defined by `Literal(Int), 10)`—is expanded into `CtValue(Int, 10)` before passing it to the constructor function call. Literals are always converted to corresponding `CtValue` nodes by taking the type and the actual memory footprint of the literal data.

4.3.8 Discussion

The Sparrow reference compiler that we have implemented follows the technique described here. Of course, the implementation is slightly more complex than the scheme presented here, but the basic principle is the same. For example, because of mutual node dependencies, we use two functions to check if a node is semantically correct[9], instead of one (we named it here $type()$).

By default all the nodes that we create are considered run-time. If we were to consider them `rtct`, then we would have two main problems:

- we would compile for `ct` a large amount of code that will not be used at compile-time
- because an `rtct` code can only invoke `rtct` code, we would need to make everything `rtct`; there would be no run-time only, as it could never be accessed from an `rtct` context

To differentiate between the compile-time and run-time modes, the Sparrow syntax allows so-called *modifiers* to be added to declarations (functions, classes, and variables). The modifiers corresponding to the three evaluation modes are: `[rt]`, `[ct]`, and `[rtct]`. In addition to these three, there is another modifier that can only be applied to functions: `[autoCt]`. A declaration that has the `[rtct]` evaluation mode will be considered run-time or compile-time depending on the context in which it is used. On the other hand, an `[autoCt]` function will have a compile-time evaluation mode if all the parameters are compile-time, regardless of the evaluation mode of the caller. This is how the factorial function from listing 4.1 could be called at compile-time from a run-time context.

Although compile-time execution and evaluation may seem at a first glance to be exotic features of limited use, they are actually ubiquitous in Sparrow. Firstly, all the generics that have type parameters actually work using compile-time expressions of type `Type`, which is of course defined only for compile-time. Secondly, an even more common feature in Sparrow—the references—also use compile-time evaluation. It turns out that `@Int` (reference to an `Int` object) is a prefix operator call implemented as a compile-time function that takes a type and returns a type (`@` operator is applied to value `Int`). Furthermore, each time a variable is declared, the type of the variable is considered a compile-time expression of type `Type`.

Executing algorithms at compile-time has the same complexity as executing them at run-time; the execution speed of metaprograms is proportional to the execution speed of the same programs at run-time. However, as typically the compile-time programs are interpreted, they are usually slower than their run-time counterparts, which are compiled. The compiler can chose to compile and optimize the compile-time code before executing it, but in this case we have a performance penalty for compiling and optimizing the metaprogram

[9] $semanticCheck()$ and $computeType()$

before the actual execution (see chapter 10 for an in-depth discussion about hyper-metaprogramming costs).

4.3.9 Summary

We presented in this chapter the design of the Sparrow programming language, a language that strives to be as simple as possible, while still providing high-level abstractions as library features.

We first described the language's main principles—efficiency, flexibility, and naturalness—as they dictate the general direction for Sparrow's development. We discussed how these principles are apparently contradictory, and how we can employ metaprogramming to integrate them into the same language. Using static metaprogramming techniques, one can help to increase the flexibility, efficiency, and naturalness of programming languages. Moreover, we discussed how static metaprogramming can help to *glue* these principles together in the same language.

We then proceeded to describe hyper-metaprogramming, the feature that allows the user to execute any complex algorithm at compile-time, thus extending the compiler. The way hyper-metaprogramming is defined, the user will be able to write metaprograms that are executed at compile-time with the same ease as they would have been written for run-time. Using hyper-metaprogramming one can control the entire compilation process, thus extending the compiler and the language.

In the last part of the chapter, we described the reduction mechanism. This dictates how the compiler is organized, and how we can implement more complex features on top of simpler features. We provide a theoretical framework for how the reduction mechanism works, and also explain the manner in which hyper-metaprogramming can be implemented in a programming language.

Without translation, I would be limited to the borders of my own country. The translator is my most important ally. He introduces me to the world.

I. Calvino

CHAPTER 5

A SKETCH OF TRANSLATION TO LLVM

This chapter describes the way in which the Sparrow compiler translates Sparrow code into LLVM code. If the translation description were formal, this transformational approach could serve as a language description; however we have chosen to make this semi-formal, ignoring some less important details. We trade formalism for readability.

Our description, in accordance with the reduction mechanism described in the previous chapter, focuses on how individual node kinds are explained into more and more basic node kinds, down to the basic level, where they are translated into LLVM code.

After a short overview of the compiler architecture (section 5.1), we provide the translation of our basic nodes (Feather nodes) to LLVM (section 5.2), and finally describe how the Sparrow nodes translate to these basic nodes (section 5.3).

5.1 Overview

The architecture of the Sparrow compiler is engineered to match the reduction principle. The main architectural components are presented in fig. 5.1.

The *Nest* component provides the basic support for all the components in the system. It defines the AST node abstraction, the symbol table abstraction, and the general compiler flow. Moreover, it provides utilities like location information and error handling routines.

A node object has three main data parts: the children nodes, the referred

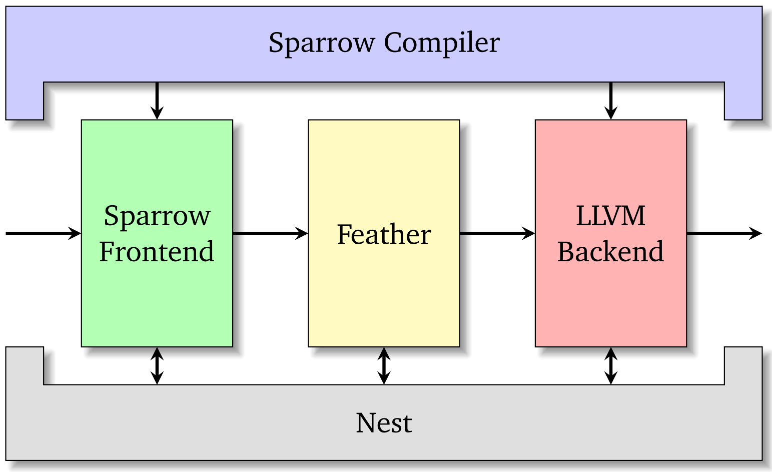


Figure 5.1: The architecture of the Sparrow compiler

nodes, and possibly some attributes. Most of the nodes are defined in terms of other nodes, which correspond to the children nodes. This *contains* relation is similar to a composition relationship in object-oriented programming. For example, a `MemLoad` gets the expression indicating the memory reference to load from. The *refers to* relation corresponding to the second data part of an AST node is weaker than the *contains* relation, and indicates that the current node references another node, without actually containing it; this is similar to the *aggregation* relation in object-oriented programming. For example, a `VarRef` node uses this relation to indicate a variable declaration node; the variable declaration is independent of the existence of the `VarRef` node. The attributes part of the nodes offers an easy method of associating data to the node without using other nodes. An attribute can be a string, a number, a type, or even another node. The attributes are typically used by different node kinds internally to encode the node information.

The basic node implementation also provides support for explaining the node in terms of another nodes. The actual node implementations only need to provide the semantic checks and possibly the explanation rules.

The Sparrow Frontend component is responsible for compiling Sparrow code. It performs the lexing and parsing of the source code, and translates it to Sparrow-specific nodes. All the language constraints are expressed in this component. These nodes will always be explained in terms of Feather nodes.

The Feather component provides the basic set of node kinds that can be translated to machine code. They correspond to the $\mathcal{N}_\perp$ node kinds from the previous chapter. They are designed to cover the basic functionality of a generic programming language, and do not include any Sparrow-specific features. For example, one can write a C frontend whose nodes are explained into Feather nodes.

The LLVM Backend is responsible for translating the Feather nodes into LLVM code, and further into native machine code. It contains two execution environments (implemented as LLVM modules), one corresponding to compile-time and one to run-time. The code that reaches the compile-time execution environment is interpreted by the compiler. The code from the run-time execution environment gets compiled, optimized, and transformed into native machine code. The LLVM Backend component can only understand Feather node kinds; it does not know any Sparrow-specific abstractions.

Finally, the Sparrow Compiler component acts as a *compiler driver*. It provides the general compiler flow, making sure that the programs are passed from frontend to backend to produce the final compiled binaries.

The following sections provide descriptions for the main functionalities of the different components. We focus on how the nodes get translated, so that the reader can understand how complex Sparrow structures can be translated into efficient LLVM code. The reader should note that we sacrificed formalism in order to favor readability; we try to describe the main mechanism involved in translation rather than focusing on providing a complete translation scheme. For example, we leave out from our explanation features such as macros, access modifiers, inheritance, as well as some corner-cases related to run-time/compile-time interaction.

5.2 Feather nodes

5.2.1 General nodes

BackendCode

A `BackendCode(code, evalMode)` node translates directly into the given code for the specified evaluation mode. The code is a text conforming with the LLVM IR specification [LLVM]. The code must reflect top level LLVM definitions (e.g., functions, data structures, global variables, and global constants). This node is used to implement some of the most fundamental operations in Sparrow, as it allows injecting low-level code into the compiler.

On top of this node, Sparrow adds three functions:

```
fun[ct, native("$injectBackendCodeRt")]
                 injectBackendCodeRt(code: StringRef);
fun[ct, native("$injectBackendCodeCt")]
                 injectBackendCodeCt(code: StringRef);
fun[ct, native("$injectBackendCodeRtCt")]
                 injectBackendCodeRtCt(code: StringRef);
```

These are implemented as compiler intrinsics and map directly to the `BackendCode` nodes. Below is an example of injecting a primitive operation like integer multiplication:

```
injectBackendCodeRtCt("
    define i32 @mulInt(i32 %x, i32 %y) {
      %1 = mul i32 %x,%y
      ret i32 %1
    }
");
fun[native("mulInt"), autoCt] *(x,y: Int): Int;
```

Nop

A `Nop` node translates to nothing in LLVM. We use this node for expressions and declarations that do not expand into something directly visible in the code. For example, a compile-time `If` instruction will expand to `Nop` if the condition (evaluated at compile-time) is false and there is no else clause.

NodeList

A `NodeList(child1, child2, ...)` node is translated to the concatenation of the translations of all its children. This node is extensively used to create more complex nodes out of simple nodes. The actions for the simpler nodes will simply be chained together.

LocalSpace

A `LocalSpace(instr1, instr2, ...)` node is translated very similarly to the `NodeList` node. The only difference is that it creates a local scope around all the children and treats each children as an instruction. This means that all the temporary destruct actions are translated just after an instruction is translated, and at the end the scope destruct actions are applied.

ScopeDestructAct

A `ScopeDestructAct(act)` node has the same translation as its argument `act`. The only purpose of this node is to indicate to the backend the actions (typically variable destructions) that need to be applied at the end of a lexical scope.

For example, for a local variable we need to call the destructor at the end of the scope. For this, we create a node of the `ScopeDestructAct` kind, passing it a function call node to the appropriate destructor. The end of the scope for the variable will be indicated by the end of the `LocalSpace` node surrounding it.

For example, the following code:

```
{
    var a = 10;
    writeLn(a);
}
```

is represented (after expansion) by:

```
LocalSpace(
    NodeList(
        Var(name="a", type=Int),
        FunCall("Int_ctor", VarRef("a"), CtValue(Int, 10)),
        ScopeDestructAct(
            FunCall("Int_dtor", VarRef("a"))
        ),
        Nop()
    ),
    FunCall("writeLn", VarRef("a"))
)
```

and will be translated into something similar to:

```
%a = alloca i32
call void @Int_ctor(i32* %a, i32 10)
%1 = load i32* %a
call void @writeLn(i32 %1)
call void @Int_dtor(i32* %a)
```

The reader should note how the destructor call was made after the call to the `writeLn` function.

TempDestructAct

Similar to `ScopeDestructAct`, this node just translates to the given action. The backend will recognize these actions and will inject the appropriate code whenever an instruction is finished. Let us look at an example:

```
class Foo {...}
fun getFoo(i: Int): Foo = ...;
fun test {
    combineFoo(getFoo(1), getFoo(2));
    writeLn(123);
}
```

The body of this function will be represented similarly to:

```
LocalSpace(
    FunCall("combineFoo",
        NodeList(
            Var(name="tmpVar1", Type=Foo),
            FunCall("getFoo", CtValue(Int, 1)),
            TempDestructAct(FunCall("Foo_dtor", VarRef("tmpVar1"))),
            VarRef("tmpVar1")
        ),
        NodeList(
            Var(name="tmpVar2", Type=Foo),
            FunCall("getFoo", CtValue(Int, 2)),
            TempDestructAct(FunCall("Foo_dtor", VarRef("tmpVar2"))),
            VarRef("tmpVar2")
        )
    ),
    FunCall("writeLn", CtValue(Int, 123))
)
```

This may be translated to:

```
%tmpVar1 = alloca %Foo
call void @getFoo(%Foo* %tmpVar1, i32 1)
%tmpVar2 = alloca %Foo
call void @getFoo(%Foo* %tmpVar2, i32 2)
call void combineFoo(%Foo* %tmpVar1, %Foo* %tmpVar2)
call void @Foo_dtor(%Foo* %tmpVar1)
call void @Foo_dtor(%Foo* %tmpVar2)
call void @writeLn(i32 123)
```

The reader should note how the destruct actions are applied immediately after the current instruction is done, but not earlier. This way, Sparrow makes sure to keep the temporary objects in memory while needed.

GlobalConstructAct and GlobalDestructAct

We covered how one can create actions to be applied when the current instruction and the current scope is finished. There is one more important case to handle for the delayed destruct actions: the destruct actions that need to be executed at the end of the program. These are useful for examples to call the destructor for global variables.

For scope and temporary destruct action we did not have a need to have a dedicated node to specify the corresponding construct actions. These action could be directly encoded into the node as regular actions, and they were translated in the place in which the action was used. However for global variables we need to be able to indicate which construct action needs to be executed, and we cannot specify this when the LLVM global variable is translated. We therefore need to have an additional node `GlobalConstructAct`. The action passed to this node will be invoked when the program starts, before invoking the main function.

Let us take an example of how a global variable has its construct and destruct actions translated:

```
NodeList(
    Var(name="myGlobal", type=Foo)
    GlobalConstructAct(FunCall("Foo_ctor", VarRef("myGlobal"))),
    GlobalDestructAct(FunCall("Foo_dtor", VarRef("myGlobal")))
)
```

This gets translated into something similar to:

```
@myGlobal = global %Foo zeroinitializer
define private void @__global_ctor1() {
  call void @Foo_ctor(%Foo* @myGlobal)
  ret void
}
define private void @__global_dtor2() {
  call void @Foo_dtor(%Foo* @myGlobal)
  ret void
}
@llvm.global_ctors = appending global [1 x { i32, void ()* }] [
    { i32, void ()* } { i32 0, void ()* @__global_ctor1 }
]
@llvm.global_dtors = appending global [1 x { i32, void ()* }] [
    { i32, void ()* } { i32 0, void ()* @__global_dtor2 }
]
```

As the global variables need to be specified at top-level in LLVM, and as our actions are expressions that we need to call, the actions get wrapped into some functions. Then, these functions will be introduced into two LLVM-specific tables (`@llvm.global_ctors` and `@llvm.global_dtors`). If we have multiple construct/destruct actions, we should provide an index number for the actions to be executed; in our case, the passed index is `0` both for constructor and destructor actions.

Both the `GlobalConstructAct` and `GlobalDestructAct` nodes have different meanings if the given action has meaning only at compile-time. In the case of `GlobalConstructAct`, the action is executed as soon as the node is semantically checked (invoking the *ctEval* mechanism discussed in section 4.3.6). For `GlobalDestructAct` the destruct action will be ignored (it does not make much sense to execute compile-time actions when the compilation is ending).

TypeNode

This node is never translated directly to LLVM. It is used as a wrapper over a type, to be used in other nodes that rely on types.

ChangeMode

Again, this node is not translated directly to LLVM. Is is used to change the evaluation mode (`rt`, `ct` and `rtct`) in the current program context. All the sub-nodes will be translated as usual, but in a different context.

5.2.2 Declarations

Class

A class node has a name, a list of fields, and an evaluation mode. All the fields should be of kind `Var`. Here is an example of a class node:

```
Class(name=MyClass, fields=NodeList(
        Var(name="field1", Type=Int),
        Var(name="field2", Type=Double),
        Var(name="field3", Type=Foo)
    ), evalMode=rt)
```

This would be translated into something like:

```
%MyClass = type { i32, double, %Foo }
```

If the class has `native` modifiers, then it gets translated into a native LLVM type. If N is the number of bits of a type, then Sparrow supports the native names of `i`N and `u`N (signed and unsigned numeric type with N bits). As LLVM does not distinguish between signed and unsigned types, the unsigned types will be translated int the same way as signed types (e.g., `u32` will be translated into `i32`). In addition to integer types, we also support `float` and `double` as native names for classes, to represent single and double precision floating point numbers.

Here is a list of basic types that Sparrow defines in the standard library[1]:

```
class[native("i1"), noDefault, rtct] Bool {...}
class[native("i8"), noDefault, rtct] Byte {...}
class[native("u8"), noDefault, rtct] UByte {...}
class[native("i16"), noDefault, rtct] Short {...}
class[native("u16"), noDefault, rtct] UShort {...}
class[native("i32"), noDefault, rtct] Int {...}
class[native("u32"), noDefault, rtct] UInt {...}
class[native("i64"), noDefault, rtct] Long {...}
class[native("u64"), noDefault, rtct] ULong {...}
class[native("u64"), noDefault, rtct] SizeType {...}
class[native("i64"), noDefault, rtct] DiffType {...}
class[native("float"), noDefault, rtct] Float {...}
class[native("double"), noDefault, rtct] Double {...}
class[native("i8"), noDefault, rtct] Char {...}
```

Function

A `Function` node will be translated directly into an LLVM function. Here is an example:

[1] the `noDefault` modifier indicates that we should not generate standard constructors and destructor for these types; the `rtct` modifier indicates that these types can be used both at run-time and compile-time

```
Function(name="plus", TypeNode(Int),
    NodeList(Var("a", Int), Var("b", Int)),
    LocalSpace(
        Var(name="res", type=Int),
        FunCall("assignInt", VarRef("res"),
            FunCall("plusInt", VarRef("a"), VarRef("b"))
        )
        Return(VarRef("res"))
    ), evalMode=rt)
```

gets translated into something similar to:

```
define i32 @plus(i32 %a, i32 %b) {
    %a.addr = alloca i32
    store i32 %a, i32* %a.addr
    %b.addr = alloca i32
    store i32 %b, i32* %b.addr
    %res = alloca i32
    br label %code
code:                                             ; preds = %0
    %1 = load i32* %a.addr
    %2 = load i32* %b.addr
    %3 = call i32 @plusInt(i32 %1, i32 %2)
    call void @assignInt(i32* %res, i32 %3)
    %4 = load i32* %res
    ret %4
}
```

There are two important aspects that are revealed by this rather trivial example. The first one is that all the LLVM stack variables are declared at the beginning of the function[2]. Then we create local variables for our parameters. This is needed to guarantee that a variable is something that we can take the address of; for example, the `VarRef` nodes always return pointers to the actual memory referred by the variable.

Our backend implementation renames a function given for translation if a function with that name already exists. This is especially useful for overloaded functions, when different functions with the same name exist, typically in the same package. To bypass this renaming the user can set a fixed LLVM translation name to a function by setting the *native* attribute of the node with the desired name.

The native attribute can also be used to make a connection to functions that are not defined in Sparrow. If we provide a native attribute to a function, we are not required to also provide a body to that function. This way, we just *declare* the function, and rely on LLVM or an external library to implement it. Here are a few examples:

[2]this is done primarily to avoid multiple allocations for the same variables in the case of loops

```
fun[rtct, native("exit")] exit(code: Int);
fun[rtct, native("system")] system(x: @Char): Int;
fun[rtct, native("fopen")] fopen(filename, mode: @Char): @Byte;
```

If a function has a non-trivial return type, the Sparrow compiler will construct the result in a location given as parameter. This parameter will be marked accordingly in the generated LLVM code with the `sret` attribute. Similarly, complex objects will be passed through pointers at low-level; the actual objects will be stored in memory at a unique location. This way, an object can reference itself without pointing to an invalid memory location.

Var

There are four types of variables nodes, distinguished by their context: local variables, global variables, parameters, and class fields. For all these types the corresponding `Var` nodes look the same (this is why we do not have four different node kinds).

A local variable will be translated directly into an `alloca` instruction:

```
%res = alloca i32
```

A global variable will be translated in the following way (at top-level):

```
@globalInt = global i32 0
@globalFoo = global %Foo zeroinitializer
```

The example from the `Function` node kind shows how parameters are defined. Besides the actual parameters defined for the function, we define some local variables that we initialize with the given arguments; we do this because we want to be able to take references to these arguments.

The field variables were shown in the example in the `Class` section. In the case of fields, when translating we discard the variable name, as it is not useful for the LLVM code[3].

5.2.3 Expressions

CtValue

There are multiple ways of translating a value node, based on the type of the value. If the type is an integer or a floating point constant, then the translation is direct. Value `42` of type `Int` gets transformed into `i32 42`, value `3.1415` goes into `double 3.1415`, and so on.

If the value is of type `StringRef` then we have a special translation. In Sparrow, the `StringRef` type is a pair of pointers to the characters of a string,

[3]accessing class fields is done in LLVM with the `getelementptr` instruction, which accesses the fields by their index

one pointer for the beginning of the string and one for the end. Unlike C++ and C, Sparrow can represent strings that contain null characters inside them[4]. Moreover, we can easily take slices of our strings without needing to copy the full string. The following code provides an example on how a `StringRef` constant is translated:

```
define void @testStringConstant() {
    %const.bytes = alloca [5 x i8]
    %const.struct = alloca %StringRef
    br label %code
code:                                               ; preds = %0
    store [5 x i8] c"true\00", [5 x i8]* %const.bytes, align 1
    %2 = getelementptr [5 x i8]* %const.bytes, i32 0, i32 0
    %3 = getelementptr [5 x i8]* %const.bytes, i32 0, i32 4
    %4 = getelementptr %StringRef* %const.struct, i32 0, i32 0
    %5 = getelementptr %StringRef* %const.struct, i32 0, i32 1
    store i8* %2, i8** %4, align 1
    store i8* %3, i8** %5, align 1
    %6 = load %StringRef* %const.struct
    ret
}
```

In LLVM a string constant is represented as an array of bytes. We need to take the pointer to the first character and the pointer to the end of the string, and then store them in the corresponding members of the `StringRef` structure.

If the value is a pointer we translate the pointer as an integer[5] and then use an `inttoptr` instruction to convert it to the actual LLVM pointer type. The pointer constants are especially useful in compile-time code when we can pass pointers to and from the compiler. Here is a small example of a *ctEval* function that returns an `AstNode`, where the pointer is a compile-time constant:

```
define private void @ctEval243(%AstNode* sret) {
  %tmp.v = alloca %AstNode
  br label %code
code:                                               ; preds = %1
  %2 = inttoptr i64 140252279754784 to i32*
  call void @ctor244(%AstNode* %tmp.v, i32* %2)
  %3 = load %AstNode* %tmp.v
  store %AstNode %3, %AstNode* %0
  ret void
}
```

[4]the current implementation of the standard library always adds an extra null element at the end of the strings, for better interoperability with the system functions

[5]currently we only support 64-bit pointers; changing the size of the pointer should not be a major change in the compiler

A constant whose type is a structure with size zero will be translated into a `zeroinitializer` constant.

A non-zero sized structure constant will be translated to an array of bytes and then reinterpreted as the actual structure type. Here is an example of a function that initializes a global variable `valueType` of type `Type` with a constant[6]:

```
define private void @ctEval110() {
    %const.arr = alloca [8 x i8], align 1
    br label %code
code:                                                    ; preds = %0
    store [8 x i8] c"P\9B\82\03\8F\7F\00\00", [8 x i8]* %const.arr
    %1 = bitcast [8 x i8]* %const.arr to %Type*
    %2 = load %Type* %1
    call void @_Type_copy_ctor(%Type* @valueType, %Type %2)
    ret void
}
```

VarRef

A `VarRef` node refers to the value of a variable. The referred variable must not be a class field. If the variable has type `Int`, then a corresponding `VarRef` node will have type `@Int`—we always keep the reference to the variable, as we may need to change the content of the variable.

The `VarRef` is translated to LLVM as the actual name of the variable. There is no new instruction generated.

Note that in our compiler a `VarRef` node is created using a pointer to the actual `Var` declaration. However, in the examples presented here, we pretend that these nodes are constructed from the name of the variables; we do this just for clarity, as it is hard to represent pointers in text.

FieldRef

A `FieldRef(obj, fieldDecl)` node is a reference to a field variable. It is somewhat similar to `VarRef`, but it also needs to take the object from which the field is accessed. The type of the object must be a class type that contains the given variable declaration.

This node will be translated into LLVM using a `getelementptr` expression, applied to the index of the field. Here is such an example of a field access:

```
%obj = alloca %StringRef
%1 = getelementptr %StringRef* %obj, i32 0, i32 0 ; first field
%2 = getelementptr %StringRef* %obj, i32 0, i32 1 ; second field
```

[6] a `Type` constant contains a pointer to the actual type object in the compiler

As in the case of `VarRef`, a `FieldRef` object will be a reference type, in order to be able to manipulate the field value.

MemLoad

A `MemLoad(expr)` will dereference the given expression. The argument must have a type with at least one reference. The type of this instruction is the same as the given expression, but with one reference less.

In LLVM this node will be translated using a `load` instruction, as shown in the following example:

```
%expr = alloca i32       ; type = i32*
%res = load i32* %expr   ; type = i32
```

MemStore

A `MemStore(value, address)` is an operation that stores a value at a given address. The address must be a reference to the type of the value. This gets translated into a `store` LLVM instruction, as shown in the following code:

```
%x = alloca i32          ; an integer variable
store i32 42, i32* %x    ; store value '42' into it
```

Both `MemStore` and `MemLoad` support more advanced features like atomic ordering, volatility, and alignment. They are not yet fully utilized by the Sparrow frontend.

FunCall

A function call node has the form `FunCall(funDecl, args...)`. The number of arguments passed must match the number of arguments of the function declaration, and the types of the arguments must be equal to the types of the function parameters, possibly ignoring the evaluation mode (no conversions are applied at this level).

In order for this node to be type checked the evaluation mode of the arguments must correspond to the evaluation mode dictated by the function declaration. If the function declaration is compile-time, all the arguments must also be compile-time. It is also an error to call an `rt` function from a `ct` or an `rtct` context. If the function is marked as `autoCt` and all the arguments are compile-time, the function call will be made at compile-time.

The resulting type of the function call object will be computed to match the compile-time and run-time execution rules. For example, if the function needs to be run at compile-time, even if the function was declared `rtct`, the resulting type will have the `ct` evaluation mode.

A `FunCall` node will be translated into a simple `call` LLVM instruction, after all the arguments are translated. For example, for the following nodes:

```
FunCall("_ass_64_64", VarRef("sz1"), FunCall("size", VarRef("x"))
```

the following LLVM code would be generated:

```
%1 = call i64 @size(%StringRef* %x.addr)     ; call returns an i64
call void @_ass_64_64(i64* %sz1, i64 %1)     ; no return value
```

The current version of the compiler implements some compiler intrinsics by treating some function calls specially. For example the logical *and* and *or* operations are implemented this way. They are identified by their native names, `$logicalOr` and `$logicalAnd`. We use this method because otherwise we would have to evaluate all arguments, preventing short-circuiting.

There is another special case of the `FunCall` node for calling functions based on pointers, which is covered next.

FunRef

Most of the time we call functions by name. We just create a `FunCall` node specifying the function declaration to be called and the arguments to be used. However, there are cases in which one wants to make the call to a pre-computed function pointer, without knowing exactly which function will be called. For this, one needs to use the `FunRef` node.

A `FunRef(funDecl, resType)` node takes the reference to the given function and casts it to the given type. Of course, the given type must be able to store a pointer to a function. By doing this cast, we allow the user to treat pointers to functions as regular data.

In the Sparrow code we would encapsulate this into a `FunctionPtr` class. This class would have a call operator with the native name `$funPtr`; a `FunCall` translation would recognize this and translate it to a call to a pointer to a function rather than to a specified function name.

Here is what the Sparrow generic class looks like for a function pointer that takes one parameter:

```
class FunctionPtr(resT, T1: Type) {
    using arity = 1;
    fun[native("$funptr")] () (p1: T1): resT;
    fun call(p1: T1): resT = this(p1);
    private var funPtr: @Byte;
}
```

If one constructs an instance of this class for functions that take an `Int` as argument and return an `Int`, and then calls it, the LLVM translation will look like:

```
%1 = bitcast %"FunctionPtr[Int, Int]"* %obj to i32 (i32)**
%2 = load i32 (i32)** %1
%res = call i32 %2(i32 %arg)
```

The first two lines correspond to the translation of the `FunRef` node while the last line corresponds to the special translation of the `FunCall` node.

Bitcast

A `Bitcast(destType, exp)` does a reinterpret-cast over the given expression into the given destination type. Both the destination type and the type of the given expression must have storage (they must not be `Void`) and they must be references.

Such a node would translate directly into a `bitcast` LLVM instruction, as shown below:

```
%intPtr = bitcast i8* %bytePtr to i32*
```

Null

A `Null(destType)` node creates a node to represent a *null* value of the given type. The given type must be a reference type. The LLVM translation will be a `null` constant of the corresponding type:

```
store i8* null, i8** %dest
```

StackAlloc

A `StackAlloc(typeNode, numElements, alignment)` node is similar to a variable, but can also create arrays on the stack. It translates into an `alloca` instruction. Currently it is not used.

Conditional

A `Conditional(condition, alt1, alt2)` node emulates the behavior of the ternary operator in C++ or C, but with mode constraints: no conversions are made. In Sparrow, one would write a conditional expression as a function call: `ife(condition, alt1, alt2)`. The condition must be an `i1` or `u1` type (possibly with references), and the types of the two alternatives must match (ignoring any difference in the evaluation mode). The resulting type will be the type of the alternatives (with the right evaluation mode).

Compared to other expressions, the translation of this node is more complex. Besides using branch instructions (and therefore `phi` instructions to *join* the results from the two possible paths), we need to make sure to translate the temporary and scope destruct actions only from the selected alternative. For translating these destruct actions, we need another branch instruction.

Let us provide an example. The following Sparrow code:

```
fun testCond(cond: Bool) {
    writeLn(ife(cond, genFoo(1), genFoo(2)).x);
}
```

where `ife` is a macro that expands to a conditional node, will be translated into:

```
define private void @testCond(i1 %cond) #5 {
  %cond.addr = alloca i1
  store i1 %cond, i1* %cond.addr
  %"$tmpC" = alloca %Foo
  %"$tmpC1" = alloca %Foo
  br label %code
code:                                      ; preds = %0
  %1 = load i1* %cond.addr
  br i1 %1, label %cond_alt1, label %cond_alt2
cond_alt1:                                 ; preds = %code
  call void @genFoo(%Foo* %"$tmpC", i32 1)
  br label %cond_end
cond_alt2:                                 ; preds = %code
  call void @genFoo(%Foo* %"$tmpC1", i32 2)
  br label %cond_end
cond_end:                          ; preds = %cond_alt2, %cond_alt1
  %cond2 = phi %Foo* [%"$tmpC",%cond_alt1],[%"$tmpC1",%cond_alt2]
  %2 = getelementptr inbounds %Foo* %cond2, i32 0, i32 0
  %3 = load i32* %2
  call void @writeLn(i32 %3)
  br i1 %1, label %cond_destruct_alt1, label %cond_destruct_alt2
cond_destruct_alt1:                        ; preds = %cond_end
  call void @dtor628(%Foo* %"$tmpC")
  br label %cond_destruct_end
cond_destruct_alt2:                        ; preds = %cond_end
  call void @dtor628(%Foo* %"$tmpC1")
  br label %cond_destruct_end
cond_destruct_end: ;preds=%cond_destruct_alt2,%cond_destruct_alt1
  ret void
}
```

5.2.4 Statements

If

An `If(condition, thenClause, elseClause)` represents a conditional instruction. Any clause may be null, but not both of them. As in the case of a conditional expression, the condition must have type that is translated into an `i1` or `u1` (possibly with references). The types of the two clauses do not matter.

To translate this node, we create four LLVM basic blocks: `if_block`, `if_then`, `if_else`, and `if_end`. In the first block we translate the condition expression. Based on its value we jump to the corresponding *then* or *else* block. The `thenClause` and `elseClause` will be translated as instructions in the corresponding block. This means that all the temporary destruct actions will be translated at the end of each block. From these two blocks we create an unconditional jump to the final block. We do not translate anything in this block, but the rest of the translation will be put there.

For the following Sparrow code:

```
fun test(cond: Bool) {
    if ( cond )
        f();
    else
        g();
}
```

the following LLVM code is produced:

```
define private void @test(i1 %cond) #4 {
    %cond.addr = alloca i1
    store i1 %cond, i1* %cond.addr
    br label %code
code:                                   ; preds = %0
    br label %if_block
if_block:                               ; preds = %code
    %1 = load i1* %cond.addr
    br i1 %1, label %if_then, label %if_else
if_then:                                ; preds = %if_block
    call void @f()
    br label %if_end
if_else:                                ; preds = %if_block
    call void @g()
    br label %if_end
if_end:                                 ; preds = %if_else, %if_then
    ret void
}
```

The temporaries for the condition are still available for the whole scope of the **`if`** instruction.

If the `If` node is explicitly set to the compile-time evaluation mode, then the node's behavior is slightly different. First, the condition is evaluated at compile-time (if this cannot be done, an error is issued). Then, the node of the selected branch will be set as the explanation of the `If` node. The other node will not even be compiled. The resulting LLVM code will not contain a branching instruction; only the selected branch will be translated.

For example, if we change the above code into a compile-time **`if`** like this:

```
fun test(cond: Bool) {
    if[ct] ( typeOf(cond) == Int )  // compile-time false
        f();
    else
        g();
}
```

we obtain an LLVM translation like the following:

```
define private void @test(i1 %cond) #4 {
    %cond.addr = alloca i1
    store i1 %cond, i1* %cond.addr
    br label %code
code:                                        ; preds = %0
    call void @g()
    ret void
}
```

This is one of the main ways a programmer can conditionally generate code based on compile-time expressions.

While

A `While(condition, body, step)` node represents a traditional **`while`** structure, but it also allows specifying a step instruction to be executed between loops. In this respect it is similar to the **`for`** instruction from C and C++.

When translating to LLVM we create again four basic blocks: `while_block` for expressing the condition, `while_body` for translating the body of the while, `while_step` where the step action is placed, and `while_end` where the instruction pointer jumps when the **`while`** loop finishes.

When translating the condition, a conditional jump is created to reach the body (`while_body`) or the end of the while (`while_end`). The body block is followed by the step block even if there is no step action; at the end of the step block there is an unconditional jump the the `while_block` block. Both the body and the step are translated like instructions, so the temporary destruct actions are also translated.

The following Sparrow code:

```
fun testWhile(n: Int) {
    var i = 0;
    while ( i<n ; ++i )
        writeLn(i);
}
```

will be translated into:

```
define private void @testWhile(i32 %n) #5 {
    %n.addr = alloca i32
    store i32 %n, i32* %n.addr
    %i = alloca i32
    br label %code
code:                                   ; preds = %0
    store i32 0, i32* %i
    br label %while_block
while_block:                            ; preds = %while_step, %code
    %1 = load i32* %i
    %2 = load i32* %n.addr
    %3 = call i1 @"<631"(i32 %1, i32 %2)
    br i1 %3, label %while_body, label %while_end
while_body:                             ; preds = %while_block
    %4 = load i32* %i
    call void @writeLn(i32 %4)
    br label %while_step
while_step:                             ; preds = %while_body
    %5 = call i32 @"pre_++205"(i32* %i)
    br label %while_block
while_end:                              ; preds = %while_block
    call void @dtor16(i32* %i)
    ret void
}
```

As with the `If` node, the `While` node can be used at compile-time with a special meaning. While the condition (which needs to be a compile-time expression) evaluates to **`true`** the body is *included* in the final code. The step condition must also be evaluable at compile-time.

Here is a code that uses a **`while`** at compile-time to repeat an operation several times:

```
var[ct] i = 0;
while[ct] ( i<5; ++i )
    writeLn(ctEval(i));
```

This code is equivalent to the following code:

```
writeLn(0);
writeLn(1);
writeLn(2);
writeLn(3);
writeLn(4);
```

This is another powerful method for generating code at compile-time.

Break and Continue

A `Break` node does not take any arguments when constructing. It operates on the inner most **`while`** structure and produces an unconditional jump to the `while_end` label.

Similarly the `Continue` node produces a jump to the `while_step` basic block of the inner most **`while`** structure.

Before doing the unconditional jump, these nodes must *unwind* the stack and translate the temporary and scope destruct actions up to the `While` node.

Both of these nodes must reside inside a `While` node for the program to be valid.

Return

A `Return(expr)` node can be created with a valid expression or with a null expression, depending on the function result type. If the function has a result type, the `Return` node needs to be created with an expression of the same type as the function result type. If the function returns `Void` no expression needs to be given to the `Return` node.

If the function has a complex result type for which the function takes a return parameter, then a `store` operation will place the result of evaluating the expression into the result parameter, and a simple `ret` instruction is generated without an expression. If the function has a simple result type (no return parameter), then a `ret` instruction with the appropriate value will be created. If the function has a `Void` result type, a simple `ret` instruction will be generated.

Before yielding the `ret` instruction the translating process performs an *unwind* of the stack, translating all the temporary and scope destruct actions corresponding to the current function.

For the following Sparrow code:

```
fun genFoo(n: Int): Foo {
    var i = n+1;        // scope destruct here
    return Foo(i);
}
```

the following LLVM code will be generated:

```
define private void @genFoo(%Foo* sret %_result, i32 %n) #5 {
    %_result.addr = alloca %Foo*
    store %Foo* %_result, %Foo** %_result.addr
    %n.addr = alloca i32
    store i32 %n, i32* %n.addr
    %i = alloca i32
    br label %code
code:                                        ; preds = %0
    %1 = load i32* %n.addr
    %2 = call i32 @"+206"(i32 %1, i32 1)
    store i32 %2, i32* %i
    %3 = load %Foo** %_result.addr
    %4 = load i32* %i
    call void @ctor629(%Foo* %3, i32 %4)
    call void @dtor16(i32* %i)          ; scope destruct action
    ret void
dumy_block:                             ; No predecessors!
    call void @dtor16(i32* %i)
    ret void
}
```

5.3 Sparrow frontend

The SparrowFrontend nodes are higher-level nodes that handle Sparrow specific constructs by translating them to Feather nodes. Most of the time, they can be thought of as syntactic sugar over the Feather nodes.

There are several abstractions the arise often when implementing the translation. Among these abstractions are: implicit conversions, name lookup, function overloading, and generics.

Implicit conversions

The Feather constructs are designed to be as simple as possible, and therefore no implicit conversions are allowed at that level. All the implicit conversions that Sparrow allows are implemented at SparrowFrontend level.

There are three levels of conversion: *direct*, *implicit*, and *custom*. The direct conversion is preferred to the other conversion types, and the implicit conversion is preferred to the custom conversion.

Here is the list of conversions that Sparrow allows, together with the type of conversion:

- direct: same type—a type can always convert to itself
- direct: change mode—handles the basic evaluation mode changes
- direct: data to concept—allows a data type to be used in place of a concept that it models
- direct: concept to concept—a more specialized concept can be converted to a more general concept (e.g., `BidirRange` can be converted to `Range`)
- direct: l-value to reference
- implicit: null to reference—the **`null`** constant can be converted to any reference type
- implicit: reference to non-reference—we can always dereference a type to obtain a type with one less reference
- implicit: non-reference to reference—types that do not have any references can be converted to a type with one reference; this is done by creating a temporary variable for the reference
- custom: conversion via conversion constructor—the compiler searches for a conversion constructor to be applied to convert between two apparently unrelated types (different classes)

Some of these conversions requires some code to be added. For example, dereferencing implies creating a `MemLoad` node around the source expression, and the custom conversion implies creating a new variable and calling a custom conversion constructor.

Name lookup

The Sparrow compiler uses the standard symbol table technique [Mak09; ALSU06; GJLB00; CT11] to keep track of the names of all the symbols. The symbol table can be viewed as a tree containing nodes with name and without name. The nodes with name are actually pairs of symbol names and the AST nodes that introduce those names. The nodes without name correspond to AST nodes like `LocalScope` that do not introduce new names but introduce new scopes.

A search in the symbol table always starts from a given node, but the direction of the search can be both upward and downward.

An upward search is used when we are searching for a name from the current context towards the root of the symbol table tree; for example, we use this search when resolving a variable or a function to be called. If the name is not found in the current level of the symbol table, the algorithm moves to the next level. If a name is found in a level, the search does not continue to the upper levels.

A downward search is performed when we have a qualified name (e.g., a compound expression) and we must look for the name inside a declaration. In this case we traverse the table from the node corresponding to the qualified node, searching the children names. In this case, only one level in the symbol table is searched.

Function overloading

Sparrow supports function overloading. This means that we can have multiple functions (or function-like entities) with the same name but with different parameters sets, and the caller can distinguish between them by the argument types.

Note that we are using the term *function* here to mean *anything that is callable*. For example, instantiating a generic class is also done using the function overloading mechanism.

In Sparrow, the function overloading algorithm goes as following:

- first, the declarations that match the name that needs to be called are selected; this selection is only based on the name (see name lookup above)
- the declarations that are not callable are discarded; e.g., a variable can be selected in the previous step, but it is not callable

- the candidates are filtered, so that only declarations compatible with the calling arguments are retained; in this step we check the parameter count, whether there are conversions from the actual parameter types to the formal parameter types, and also at this point we attempt to instantiate generics
- from the resulting list of declarations we select the *most specialized* one; if no such thing exists, an error occurs
- if at any step we remain with zero declarations, an error is generated

A function f is more specialized than another function g if all the parameters from f are convertible to the parameters of g, but the converse is not true. For example `f(Int, Char)` is more specialized than `f(Double, Char)`, as `Double` is convertible to `Int`, but `Int` is not convertible to `Double`.

If successful, this process will select only one declaration. This is then used for the function application.

5.3.1 Declarations

Package

A `Package(name, children)` node adds an entry to the symbol table with the given name (pointing to the `Package` node). The `children` node must be of the `NodeList` kind and represents the children of the package. Only top-level nodes are allowed as children of a `Package` node (classes, functions, variables, `Nop`, `BackendCode`, `GlobalConstructAct`, `GlobalDestructAct`).

A `Package` node expands to its children nodes.

SprCompilationUnit

An `SprCompilationUnit(packageName, imports, decls)` node represents a Sparrow compilation unit.

This essentially expands to the given list of declarations. If `packageName` is a valid qualified identifier (a chain of `CompoundExp` and `Identifier` nodes) then we create packages corresponding to all the package names and put our declarations inside. Otherwise, this argument must be null.

The list of imports will tell the compiler which files are needed in order to compile the current file.

Using

A `Using(alias, usingNode)` node allows shortcutting the name lookup table and creating aliases.

If the `alias` argument is not passed, the node copies the declaration(s) from the symbol table corresponding to the `usingNode` into the current symbol table. The `usingNode` must refer to one or more declarations.

If the alias argument is provided, it enters a new entry in the symbol table with the alias name that corresponds to the given usingNode. Note that in this case the usingNode argument does not need to refer to a declaration.

This node expands to a Nop node.

Examples:

```
using Meta.*;           // bring all the declarations from the
                        // Meta packate into our symbol table
using count = 10;       // count is always expanded to 10
using ValType = @Int;   // ValType is expanded to @Int type
```

SprFunction

An SprFunction(name, returnType, parameters, body, ifClause) will represent any possible function definitions in Sparrow, including generics.

If one of the parameters has a compile-time type, then this is a generic and the node will simply expand to a GenericFunction node.

If it is not a generic then this will expand to a simpler Feather Function node with the given body. The parameters of the functions are passed to the basic Function node, with two possible additional parameters.

If this is a member function (declared inside an SprClass node without the static modifier), then an additional **this** parameter is added with the type of a reference to the class in which the function is declared.

If the function returns a complex type (not a basic numeric type and not a reference), then we add a new parameter corresponding to a reference to the return type.

If this is a member function and the function name is ctor or dtor we add some internal modifiers to it so that we provide automatic initialization and destruction for the members of the class.

If the function is global, its name is ctor or dtor, it takes no parameters, and returns nothing, then we create a GlobalConstructAct, or respectively a GlobalDestructAct node that calls this function.

Example:

```
SprFunction("test", TypeNode(Foo),
    NodeList(Var("a", Int)),
    LocalSpace(
        ...
    )
)
```

gets expanded to:

```
Function("test", TypeNode(Void),
    NodeList(Var("result", @Foo), Var("a", Int)),
    LocalSpace(
        ...
    )
)
```

SprParameter

An `SprParameter(name, typeNode, init)` is a variable used in place of a parameter. Typically this expands to a Feather `Var` node. However it can be used to provide a default value for the parameter; the logic for using the default initializer is implemented at the call site.

SprVariable

An `SprVariable(name, typeNode, init)` is a generalization of the basic `Var` node. It can be used for fields in a class, local variables, and global variables (parameters are handled by the `SprParameter` node).

If the type argument is not passed, the variable can deduce the type from the `init` node. When taking the type from the initializer, any references are removed.

If the variable is not a field, then construct and destruct actions will also be created. If the variable is local then the construct action is simply a call to the appropriate constructor, and the destruct action is a `ScopeDestructAct` with a call to the appropriate destructor. If we have a global variable, the constructor call will be placed inside a `GlobalConstructAct` node and the destructor call will be placed inside a `GlobalDestructAct`.

The node is finally explained by a `NodeList` node that contains the Feather `Var` node, the construct action, and the destruct action. To make the type of this node be `Void`, we also add to the resulting `NodeList` a `Nop` node.

For example, the `SprVariable("a", null, CtValue(Int, 10))` node corresponding to a local variable definition `var a = 10` will be explained by the following node:

```
NodeList(
    Var(name="a", type=Int),
    FunCall("Int_ctor", VarRef("a"), CtValue(Int, 10)),
    ScopeDestructAct(
        FunCall("Int_dtor", VarRef("a"))
    ),
    Nop()
)
```

For fields, an `SprVariable` is explained by a simpler `Var` node, as the constructors and destructors of the class will handle the construction and destruction of the field.

SprClass

An `SprClass(name,parameters,baseClasses,ifClause,children)` node handles all the Sparrow classes, from the simple structures to the more complex generic classes. This will add an entry to the symbol table with the given name.

If the `parameters` argument is not null, this will be expanded into a `GenericClass` node, passing the `parameters` and `ifClause` arguments. Otherwise the `ifClause` node must be null.

The children of the class can be any declaration nodes (classes, functions, variables, and using nodes). This node will separate the fields of the class from the rest of the declarations. The fields of the class are used to create a simplified `Class` node corresponding to the Sparrow class. The rest of the declarations are placed outside the generated `Class` node.

For example, if we have a declaration like the following:

```
SprClass("MyClass", null, null, null, NodeList(
    SprVariable("a", Int, null),
    SprVariable("b", Double, null),
    SprFunction("ctor", null, null, LocalSpace(...), null),
    SprFunction("dtor", null, null, LocalSpace(...), null),
    SprFunction("print", null, null, LocalSpace(...), null),
    SprFunction("set", null, NodeList(...), LocalSpace(...), null),
))
```

then, the following nodes will be generated:

```
NodeList(
    Class(name="MyClass", fields=NodeList(
        Var("a", Int),
        Var("b", Double)
    )),
    Function("ctor", TypeNode(Void),
        NodeList(Var("$this", @MyClass)),
        LocalSpace(...)
    ),
    Function("ctor", TypeNode(Void),
        NodeList(Var("$this", @MyClass)),
        LocalSpace(...)
    ),
    Function("print", TypeNode(Void),
        NodeList(Var("$this", @MyClass)),
        LocalSpace(...)
    ),
    Function("set", TypeNode(Void),
        NodeList(Var("$this", @MyClass), ...),
        LocalSpace(...)
    )
)
```

If the class has parameters, the node will be explained by a `GenericClass` node.

GenericClass

A `GenericClass(origClass, parameters, ifClause)` node represents a generic class. All the parameters must be compile-time and not concept types.

Like C++ templates [VJ02], generics in Sparrow are ad-hoc, making each generic *instance* behave differently for each given set of parameters. Therefore, using a generic is a two-step process:

- *instantiation*. From the generic class, a new *instance* class will be generated, without any parameters.
- compilation / usage of the generated class. This follows the regular simplification process for compiling the class.

The instantiation happens whenever a class is provided with some parameters. To instantiate a class, we need to *bind* all the class parameters by providing some values for them, so that the body of the generic can use them. To bind these parameters we create compile-time variables for each parameter of an instance.

As a generic can be instantiated with different sets of arguments, we can have multiple instances of the same generic. However, the compiler will not create multiple instances if the same set of arguments is passed from different places; the same instance of the generic is used.

Let us take an example. Suppose that we want to create a generic `Pair` class. In Sparrow, the code would look like:

```
class Pair(T, U: Type) {
    var first: T;
    var second: U;
}
```

This generic can be used in the following way:

```
var p1: Pair(Int, Double);      // first instantiation
var p2: Pair(Char, Bool);       // second instantiation
var p3: Pair(Int, Double);      // first instantiation reused
assert( typeOf(p1.first) == Int );
assert( typeOf(p1.second) == Double );
```

For the `Pair` class, an `SprClass` node will be generated, which will be explained in terms of a `GenericClass` node; for the exemplified instantiations, the `GenericClass` node will be explained by an AST structure similar to:

```
NodeList(
    NodeList(       // first instantiation
        SprVariable("T", Type, Int),
        SprVariable("U", Type, Double),
        SprClass("Pair", null, null, null, NodeList(
            SprVariable("first", VarRef("T"), null),
            SprVariable("second", VarRef("U"), null)
        ))
    ),
    NodeList(       // second instantiation
        SprVariable("T", Type, Char),
        SprVariable("U", Type, Bool),
        SprClass("Pair", null, null, null, NodeList(
            SprVariable("first", VarRef("T"), null),
            SprVariable("second", VarRef("U"), null)
        ))
    )
)
```

As can be seen, a new class will be generated for each instance of the generic class. Because the parameters are added as variables above, the body of the class will use the generic parameters without any change in their structure.

Each generic class can have an *if clause* attached to it. This comprises a compile-time expression that needs to evaluate to Bool. If the expression cannot be evaluated at compile-time, and if it does not evaluate to **`true`**, the generic will not be instantiated. This expression is evaluated in the scope of an instance, where the bound variables are available, but before the class node is copied into the resulting instance `NodeList`.

For example, suppose we want to instantiate with `Int` and `Char` the following generic class:

```
class BigNum(T: Type) if sizeOf(T) >= 4 {
    var num: T;
}
```

The generic instantiations will be:

```
NodeList(
    NodeList(       // first instantiation
        SprVariable("T", Type, Int),
        // "sizeOf(T) >= 4" evaluates to true
        SprClass("BigNum", null, null, null, NodeList(
            SprVariable("num", VarRef("T"), null)
        ))
    ),
    NodeList(       // second instantiation
        SprVariable("T", Type, Char)
        // "sizeOf(T) >= 4" evaluates to false
        // the instance class is not generated
    )
)
```

Whenever a generic cannot be instantiated (in a function application), the compiler behaves as if the class does not exist, without producing an actual error. The compiler will try to use other overloads if possible; if that is not possible, an error is issued, informing the user that no viable overload is found.

GenericFunction

A `GenericFunction(origFun, parameters, ifClause)` node is to an `SprFunction` what `GenericClass` is to `SprClass`.

It has the same instantiation mechanism as an `GenericClass`, and the **`if`** clauses behave similarly. Apart from the fact that it operates on functions instead of classes, the only difference is the way parameters are treated.

If in an `SprClass` all the parameters must be generic parameters, in an `SprFunction` one can have both generic parameters and regular parameters. There are two important aspects to be discussed: when a function becomes a generic, and how to distinguish between generic parameters and regular parameters.

A function declared with the `ct` modifier is not a generic function. It can be transformed into a compile-time generic using the `ctGeneric` modifier—in this case all the parameters are considered generic parameters. A non-compile-time function (run-time, or both run-time and compile-time) is a generic if it has at least one compile-time parameter or at least one parameter which is a concept (e.g., `AnyType`, `Range`, `Number`).

Let us assume that we have a function that contains regular run-time parameters, concept parameters and compile-time parameters. The compile-time parameters are always treated as generic parameters; the regular run-time parameters are treated as regular function parameters, and will not take part in the instantiation process. The generic parameters belong to both categories. The type corresponding to a concept that is used when instantiating the generic will be considered the compile-time part that acts as a generic parameter; the generated function will contain a parameter with the instantiated type, so that the value can be passed to the instantiated function.

For example, the following function:

```
fun f(a: Double, b: Int ct, c: Number) {...}
```

if instantiated as `f(3.14, 10, Short(24))`, the following instantiation is produced:

```
NodeList(
    NodeList(
        SprVariable("b", Int, 10),
        SprVariable("c", Type, Short),
        SprFunction("f", null, NodeList(
            SprVariable("a", Double, null),
            SprVariable("c", VarRef("c"), null)
        ), ...)
    )
)
```

Note that the `c` parameter is both a generic parameter (has a value at compile-time denoting the type to be used) and a run-time parameter. The compiler allows both parameters to have the same name.

The **`if`** clauses work in the same way as they do in the case of classes. The types of the concept parameters can appear in **`if`** clauses.

SprConcept

An `SprConcept(name, paramName, baseConcept, ifClause)` represents a Sparrow concept declaration.

An example of a concept declaration in Sparrow is:

```
concept BidirRange(x: Range)
    if typeOf(x.back()) == x.RetType
    && isValid(x.popBack())
    ;
```

In this example, the name of the concept is `BidirRange`, the parameter name is `x`, the base concept is `Range`, and the **`if`** clause is the expression formed by the conjunction of the `==` and `isValid` predicates.

A concept is essentially a predicate on types. A type can either *model* a concept or not. The *type models concept* relationship should not change over time. If one finds that a type models a concept, the type will always model that concept, no matter how the global state of the compiler changes.

With this in mind, one of the easiest ways of implementing a concept is using generics—and this is the Sparrow compiler approach. For each concept we create a dummy generic parameterized over a type variable. If a base concept is given, the type must model the given base concept. The **`if`** clause of the concept will become the **`if`** clause of the generic. If we can instantiate the concept generic for a given type it means that the type models the concept.

For concepts, there is no instantiated entity. We care only about the instantiation test: is the type modeling the requirements of the concept or not?

5.3.2 Expressions

Literal

A `Literal(typeName, data)` node represents a literal as it appears in the Sparrow grammar. The type is a string constant and the data is also encoded as a string. The `Literal` node will check that the type exists, that the size of the given data matches the properties of the type, and will explain itself as a `CtValue` node.

Identifier

An `Identifier(name)` is constructed from a string and refers to the declaration with that name. The node will first look up any declarations with that name in the symbol table. If no declarations are found, an error is issued.

Depending on the kind of the found declaration the node can be expanded to a `VarRef`, `FieldRef`, or a `TypeNode` (if the declaration is explained to a `Class` or an `SprConcept`). If the declaration refers to a `Using` node (in the case in which it was constructed with alias) this will be explained in the expression given to the `Using` node.

If we cannot find a node to represent the reference to the declarations found by name lookup, we create a `Nop` node; we attach the list of the found declarations as a property to this node. The overloading algorithm and the compound expressions are able to read this node property. Note that an identifier can refer to multiple declarations; a good example is the name of an overloaded function.

CompoundExp

A `CompoundExp(baseExp, idName)` node corresponds to a Sparrow dot expression: `baseExp.id`.

This node is similar to an `Identifier` node, but the name lookup is performed in the scope of the base expression (downward search). First we need to get the declarations referred to by the base expression; if none is found, an error is issued. Then we search in the symbol table corresponding to each declaration for the given id. At least one result needs to be generated in order for the node to be valid. The expansion of this node is from this point similar to the one used for `Identifier` nodes.

StarExp

A `StarExp(baseExp)` is similar to a `CompoundExp` node, but without a given identifier. If the base expression refers to some declarations, then this node will refer to all the declarations found inside these declarations. In Sparrow this is represented by a `.*` suffix and it is especially useful for **`using`** directives:

```
using Meta.*;
```

This

A `This` node is created each time the **`this`** keyword is used. This just expands to an identifier with the name `$this`. Sparrow methods will use this name for the special parameter added to encode the current object.

FunApplication

A `FunApplication(base, arguments)` is one of the most used nodes in Sparrow. Because it can be used in multiple ways, it is also one of the most complex nodes.

The node analyzes the `base` node and gets the declarations referred by it. These represent the entities that can be called. There are several types of such entities that can be called: regular functions, constructor calls (class name followed by arguments), generic calls (generic functions or generic classes), and concept calls. Each of these entity types does the calling in a different manner.

A regular function call is explained by a regular `FunCall` node.

A class constructor call will expand into something similar to:

```
NodeList(
    Var(name="tmp.v", type=TypeNode(<cls_type>)),
    FunCall("ctor", VarRef("tmp.v"), <args>),
    TempDestructAct(
        FunCall("dtor", VarRef("a"))
    ),
    VarRef("tmp.v")
)
```

It creates a temporary variable of the given class, calls the constructor, makes sure the variable is destructed when the instruction ends, and returns a reference to the variable.

A function application to a generic class (e.g., `Vector(Int)`) will create a type. Therefore it will be explained by a `TypeNode` node.

A function application to a generic function, will result in a call to a function that has the generic parameters bound. The arguments of the resulting function call will be the ones corresponding to the non-compile-time parameters of the generic.

Before actually selecting what type of entity needs to be called, the overloading process is invoked. Having multiple declarations the function application can use, this algorithm selects the most specialized entity to be called. It also makes sure that the right conversions are applied to the given arguments in order to match the parameters of the entity to be called.

The `FunApplication` node also has extensive logic for dealing with the evaluation mode. It computes what mode the called entity should have, the mode of the arguments passed to that entity, and the evaluation mode of the result.

In addition, this node checks for some special compiler intrinsics, handling them accordingly. Examples of such intrinsics are: `ctEval`, `sizeOf`, `reinterpretCast`, `isValid`, and `lift`.

OperatorCall

An `OperatorCall(arg1, op, arg2)` node handles the specifics of prefix, postfix, and infix operator calls. Postfix operators have `arg1` non-null and `arg2` null; a prefix operator will have `arg1` null and `arg2` non-null. An infix operator will have both arguments non-null.

With some exceptions, this node is explained in terms of a `FunApplication` node. Still, the rules for searching for the name of the function to be called are not trivial. A function with the name of the operator is searched for in the class corresponding to the first valid argument; if no such function is found, the search will be performed near this class; otherwise the search will start from the current node upwards.

For unary operators we perform two name searches in all three cases. First we try searching using the `pre_` and `post_` prefixes and only if nothing is found do we search with the initial operator name. This allows us to distinguish between prefix-only and postfix-only operators.

If our algorithm does not produce a result, there are several fall-back cases:

- `a != b` gets transformed into `!(a == b)`
- `a > b` gets transformed into `b < a`
- `a <= b` gets transformed into `!(b < a)`
- `a >= b` gets transformed into `!(a < b)`
- `a >= b` gets transformed into `!(a < b)`
- `a <op>= b` gets transformed into `a = a <op> b`

There are several cases which are treated differently from this algorithm. These includes the handling of reference equality, reference inequality, reference assignment, the dot operator, and a generic function application operator. The dot operator gets translated into a `CompoundExp` expression, and the function application operator gets translated directly into a `FunApplication` node. We use them as operators so that we can assign them precedence orders[7].

InfixExp

An `InfixExp(arg1, op, arg2)` represents an infix operator. It is always explained in terms of an `OperatorCall`, but handles the precedence and associativity of the operators. If the two arguments of an infix expression are also `InfixExp` nodes, then this node can swap the arguments around.

For example, let us consider the expression `a op1 b op2 c`, represented by the `InfixExp(a, op1, InfixExp(b, op2, c))` nodes. If the precedence of `op1` is smaller than the precedence of `op2`, then the expression is

[7] the `f(a)` is transformed into `f __fapp__ (a)` which is later explained by a `FunCall(f, NodeList(VarRef("a")))` node; an expression like `x.y->z(a)` is transformed into `x __dot__ y -> z __fapp__ (a)`—here `__dot__` and `->` have the same precedence, which is lower than `__fapp__`

not changed; if the precedence is greater, then the code is transformed into `InfixExp(InfixExp(a, op1, b), op2, c)`. If `op1` is the same as `op2` and they have right associativity, the two nodes are also swapped.

SprConditional

An `SprConditional(cond, alt1, alt2)` is similar to the `Conditional` Feather node, except that it provides automatic conversions, so that the two expressions can be of two different types.

If the first expression has type T_1 and the second expression has type T_2, then the `SprConditional(cond, alt1, alt2)` expression is explained by `Conditional(cond, conv(alt1, R), conv(alt2, R)`, where `R` is the *common type* between T_1 and T_2. This common type will be the most specialized type to which both T_1 and T_2 can convert; except for numerical type, this type will be either T_1 or T_2. If there is no common type, the compiler yields an error. Here, the notation `conv(exp, R)` represents the conversion of the given expression to the given type `R`.

LambdaFunction

A `LambdaFunction(params, retType, body, closureParams)` node represents an anonymous function. It offers an easy way of defining functions inside expression.

For example the following Sparrow code:

```
var a = 10;
var f = (fun.{a} (x: Int): Int = x+a);
```

will create the following lambda function nodes:

```
LambdaFunction(NodeList(Var("x", Int)), Int,
    InfixExp(VarRef("x"), "+", VarRef("a")),
    NodeList(VarRef("a")
)
```

The explanation of this node is performed in two parts: one at top-level and one at the level in which the lambda function appears. At the top-level, we introduce the following definition:

```
SprClass("$lambdaEnclosure", null, null, null, NodeList(
    SprFunction("()", params, retType, body)
    SprFunction("ctor", null, null, LocalSpace(), private)
    SprFunction("ctor", <closureParams_vars>, null, <init_body>)
    <vars_for_closureParams>
))
```

The default constructor is made private. For each closure parameter, we create a corresponding variable in the class and we add to the last constructor a parameter and an initialization instruction for the class variable. This way, the generated class can store the values of all the closure parameters and the body of the function (represented by the call operator) will be able to use them.

At the level where the lambda function is used, the following code is injected:

```
FunApplication(TypeNode("$lambdaEnclosure"), closureParams)
```

This will create a call to the constructor of the class that we created, passing the list of closure parameters as arguments to the constructor.

5.3.3 Statements

SprReturn

An `SprReturn(expr)` node is an extension of the `Return` node, adding the possibility of implicit conversions between the passed expression and the function result type. This node is transformed into `Return(conv(exp, RetType)`, where `RetType` is the function return type. As with the basic `Return` node, if the enclosing function has the `Void` return type, no expression should be given to the `SprReturn`; similarly, if the function has a non-`Void` return type, an expression convertible to `RetType` must be given to the `SprReturn` node.

For

A `For(name, typeExpr, range, action)` node is a generalization of `While` loops that works on Sparrow ranges. This node translates into:

```
SprVariable("$rangeVar", Identifier("Range"), range)
While(
    OperatorCall(null, "!",
        FunApplication(
            CompoundExp(VarRef("$rangeVar"), "isEmpty"),
            NodeList()
        )
    ),
    LocalSpace(
        SprVariable(name, typeExpr,
            FunApplication(
                CompoundExp(VarRef("$rangeVar"), "front"),
                NodeList())
            ),
        action
    ),
    FunApplication(
        CompoundExp(VarRef("$rangeVar"), "popFront"),
        NodeList())
    )
)
```

If the `typeExpr` node is not given to the `For` node, the compiler will use the type of the `CompoundExp(VarRef("$rangeVar"), "RetType")` expression instead; each range needs to have this type defined, as it represents the type of the values returned by the range.

5.3.4 The Concept type

The Sparrow frontend component adds a new type: `ConceptType`. This is constructed from a reference to a concept node and the number of references applied to it. It represents a value of a type that is modeling the given concept, with the exact number of references.

Examples of concepts are: `AnyType`, `Range`, and `Number`.

5.3.5 Modifiers

A `ModifiersNode(base, mods)` is constructed with a base node and one or more modifiers to be applied to that node. If `mods` is a node of the `NodeList` kind then the direct children of this node represent the modifiers to be applied to the base node.

All the modifiers are expressed as `Identifier` nodes, except the `native` modifier which is expressed as a function call.

Here is the list of modifiers that Sparrow currently supports:

- `ct`. Indicates that the declaration should be put into the compile-time evaluation context
- `rt`. Indicates that the declaration should be put into the run-time evaluation context
- `rtct`. Indicates that the declaration should be put into both compile-time and run-time evaluation contexts
- `autoCt`. Similar to `rtct`, but the function is evaluated at compile-time if all the given arguments are compile-time; applies to functions only
- `static`. Makes a member of a class (function or variable) not bound to the objects of the class; similar to the `static` keyword in C++ or Java
- `noInline`. Forces the compiler not to inline a function; used when inspecting the generated code if optimizations are turned on
- `native(name)`. When applied to functions it forces the function to be translated with the given name; it can also be used to implement intrinsics (both for classes and functions)
- `noDefault`. Specifies that no default members should be generated for a class (constructors, destructor, assignment operators)
- `initCtor`. Indicates that the compiler must create a constructor to initialize all the fields of the class

- `convert`. Applied to a constructor; indicates that the constructor can be implicitly called to implement automatic conversions
- `ctGeneric`. Marks a function as generic for compile-time only; over standard compile-time functions, it has the advantage of not compiling the body of the function until the generic is instantiated
- `macro`. Marks the function as a macro; instead of passing values to the function, the compiler passes the actual AST nodes of the calling arguments

In addition to these modifiers that can be present in code, the compiler also uses several internal modifiers to *alter* the behavior of some declarations. Here is the current list of internal modifiers that the Sparrow compiler uses:

- `IntModClassMembers`. This is added to classes that do not have the `noDefault` modifier. The compiler will attempt to automatically generate the following members: default constructor, uninitialized constructor, copy constructor, compile-time to run-time constructor, destructor, assignment operator, and equality test operator.
- `IntModCtorMembers`. Applied to constructors; generate a constructor call (or reference assignment) for all the class members that do not have constructor calls in the main body of the constructor
- `IntModDtorMembers`. Similar to `IntModCtorMembers`, this generates destructor calls for all the class members

5.4 Summary

We presented in this chapter the translation rules for a Sparrow program into LLVM. Implementing these rules, one can construct a Sparrow compiler. We chosen to adopt a semi-formal presentation style, to improve readability of the chapter; however a reader with an experience in writing compilers should not have any problem following the rules of the Sparrow language.

By using the translation rules given here, one can theoretically *find* the LLVM translation that the compiler would generate for any Sparrow program, without actually invoking the program.

The astute reader might have noticed that no translation rule incurs any performance overhead. We are not creating not-needed heap allocations, or extra indirections that can slow down the execution time. Theoretically, using these translation rules the Sparrow compiler will generate code as efficient as a C++ compiler would. Chapter 8 will prove this assumption by running some benchmarks.

PART II

Case studies

Blessed are the flexible, for they shall not be bent out of shape.

M. Chura

Chapter 6

Flexible operators

Operators are a fundamental feature in programming languages, allowing computations to be expressed in a natural and convenient manner. Custom operators are supported by many programming languages, however their underlying mechanisms typically suffer from various limitations. This chapter describes a method of defining and customizing operators at the library level in the Sparrow programming language. We argue that our approach is more flexible than what other languages have to offer, and that it allows operators to be defined and used in a more straightforward and natural way.

6.1 Operators in programming languages

Operators originated in mathematical notations, where they are used to conveniently represent various laws of composition, functions, or relations. Unsurprisingly, these concepts are just as necessary in programming languages. Most languages provide the typical set of arithmetic operators, along with relational, logical, and bitwise operators.

As the range of applications for computer programs grew, so did the need for other kinds of operators, many of which applied to non-mathematical objects. Examples include string concatenation using the + operator, and iteration using ++ and −−. Eventually, it became more convenient for some languages to offer programmers the ability to define their own custom operators for user-defined types, instead of hardcoding a fixed set of operators into the language itself. Consequently, from a semantic standpoint, operators became more similar to functions.

An operator has various properties, which are especially relevant in a programming language context. From a syntactic standpoint, an operator can be prefix, infix, or postfix. Arity refers to the number of operands on which an operator is applied. Precedence and associativity determine the order of evaluation in compound expressions.

Although not all programming languages support operators in the traditional sense, the vast majority of programming languages have support for unary and binary operators. However, support for operator customization varies greatly across the spectrum of languages.

In Lisp there is no distinction between functions and operators; they are all written as the first token in a list expression. Haskell is more flexible, allowing the use of arbitrary symbol tokens as operators, infix and postfix (there is only one prefix operator: `-`), and the use of prefix functions with infix notation. In addition, the user can customize the precedence and the associativity of these operators.

For our purposes, we are interested in languages that use C-like syntax for expressions. Languages like C and Java have a fixed set of operators, their precedence and associativity rules are defined by the language, and they do not allow the user to overload these operators; they are the least flexible from this point of view. In C++, even though the operators and the precedence rules are fixed, the user can overload operators for custom types. Scala is even more flexible, allowing the user to create operators from arbitrary symbol tokens; identifiers can be used as infix and postfix operators, however the set of prefix operators is fixed (`-`, `+`, `!` and `~`). On the downside, Scala's rules for computing the precedence and associativity are fixed; they are based on the first and last characters of the token, except for assignment operators (`=`, `+=`, `/=`, etc.) which are handled differently.

Defining custom operators in C++ is done via a special operator construct, similar to a function declaration. To distinguish between prefix and postfix operator calls, C++ requires a dummy integer parameter to be added when defining a postfix operator. Overloading infix operators in C++ can be performed both in the namespace of the operand type (recommended), or inside the class of the left operand. In Scala an infix expression `A op B` is syntactic sugar for `A.op(B)`, and the symbol operators are just regular identifiers. Similarly a postfix expression `A op` is equivalent to `A.op()`. On the other hand, a prefix expression `op A` (where op can only be `-`, `+`, `!` and `~`) is treated as `A.unary_op()`.

In this chapter we propose a flexible and natural mechanism for user-defined operators in our language, Sparrow. Our approach is library-based, and supports binary infix, as well as unary prefix and postfix operators. Furthermore, precedence and associativity are fully customizable for each operator, also at the library level. We argue that our solution is more versatile than those used in other languages, and can be employed with minimal effort from the programmer's part.

6.2 Flexible operators in Sparrow

6.2.1 Design

Operators in Sparrow are designed to meet several requirements:

- operators should be defined in the library, not by fixed grammar rules
- the user must be able to create custom operators (infix, prefix, and postfix)
- the names of operators can be (almost) any combination of symbols
- the user must be able to use identifiers as operators (infix, prefix, and postfix)
- definitions of custom operators are identical to definitions of regular functions
- the user must be allowed to set the precedence of the operators and change their associativity rules

These requirements ensure that programmers will have a large range of possibilities at their disposal when defining operators. Additionally, the operators will be easy to create and use.

6.2.2 Grammar rules

The implementation for Sparrow operators begins with a set of lexical rules [ALSU06]:

```
Letter  ::= 'a'-'z' | 'A'-'Z' | '_'
Digit   ::= '0'-'9'
OpChar  ::= '~' | '!' | '@' | '#' | '$' | '%' | '^' | '&' | '-'
            | '+' | '=' | '|' | '\' | ':' | '<' | '>' | '?' | '/'
            | '*'
Oper    ::= OpChar {OpChar}[1]
            | (OpChar | '.') '.' {OpChar | '.'}
Id      ::= Letter {Letter | Digit} ['_' Oper]
```

Identifiers from languages like C++, containing only letters and digits, will be accommodated easily by this scheme.

In addition, the user can construct identifiers by appending symbols after a regular identifier ending in _. This allows the construction of names such as `pre_++` and `post_++`.

Almost any combination of symbols that are typically available on an ASCII US keyboard can form a lexical token corresponding to an operator, with minor exceptions which are presented below. In theory, the list of characters accepted

[1]in our implementation, the `Oper` lexical rule will not match =; this is needed to distinguish between expressions that contains the equal sign, and the ones that do not

as parts of operators can be extended as much as needed (e.g., with Unicode symbols), but for simplicity we listed only the ones that are commonly used in programming languages.

The following symbols cannot be used as part of an operator: `{`, `}`, `[`, `]`, `(`, `)`, `;` (semicolon), `,` (comma), `` ` `` (backquote) because they act as special separators in the Sparrow syntax. Also, a closer look at the way the operators are defined reveals that we cannot use the dot symbol as an operator if there is no other dot in the sequence; this ensures that a single dot can always be used as a compound expression separator.

The following are valid examples of operators: `+`, `-`, `++`, `**`, `#$`, `=/=/=/=`. Valid examples of identifiers include `foo`, `bar123`, `_123`, `oper_$#@`.

Based on the above lexical rules, we defined the following parsing rules [ALSU06] for operators:

```
IdOrOperator  ::= Id | Operator
PostfixExpr   ::= InfixExpr [IdOrOperator]
InfixExpr     ::= PrefixExpr
                | InfixExpr IdOrOperator InfixExpr
PrefixExpr    ::= SimpleExpr
                | Operator PrefixExpr
                | '`' Id '`' PrefixExpr
SimpleExpr    ::= Id
                | ...
```

The full grammar listing (both lexical and parsing rules) can be found in appendix A.

These simple rules provide a wide range of usage possibilities, and are similar to the ones in Scala. The main difference is the fact that we allow any operator or backquoted identifier to be part of a prefix expression.

Here are some examples of expressions involving operators: `a + b` (add `a` and `b`), `a + b * c` (add `a` to the product of `b` and `c`), `++a++` (prefix `++` on `a`, then apply the postfix `++`), `v1 dot v2` (dot product between `v1` and `v2`), `` `length` a `` (calling length on variable `a` with prefix call notation), `1..100` (the numeric range between 1 and 100, exclusive), `@Int` (type representing a reference to `Int`).

Ignoring prefix operators, if we have a sequence of identifiers and operators, then the terms in odd positions are always operands, and the terms in even positions are always operators. If the sequence has an even number of terms, then the last term is a postfix operator. Note that a SimpleExpr cannot contain operators; therefore we cannot find operators in odd positions (if there are no prefix expressions in the sequence).

Prefix operators always have precedence over other types of operators. If the compiler finds an operator in an odd position in the sequence, it means that it has encountered a prefix expression. This way, there is no ambiguity in distinguishing between prefix, infix, and postfix expressions.

Although these simple grammars rules cover most of the required Sparrow functionality, there are two main cases that are not handled.

The first case concerns the ternary operator construct present in languages like C++: `condition ? alt1 : alt2`. Not only does this not fit grammatically in our scheme, but there are also some additional semantic issues associated with it. While any prefix, postfix, and infix operator can be implemented as a function call (macro or regular function), there is no way to implement the ternary operator in this manner. This is mainly due to the fact that one alternative is not evaluated at all, whereas for a regular function all the arguments need to be evaluated before the actual function call. Normally, the boolean short-circuit operators (`&&` and `||`) would share the same problem, but they can be implemented in terms of the ternary operator. In Sparrow, this is implemented as a macro function (see chapter 11) with three parameters: `ife(condition, alt1, alt2)`.

The second case is related to the handling of the = operator. The way the Sparrow grammar is constructed, there are situations in which we want to prohibit the use of the = operator inside expressions. For example, let us look at a variable definition:

```
var name: typeExpr = initializer;
```

Here, `typeExpr` is actually an expression that can contain prefix, infix, and even postfix operators. For instance, denoting a reference type requires a simple prefix operator call: `@Int`; creating a map from `Int` to `Double` can also be written in operator form as `Int Map Double`. If the = operator were allowed inside the type expression, then variable declarations would be ambiguous. The compiler would not know whether you are using the infix = operator or separating the type expression from the initializer of the variable. To resolve this, the relevant grammar rules have two versions: one that allows the = operator, and one that does not. The latter is used for variable declarations. See appendix A for the full Sparrow grammar. If the symbol = had not been used by Sparrow as a syntactic separator, this limitation would not have existed.

6.2.3 Defining operators

In Sparrow, defining an operator is identical to defining a regular function:

```
class Complex {...}
fun + (x, y: Complex) = Complex(x.re+y.re, x.im+y.im);
fun - (x: Complex) = Complex(-x.re, -x.im);
```

Here we defined a binary operator and a unary operator. These definitions showcase a convenient method of defining a function that simply returns an expression.

With these definitions, one can actually use the two operators as infix, prefix, and postfix:

```
var a, b: Complex;
cout << a+b << endl;
cout << -a  << endl;
cout << a-  << endl;
```

As the reader may have noticed, the definition of the unary minus operator makes it possible for it to be used both as a prefix and a postfix operator. Sometimes this is not desirable; for example, in our case the `a-` notation is confusing. Another important example is the `++` operator, where the postfix and prefix versions have different semantics. To distinguish between the two types of operator calls, one can use the `pre_` and `post_` prefixes when defining the operators, as shown below:

```
fun pre_++ (x: Complex):@Complex { x+=1; return x; }
fun post_++(x: Complex):Complex { var old=x; x+=1; return old; }
```

6.2.4 Operator lookup

Although operators behave like functions, there are some differences in terms of the way they are looked up. For a function call `f(a1, a2, ...)` a name lookup is initiated for the name `f` from the context of the function call, going upward until one or more definitions with the name `f` are found. Afterwards, a function overload selection algorithm is applied to determine which definition should be called with the current arguments.

For operators, the algorithm is different. There are three contexts in which the operator is looked up:

- inside the class of the first operand (left operand for infix expressions, and the only operand of the prefix and postfix expressions)
- in the package that contains the class of the first operand
- starting from the context of the operator call and going upward

For unary operators, the compiler can search for two operator names:

- with a name prefix (`pre_` for prefix operators, and `post_` for postfix operator)
- without any prefix, just the operator name

Putting it all together, the compiler performs the following searches in order:

- in the class of the first operand, with a name prefix (for unary operators)
- in the class of the first operand, with the actual operator name
- in the package that contains the class of the first operand, with a name prefix (for unary operators)
- in the package that contains the class of the first operand, with the actual operator name

- upward from the context of the operator call, with a name prefix (for unary operators)
- upward from the context of the operator call, with the actual operator name

If a matching definition is found in any of these steps, the algorithm will not search any further.

Typically, operators for a new type are written inside or near the class. By choosing the definition that is closest to the class of the first operand, the algorithm maximizes the odds of finding the most suitable operator for the given argument type. This is similar to Scala [Ode11], where a + b actually means a.+(b), and to some extent to Argument-Dependent Lookup (also called Koenig lookup) form C++ [Cpp11; Koe12].

6.2.5 Precedence and associativity rules

For infix operators, we also need to consider precedence and associativity. Precedence determines the order in which different infix operators inside the same expression are called. Associativity determines whether for an expression containing only operators of the same type the order of applying the operator is from left to right, or from right to left.

For each infix operator we can associate a numeric value such that we can compare the precedence of two operators. Let us denote by p_1 the precedence of the operator `op1` and by p_2 the precedence of the operator `op2`. Then, the expression `A op1 B op2 C` would be interpreted as `(A op1 B) op2 C` if $p_1 \geq p_2$, and as `A op1 (B op2 C)` if $p_1 < p_2$. For example, multiplication and division have higher precedence than addition and subtraction.

For an infix operator `op`, an expression like `A op B op C` would be interpreted as `(A op B) op C` if `op` has left associativity, and `A op (B op C)` if `op` has right associativity. Most of the mathematical operators have left associativity, but an operation like assignment makes sense to have right associativity. Also, if one were to define an exponentiation operator, it should also have right associativity.

Because Sparrow operators are defined in the library, there should be a possibility to define precedence and associativity in the library. There should be some kind of definition that the user can write, and the compiler can search for, when determining the precedence and associativity for infix operators. The easiest way this can be achieved in Sparrow is with **`using`** directives:

```
using oper_precedence_default = 100;
using oper_precedence_+       = 500;
using oper_precedence_*       = 550;
using oper_assoc_=            = -1;
```

Such a directive associates a name with a value; it is similar to defining a constant. Whenever the `+` operator is used in the language, the compiler searches for the name `oper_precedence_+` and, if successful, evaluates the expression to get the precedence value for the operator. If the name cannot be found, the compiler will use the precedence value denoted by `oper_precedence_default`. For associativity, if the returned value is negative, then the operator is considered to have right associativity.

Although these using declarations are convenient to write, there is a better method of getting and setting the precedence rules, by using function calls:

```
setOperPrecedence("*", 550);
setOperPrecedence("**", getOperPrecedence("*") + 1);
setOperRightAssociativity("**");
```

By using these functions, the user is not required to know the underlying details of operator precedence and associativity. Moreover, the `getOperPrecedence` function will check whether a value is defined for the given operator, and if not, the default value will be returned.

Sparrow is powerful enough that it allows these functions to be defined at library level, without changing the compiler. The implementation for these functions involves macros and the Compiler API, which allow the programmer to manipulate the internal structure of the program at the Abstract Syntax Tree (AST) level [ALSU06]. The definitions of these three functions that allow the manipulation of the precedence and associativity are presented in listing 6.1. For more details on macros and on AST manipulation, the reader should consult chapter 11.

6.3 Evaluation

We presented the method through which one can construct flexible operator schemes in Sparrow. All Sparrow operators are defined in this manner, and their precedence and associativity values are set using the mechanism described above. In this section we evaluate the benefits and limitations of this approach.

Let us start with an example of an operator for raising a number to a given power:

```
fun pow(x, y: Double) = Math.pow(x, y);
fun **(x, y: Double)  = pow(x, y);
setOperPrecedence("**", getOperPrecedence("*") + 1);
                        // higher precedence than multiplication
setOperRightAssociativity("**"); // right associativity
cout << 4 * 3 ** 2 << endl;      // 36 == 4 * (3**2)
cout << 4 ** 3 ** 2 << endl;     // 262144 == 4 ** (3**2)
```

As illustrated, it is easy to define the operator and to set its precedence and associativity. Furthermore, the newly created operator can be integrated seamlessly with standard arithmetic operators.

```
using Meta.*;
using Meta.Feather.*;
using Meta.SprFrontend.*;
fun[ct,macro] setOperPrecedence(oper, value: AstNode): AstNode {
    var loc = oper.location();
    var loc2 = value.location();
    var operName = astEval(oper);
    var precValue: Int = astEval(value);
    return mkSprUsing(loc, "oper_precedence_" + operName,
        mkIntLiteral(loc2, precValue));
}
fun[ct,macro] getOperPrecedence(oper: AstNode): AstNode {
    var loc = oper.location();
    var operName = astEval(oper);
    var args = mkNodeList(loc);
    addToNodeList(args,
        mkIdentifier(loc, "oper_precedence_" + operName));
    addToNodeList(args,
        mkIdentifier(loc, "oper_precedence_default"));
    return mkFunApplication(loc,
        mkIdentifier(loc, "valueIfValid"), args);
}
fun[ct,macro] setOperRightAssociativity(oper: AstNode): AstNode {
    var loc = oper.location();
    var operName = astEval(oper);
    return mkSprUsing(loc, "oper_assoc_" + operName,
        mkIntLiteral(loc, -1));
}
```

Listing 6.1: The macro functions used to manipulate precedence and associativity for operators, using Sparrow's Compiler API functionality

Another example involves the use of map and filter operations, heavily used in the context of functional programming:

```
fun filter(range: Range, pred: AnyType) = mkRangeF(range, pred);
fun map(range: Range, f: AnyType) = mkRangeM(range, f);
fun .. (start, end: Number) = mkNumRange(start, end);

printRange(1..10 filter \isOdd map (fun n=n*2)); // 2 6 10 14 18
```

The last line of code contains some interesting features that are worth mentioning. Firstly, the expression `1..10` represents a numeric range [Ale09] constructed using the two-dots operator. Secondly, `(fun n = n*2)` is an anonymous function that returns the double of its argument. Finally, we can chain a sequence of infix operations (`..`, `filter`, `map`) that will make the entire expression natural to the programmer. If we had written the same

	C++	Java	Scala	Haskell	Sparrow
Overloading existing operators	●	○	●	●	●
Prefix op. with custom symbols	○	○	○	○	●
Postfix op. with custom symbols	○	○	●	●	●
Infix op. with custom symbols	○	○	●	●	●
Prefix op. with identifier	○	○	○	○	●
Postfix op. with identifier	○	○	●	○	●
Infix op. with identifier	○	○	●	●	●
Custom preccedence and asssoc.	○	○	○	●	●

Table 6.1: Comparison between programming languages with respect to operators capabilities

expression without operators, similarly to what one writes in a language like C++, it would have become:

```
printRange(map(filter(mkNumRange(1,10), \isOdd), (fun n=n*2)));
```

Although this version is valid, it is much harder to read than what we obtained by using operators.

A representative example for this operator system concerns reference types (a concept similar to C++ references). In Sparrow, if one wants to declare a reference to a type, say `Int`, it writes `@Int`. Even though this pattern is pervasive in Sparrow, this is actually a prefix call to the operator `@` that acts on a type. Due to its hyper-metaprogramming capabilities, Sparrow allows the user to define functions (and operators) that act on types and can return other types. Thus, expressing a reference type becomes an operator call. Other examples of type operators in Sparrow are: `-@` (remove reference), `!@` (ensure at least one reference) and `#$` (get a value of the given type).

All these examples show that by using this method of defining operators we can actually enhance the naturalness of the programming language.

6.3.1 Comparison with other languages

In order to compare Sparrow to other languages in terms of the capabilities of their operators, we have identified several specific features. The degree of support for these features provided by Sparrow and several mainstream programming languages is outlined in table 6.1. The features are self-explanatory.

As we have explained in previous sections, Sparrow supports the complete set of features. The two functional languages come closest to supporting the full set, whereas C++ only supports overloading existing operators. Finally, Java does not allow any form of operator customization.

6.3.2 Limitations

Although this method of defining operators is flexible, extensible, and can improve code naturalness, it has some drawbacks.

The first drawback comes from the fact that the lexer allows all symbol sequences as valid tokens. For example the sequence of two characters `*-` is a valid token. This means that an expression like `a*-b` will be parsed as the infix operation `*-` applied to terms `a` and `b`. In languages like C++ and Java, this would be parsed as `a` times the negative of `b`. Because the lexer always takes the longest possible sequence of characters when forming a token, `*-` will never be interpreted as two separate tokens. To solve this problem, the user must enter a space between the two symbols, transforming the expression into `a* -b`; in this case, the parsing will be similar to the one used in C++ or Java. In most cases however, even in C++, the user will place a space between the two symbols, or use parentheses, just to make the code easier to read.

Another limitation is related to postfix operators. In Sparrow, these can be used either as the last element in an expression, or inside parentheses. Expressions such as `a++ * b`, although valid in C++ (post increment `a` and multiply the old value by `b`), are not valid in Sparrow. This expression is interpreted as: infix ++ between `a` and `*`, and then postfix with an operation named `b`; this is invalid because the second operand must not be a symbol token. There is an easy fix for this, by placing the postfix call in parentheses: `(a++) * b`. Again, in typical scenarios, the programmer rarely applies the postfix increment inside expressions.

The presented method has an additional limitation, but this time not with respect to languages like C++, but to our initial goals. We wanted to allow the possibility of defining different precedence values for operators, depending on the types of the arguments. For example, one may desire a matrix multiplication to have a different precedence than an integer multiplication. Because applying infix operators yields different types based on the order in which the operations are applied, the choice of the highest precedence operator quickly becomes ambiguous. For example let us say that we have the expression `a + b * c`, with all three operands of different (class) types; let us assume that the + operation between `a` and `b` has precedence `10` and the `*` operation between `b` and !c! has precedence 15. Moreover let us assume that the precedence of + between `a` and the result of `(b*c)` is `20` and the precedence of `*` between the result of `(a+b)` and `c` is `5`. In this case, no matter the order in which the operations are executed, it violates the precedence rules.

6.4 Conclusions

We presented a method of constructing flexible operator schemes at the library level. All Sparrow operators, their precedence, and associativity, are defined using this mechanism. Not only is this true even for basic operators (e.g. arithmetic), it also applies to the operator that creates a reference type: `@`.

The grammar rules for operators and expressions are simple and non-restrictive, allowing for a large variety of definable operators. Precedence and associativity can be easily configured using compile-time function calls.

In our evaluation we have provided several examples, showing that custom operators can be useful in a variety of scenarios. Furthermore, using our proposed method, they can be defined and used in a natural way. Despite some limitations, our approach is more powerful and flexible as it provides users with more customization options for their operators compared to other languages.

Finally, as mentioned before, one of our initial goals was to allow type-dependent precedence values. We intend to further investigate this issue and attempt to integrate this type of functionality into our current solution.

Everything should be made as simple as possible, but not simpler.

A. Einstein

Chapter 7 Generics and concepts

Generic programming is the programming paradigm in which basic algorithms and data structures are *lifted* to a more general form that typically does not depend on the concrete implementation types. Instead, the concrete types are passed as parameters to the these generic algorithms and data structures [MS89; Sie05]. Generic programming allows the same code to be used in more than one context, and moreover it allows generating efficient code for each particular use. For example, a generic `sort` algorithm, if properly written, can work on many types (e.g., vector of `Int`s, vector of `Double`s, linked list of complex numbers); depending on the concrete types used for the `sort` algorithm, efficient code will be generated: quick-sort for vectors, merge-sort for linked lists.

Although generic programming was pioneered in ML [Mil78], it was heavily acknowledged with the creation of the STL (Standard Template Library) [SL94] and its inclusion in the C++ standard. The C++ experience shows us that generic programming opens the way to new optimizations, while maintaining the same programming primitives. One can fully take advantage of a compiler's inlining functionality and can use partial specialization to speed up programs [JGS93]. For example, the C++ `sort` generic function is known to be faster than C's `qsort` when sorting a vector of integers [Mey01; Gol06]. On the other hand, C++ templates have their limitations; they are not easy to write and reason about, and when misused they generate long and hard-to-understand error messages [Sie05; GJL+07; VJ02].

Sparrow attempts to build on the generic programming experience to provide methods of producing general abstractions for which the compiler generates efficient code. Using C++ templates as the main starting point, we simplified the language rules, allowing the user to work with abstractions that are already known. Moreover, we provide *concepts*, a method of generalizing classes; they integrate naturally into the language and allow the compiler to provide easy-to-understand messages if the program contains a wrong generic instantiation.

This chapter introduces the generic programming feature from Sparrow. We briefly present the main design decisions of this feature, then we introduce the reader to simple generics and move on to progressively more complex ones. We feature two library components that are frequently used in Sparrow programs: ranges and stream output. Advanced generic topics are covered near the end of the chapter. This chapter concludes with a comparative evaluation of generic programming features; we compare Sparrow to other programming languages with support for generic programming.

7.1 Overview

Based on the experience of generic programming from C++ [Str13; Cpp11], D [Dla; Ale10], $\mathcal{G}$ [Sie05], and the work on concepts for C++ [SR03; GSJR06; SSR13] we set out to create a generic programming system in Sparrow that would be simple, easy to use, and yet powerful (flexible and efficient). The term *simple* here refers to the construction of the feature itself; according to Sparrow's design principles, there should be a limited set of language rules that apply to generic programming. The rules should be intuitive, easy to understand by users, and coherent with the rest of the design decisions in Sparrow.

Having the simplicity principle in mind, the main design decisions related to generic programming in Sparrow are:

1. Macro-like type parameterization (ad-hoc polymorphism) [Str67; CW85; Sie05]
2. Single line of parameters
3. No (complex) type inference rules
4. `if` clauses are used to impose constraints on generics
5. Concepts are predicates on types

The macro-like type parameterization for generics will create a system similar to C++ templates[VJ02]: each time a generic is supplied with a different set of parameter types, a new *instantiation* is created for the generic. Different instantiations will be compiled separately and will be totally independent of each other; they may have totally different implementations. The main disadvantage of this approach is that the guarantees that can be imposed to a generic are less strong than in the case of parametric polymorphism [Sie05]. On the other hand, implementations that use macro-like parameterization are typically faster than the ones that use parametric polymorphism.

C++ templates (which are also the inspiration for generics in D, $\mathcal{G}$, and Java) require the user to specify two sets of parameters for a generic function:

one for the template parameters, and one for the actual function parameters. In Sparrow, the user will specify only one set of parameters for a generic function. The generic (e.g., type) and non-generic (value) parameters can be intermixed as one pleases, and moreover one can use the same parameter for both type and value. We will shortly illustrate this behavior.

C++ has a set of complex rules for type inference. In Sparrow there is one rule for argument type deduction: if the generic can be instantiated with the given arguments, then the types of all the arguments can be used inside the generic. This rule comes natural to the generic programming system, in that the user will hardly notice it.

The **if** clauses and concepts are essentially expressions that are evaluated at compile-time. They can be used to specify constraints for generics and specialize the generics for efficiency. Although concepts are more similar to **if** clauses, they can be viewed as a generalization of types; a concept can be said to be *a predicate on types*—a type can either *model a concept* or not.

7.2 Generics basics

7.2.1 A minimal example

In programming there is often the need to determine the minimum of two numbers. One would typically create a function for this. But what type should the arguments have? In the most general case, this function should work with all integers types, all double types, complex types, and possibly with user defined classes. In generic programming, one can abstract the type of the arguments (and return type), and write the function independently of this type—the type is generic. In Sparrow, this function would be written as:

```
fun min(a, b: @AnyType): typeOf(a) {
    if ( b < a )
        return b;
    return a;
}
```

As can be seen in the code above, the parameters of the function can be of *any type*; the `AnyType` construct is similar to the `auto` type definition in C++ [Str13; Mey14a]. The result of the function will have the type of the first parameter. The `typeOf` function is an intrinsic function that will simply return the type of the expression given as parameter. As the first parameter is taken by reference (note the the `@` operator), the resulting type will also be a reference. For example, if the function were called with two `@Int` arguments, the resulting type would also be `@Int`.

If this generic function were called as `min(2, 5)`, an instance of the generic of the following form would be generated (for mode details see chapter 5):

```
fun min(a, b: @Int): @Int {
    if ( b < a )
        return b;
    return a;
}
```

If the generic function is called with arguments of type `Double` (e.g., `min(2.0, 5.0)`), then a new function instantiation is generated that will have the `a` and `b` parameters declared as `Double`. Different instantiations of this function will not share any code.

It is important to notice that, even if both `a` and `b` are declared under a common `@AnyType` parameter type, they can have different types. Therefore one could easily call `min(2.0, 5)` or `min(2, 5.0)`. The return types for these expressions would be `@Double` and `@Int`, respectively.

If the user wants to always use the wider type, it can use the `commonType` compile-time function. This takes two types as arguments and returns the widest type. In general, for two types T and U, if T is convertible to U, but U is not convertible to T, then `commonType(`T`,`U`) == commonType(`U`,`T`) ==` T. With this function, our code would look like this:

```
fun min(a, b: @AnyType): commonType(typeOf(a), typeOf(b)) {
    if ( b < a )
        return b;
    return a;
}
```

In C++ 03, one would write a `min` function like this:

```
template <class T>
const T& min(const T& a, const T& b) {
    if ( b < a )
        return b;
    return a;
}
```

From a syntactic point of view, there are fewer constructs used: in Sparrow there is no need to specify an additional template parameter. Although for this simple example the extra template parameter is not a syntactic burden, for more complex examples it may become one.

With this simple example, we have already covered the first three main design decisions. We have shown that different parameter types will generate different instantiations of the generic function, according to the macro-like polymorphism principles. If in C++ one would write two different sets of parameters, in Sparrow only one set is needed. We have also shown the only rule of type deduction that Sparrow has: whenever the function is instantiated with a new type of argument, that type will be implicitly used inside the generic. This rule is so intuitive, that the user hardly thinks of it when using generics.

7.2.2 Specializing the implementation

Another useful construct is the code that swaps the contents of two objects. As the algorithm is (mostly) the same for any type of objects, it can be encoded as a generic function:

```
fun swap(a, b: @AnyType) {
    var tmp = a;
    a = b;
    b = tmp;
}
```

Here, the type of the `tmp` variable is deduced to be the type (without references) of the first argument.

Although this form of swapping can be used for any value type that preserves the semantics for assignment, there are cases in which this implementation is suboptimal.

Let us take for example the case of vectors. If we are trying to swap two vectors with this implementation, we create a new vector, allocate memory for storing the elements from `a`, copy all the elements from `b` to `a`, and then copy all the elements from the temporary vector to `b`. Depending on the lengths of the two vectors, in the last two lines of the algorithms it is very probable that we need to reallocate the destination vectors. In summary, we would typically have two vector allocations, and three full copies of the vector—overall $O(n)$ complexity.

The swapping operation can be performed efficiently on vectors just by swapping the internal pointers [1]. This is be performed in $O(1)$ complexity, without any memory allocation. Therefore, most vector implementations contain an inner `swap` function that implements this optimized swapping.

In addition to the previously defined `swap` function, we can add a specialized version:

```
fun swap(a, b: @AnyType) if isValid(a.swap(b)) {
    a.swap(b);
}
```

For this generic definition we used an *if clause*. The expression given after **`if`** must be an expression that can be evaluated at compile-time. If the expression evaluates to true (for the given parameter types), then the function is defined; otherwise, the function is not defined. The `isValid` function evaluates to true if the given expression has a valid type in Sparrow; if the given expression is not semantically correct, it will return false without raising an error. The **`if`** clause behaves similarly to the `enable_if` idiom from C++ ([JWL03; Cpp11], and with Template constraints from D [Dla].

[1]a typical vector implementation contains three pointers: begin, end, and end-of-storage pointers

Borrowing the idea from C++'s SFINAE[2] feature [Cpp11], if the condition passed to an **if** clause is not semantically valid, then the whole condition is considered false, and no error is reported to the user. For example, in the following code, no error is reported if the function is called with an `Int` argument which does not have the `getValue` method defined:

```
fun f(x: AnyType) if typeOf(x.getValue()) == Int {...}
```

The **if** clauses are a very important instrument for generic programming in Sparrow. One can encode in these clauses arbitrarily complex conditions and constraints that can be checked when instantiating generics[3]. Considering the fact that generics in Sparrow are ad-hoc, this feature can help in making the generics type-safe. This can also help in optimizing code, as we will see below.

Returning to our `swap` example, having the two functions defined, we cover both the generic and the specialized cases. If the type of the first argument does not contain a `swap` method, the first function will be called—the second one will not even be defined. If, on the other hand, the type of the first argument contains a `swap` method, the second function will be defined, and, at the invocation the compiler must choose between two overloads. The compiler will always choose the most specialized one—in this case the one that has an **if** clause is more specialized than the one that does not have it.

Calling the two forms of the `swap` generic is as simple as calling a regular function:

```
var n1 = 2;
var n2 = 5;
swap(n1, n2);            // Calls the general function
var v1: MyVector = ...;
var v2: MyVector = ...;
swap(v1, v2);            // Calls the specialized function
```

Note that the user does not have to specify at any point the type of the arguments passed to the generic. They are automatically deduced from the types of the argument expressions (please refer to chapter 5 for a discussion on how these generics are instantiated).

In practice, the user may want to impose an additional constraint to the `swap` functions: the types of the arguments should be the same. This can be easily achieved by adding an `typeOf(a) == typeOf(b)` condition.

In conclusion, with these two generics we cover both the general case and the case in which we have a more efficient way of swapping between elements of a type. The caller of the generic function does not have to know the details

[2]Substitution Failure Is Not An Error

[3]given Sparrow's ability for hyper-metaprogramming, one can write arbitrarily complex conditions as simple expressions to be executed at compile-time

of the implementation. For him/her, the `swap` algorithm just works, and it does it in the most efficient way.

7.2.3 A simple concept

"A type that contains the `swap` method" is a common pattern, especially throughout the containers of a standard library. It is common enough that it deserves its own name: `Swappable`. We say that a type models the `Swappable` *concept* if it contains a `swap` method with one parameter, and we can call this method by passing an object of the same type. Sparrow has a simple but powerful support for declaring concepts. One would define the `Swappable` concept in the following way:

```
concept Swappable(x) if isValid(x.swap(x));
```

In this example, `x` is a variable denoting a value of any type. The **`if`** condition in this concept definition has the exact meaning as a standard **`if`** clause[4]. Therefore, a type *models* the `Swappable` concepts if for values of that type the condition `isValid(x.swap(x))` evaluates to true.

With this concept defined, one can write our `swap` function in two new ways (also imposing the types of the arguments to be the same):

```
fun swap_old(a, b: @AnyType)
    if isValid(a.swap(a)) && typeOf(a) == typeOf(b) {...}
fun swap_c_1(a, b: @AnyType)
    if Swappable(typeOf(a)) && typeOf(a) == typeOf(b) {...}
fun swap_c_2(a, b: @Swappable)
    if typeOf(a) == typeOf(b) {...}
```

Ignoring concept specialization (which will be presented in section 7.3.1), the three forms of the function are equivalent. We can use this equivalence to define the semantics of a concept: a concept is a form of imposing a constraint on a type—it is a predicate on types. Syntactically, it can appear as a replacement for a type name (`swap_c_2`), or it can be used as a predicate call (`swap_c_1`). Of course, the user will find the `swap_c_2` more convenient to use. Writing concepts instead of types can be a very good practice with respect to generic programming: it is easy, concise, and can improve type-safety.

As the reader may have already guessed, the general form for defining a concept is:

```
concept <name>(<typevar> [ : <base_concept>]^opt) if <ct_condition>;
```

The `: <base_concept>` part is optional. When it is not specified, `AnyType` is implicitly assumed.

[4]the current version of the Sparrow compiler actually shares the instantiation and condition checking code between generics and concepts

```
 1 class Vector(valueType: Type) {
 2     using ValueType = valueType;
 3     using ValuePassType = !@valueType;
 4     using RangeType = ContiguousMemoryRange(valueType);
 5     fun ctor(other: @Vector) {...}
 6     fun ctor(range: Range) {...}
 7     fun swap(other: @Vector) {
 8         begin swap other.begin;
 9         end swap other.end;
10         endOfStore swap other.endOfStore;
11     }
12     fun isEmpty              = begin == end;
13     fun size: SizeType       = end diff begin;
14     fun capacity: SizeType   = endOfStore diff begin;
15     fun ()(index: SizeType)  = (begin advance index).value;
16     fun front                = begin.value;
17     fun back                 = (end advance -1).value;
18     fun all                  = RangeType(begin, end);
19     private var begin, end, endOfStore: RawPtr(ValueType);
20 }
```

Listing 7.1: Example of a generic vector (simplified)

The semantics of this is the following: a type T *models* the concept if T models `base_concept` and the condition is true for that type. Of course, any type is said to model `AnyType`.

Having an optional base concept allows users to define specialization relationships between concepts. Therefore, one can specify that a concept is more *specialized*, more restrictive than other concept. This information can be used in function overloading when selecting the most specialized function. Examples of this can be found in section 7.3.1.

7.2.4 A generic vector

So far we have shown a few examples of function generics. Generics can also be used for classes. A simplified vector generic class is presented in listing 7.1.

It is important to notice the parameters of the generic class at line 1. Unlike in C++, D, $\mathcal{G}$ or Java, Sparrow uses normal parentheses for generic parameters. The parameters are placed just like in the case of regular functions. From the user's perspective, using such a class requires a set of parameters, just like functions do—no need to learn a new notion like generic parameters.

In our case, `valueType` is a variable that can be used anywhere a type can be used. `Type` is just a type that can be used at compile-time to denote another type. Classes and functions can be also parameterized with other compile-time parameters that are not types. For example, one can easily add numbers, strings, and user-defined types as parameters to a generic as long as they can be used at compile-time.

Similar to C++'s **`typedef`**, we used the **`using`** directive to create internal aliases. These are just names that will be expanded to the defining terms whenever they are used.

The instantiation of class generics is shown in the same listing at lines 4 and 19, specifically for the `ContiguousMemoryRange` and `RawPtr` class generics.

7.2.5 More insights on generics

In Sparrow, generics are declarations parameterized using compile-time parameters. One can think of them as compile-time functions from a set of compile-time arguments to a Sparrow declaration.

If a class has parameters, then of all these parameters must be compile-time—a parameterized class is always a generic.

Things are slightly different for functions: they can have both generic parameters and non-generic parameters. For a function that should be executed at run-time, all the compile-time parameters and all the concept parameters (including `AnyType` parameters) are generic parameters. The rest of the parameters are non-generic.

For the following generic function, the `g` parameters are generic, whereas the `ng` parameters are non-generic:

```
fun genericFun(g1: Int ct, ng1: Int,
               g2: String ct, ng2: String,
               g3: AnyType, g4: Range) {...}
```

We started this subsection by defining generics as declarations parameterized on compile-time parameters. We need to explain why `AnyType` is a compile-time parameter. After all, the user is allowed to call the `min` generic function (explained in section 7.2.1) by passing two run-time variables, so where are the compile-time parameters? The caveat here is that the compiler will internally generate two parameters for each concept parameter: one compile-time parameter that tracks the type of the argument, and one run-time parameter to actually pass the argument of the selected type. The `min` function corresponds to the following C++ code, the only difference being that it implicitly handles the `T1` and `T2` template parameters:

```
template<typename T1, typename T2>
T1 min(const T1& a, const T2& b) {...}
```

In practice however, the user does not have to know about the generated compile-time parameters. The system simply works. When calling the `min` function the user will see `AnyType` as a parameter and therefore it knows that it can call it with any kind of object. When analyzing the body of the function, the user reads the same `AnyType` and can have the following reasoning: *this is a parameter with a fixed, well-defined type, but I do not know what that type is; when this generic is instantiated, the compiler knows what the type is*. Although the semantics are different, from the user point of view, this feature is similar to C++'s way of declaring variables with no type: `auto x = <complex-expression>`.

This is the only rule for implicit type deduction that Sparrow has with respect to generics. Compared to C++, where the rules for type argument deduction are complex [VJ02], Sparrow greatly simplifies the way type deduction is performed.

7.2.6 One problem, multiple solutions

From our experience, Sparrow generics can express almost all types of generics found in a typical implementation of the STL [Jos12]. The only notable exceptions are constructs of the following form:

```
template<typename T>
void foo(vector<T> v) {...}
```

This imposes a constraint on the type of the argument to be an instance of the `vector` template. Because in C++ the template parameters are expressed before the actual function parameters, one parameter type can always refer to a template parameter. Also, in C++ this works with implicit type deduction; if this function is called with a `vector` object the compiler is able to deduce the type of `T`.

The proper use of concepts in Sparrow reduces the incidence of these cases. After all, according to generic programming principles [MS89; Sie05], one should strive to parameterize over general concepts and not over concrete (in most cases implementation-specific) class templates.

One solution for this problem is to use the *tag dispatching* idiom [SL94]. By placing a special tag inside the vector class, one can test for the presence of that tag:

```
fun foo(v: @AnyType) if isValid(v.thisIsAVectorTag) {...}
```

This solution is based on the fact that it is unlikely that classes other than `Vector` have the same tag defined. However this is not a guarantee.

Similar to the tag dispatching solution, we can use the actual type name:

```
fun foo(v: @AnyType)
    if TypeOp.description(typeOf(v)) == "Vector" {...}
```

Again, this is not a foolproof solution, as different classes can have the same name (in different packages).

Another solution is to define `using` clauses for the parameters of the generic. This technique is also usable in other contexts. In listing 7.1 where we described a simplified `Vector` definition, we included the `ValueType` name inside `Vector`. With this defined, one can uniquely determine whether we are instantiating with a `Vector` object, by re-instantiating the `Vector` class:

```
fun foo(v: @AnyType) if typeOf(v) == Vector(v.ValueType) {...}
```

This is more tedious to write than the similar code in C++, but it solves our problem.

Even if we do not yet have an elegant solution for this problem, we still believe that generic programming in Sparrow is easier to use and reason about than what we find in C++. In most cases, the user never has to bother with type argument deduction rules. The more advanced users that will encounter this problem will know how to work around it.

We discussed so far all the important design principles related to generics mentioned above in section 7.1. There is no other important feature of generics that the user should know about, and has not been presented yet. These few principles are enough to generate a complete system for generic programming. In the rest of this chapter we will present more examples of how generics can be used to write natural and efficient code. All these point towards an inherent flexibility of the programming language that allows these kinds of generics.

7.3 Ranges

7.3.1 Range concepts

Given a vector of numbers, how should one iterate over its elements? More generally, given a collection of elements, what are the proper abstractions to use to iterate sequentially over the elements in the collection?

In C++'s STL world one would use *iterators* [SL94; Jos12]. A *begin* and an *end* iterator are needed to represent the *range of elements* of the collection. Then, starting from the begin iterator, one needs to *increment* an iterator *until it is equal to the end iterator*. As we need two iterators to represent a range, every algorithm that iterates over that range would need two iterators as inputs; the programmer must make sure that in all cases, the paired iterators are well synchronized. Looking from a naturalness perspective, this solution seems not very intuitive; the user has plenty of notions to think of.

To overcome these problems Alexandrescu [Ale09] proposed a new solution for iterating over a sequence of elements: *ranges*.

A *range* is an abstraction that allows the user to iterate over a collection of elements that can be arranged into a sequential order[5]. As Alexandrescu points out, a range should support a minimum of three operations:

[5]Although one can iterate over non-linear structures, there is always a perfectly defined order in which all the elements in the collection are traversed; defining such an order will essentially make the structure linear with respect to this order

- Test whether the range is empty.
- Access the first element in the range.
- Advance the range to the next element; remove the first element from the range.

These three operations are enough to model the behavior of a pair of C++'s input iterators [Jos12; SL94]. Any algorithm that can be written with input iterators can also be written with ranges.

Of course, in Sparrow we define a concept for a range:

```
concept Range(x)
    if typeOf(x.RetType) == Type
    && typeOf(x.isEmpty()) == Bool
    && typeOf(x.front()) == x.RetType
    && isValid(x.popFront())
    ;
```

In addition to the three operations required for a range, we find it useful to impose for all the ranges to define an inner type `RetType` that would indicate the type of elements returned while iterating over the range.

Iterating over a range `r` is straightforward[6]:

```
while ( !r.isEmpty() ; r.popFront() )
    cout << r.front() << endl;
```

Simpler, one can write the same thing with a **`for`** structure:

```
for ( x = r ) {
    cout << x << endl;
```

Inside the **`for`** loop `x` is a new variable of the type `r.RetType`. The **`for`** is perfectly equivalent to the **`while`** structure above.

Instead of returning iterators as the STL does, the containers from the Sparrow standard library yield ranges to be used for iteration. Therefore, to iterate over all the elements in a vector, one would typically write:

```
var myVec: Vector(Int) = ...;
for ( elem = myVec.all() )
    cout << elem << end;
```

Compared to the traditional **`for`** structure from C++, our version is much cleaner. The 2011 version of the C++ standard introduced the so-called *range-based for loops*. Although it sounds similar, these *ranges* are not similar to what we have in Sparrow. The new **`for`** structure from C++ actually works on containers:

```
for ( auto x : myVec )
    cout << x << endl;
```

[6]as discussed in chapter 5, in Sparrow, a **`while`** loop also accepts a *step* action that is executed between two body executions—this is similar to the step part of a **`for`** loop from C++

As most of the STL algorithms work on iterators and not on containers, the new C++ `for` structure is not as useful as what we have in Sparrow. One can always create a container-like class to wrap over a pair of iterators, but this is cumbersome.

Out of the iterator categories used in C++, our `Range` concept only covers the functionality of an `input_iterator`. An `output_iterator` can be implemented with the same concept if the `RetType` is a reference type. We still need to cover the concepts corresponding to `reverse_iterator` and `random_access_iterator`. In Sparrow, they are defined as follows:

```
concept BidirRange(x: Range)
    if typeOf(x.back()) == x.RetType
    && isValid(x.popBack())
    ;
concept RandomAccessRange(x: BidirRange)
    if typeOf(x.size()) == SizeType
    && typeOf(x(0)) == x.RetType
    ;
```

A bidirectional range has all the requirements of a `Range` plus a method of accessing the last element from the sequence, and a method of removing it. One can use this type of range to iterate backwards over the elements. A random access range offers all the functionality of a bidirectional range, and it should also allow accessing elements in the sequence directly. For a random access range we also impose the constraint that it must have a `size` function. Although for other types of ranges we are not required to know the size of the range in order to traverse it, for a random access range we need to know its size, so that we know what is the maximum value that we can pass to the direct access function (call operator).

In this example, we imposed specialization relations between the concepts as mentioned in section 7.2.3. A `RandomAccessRange` is a specialization of a `BidirRange` which is in turn a specialization of `Range`. Firstly, this means that the specialized concepts will *inherit* all the restrictions of the base concepts. Secondly, this relationship can be used in function overloading when selecting the most *specialized version* of a function—section 7.3.3 presents a simple example in which this comes in handy.

7.3.2 C++ like interface

The user familiar with C++ may find it more convenient to use ranges as iterators. They would prefer the `++` and `--` operators instead of the `popFront` and `popBack` methods. We have a solution for these users that will leave the concepts untouched. The following function definitions are enough:

```
fun pre_++(r: @Range): r.RetType {
    r.popFront();
    return r.front();
}
fun post_++(r: @Range): r.RetType {
    var res = r.front();
    r.popFront();
    return res;
}
fun pre_--(r: @BidirRange): r.RetType {
    r.popBack();
    return r.back();
}
fun post_--(r: @BidirRange): r.RetType {
    var res = r.back();
    r.popBack();
    return res;
}
fun pre_!(r: @Range) = r.isEmpty();
fun pre_!!(r: @Range) = !r.isEmpty();
fun pre_*(r: @Range): r.RetType = r.front();
```

Sparrow allows these functions to be defined externally. The fact that with ranges we only need one object over all the iteration sequence allows one to define any operators on ranges. For example, one may want to define the + operator on ranges to concatenate them. Once again, Sparrow proved to be a flexible language for creating abstractions that are easy to use.

7.3.3 Examples of range algorithms

This section exemplifies how some of the algorithms on ranges can be written.

Let us define a simple algorithm that will compute the number of elements in a range. For a standard range or a bidirectional range this would be an $O(n)$ operation, but for random access ranges this is an $O(1)$ operation. One would write this efficiently in the following way:

```
fun rangeSize(range: Range): SizeType {
    var n: SizeType = 0;
    while ( !range.isEmpty() ; range.popFront() )
        ++n;
    return n;
}
fun rangeSize(range: RandomAccessRange): SizeType = range.size();
```

We give a simple example of concept specialization. Whenever we call `rangeSize` with a range that is not random access the first overload will be selected (as it is the only one valid). If a random access range is supplied to the `rangeSize` function, then both functions are valid and the compiler needs to select the most specialized function. In this case, `RandomAccessRange` is a specialization of `Range`, so the second method will be chosen. Again, we defined a general algorithm that works in all cases, and for special cases in which optimizations can be performed we choose a specialized version.

One widely used operation on collections of elements is determining whether the collection contains a given value or not; if it does, the algorithm must return something that identifies the position in which the element was found. As Alexandrescu argues [Ale09], such a function should return a range starting with the position of the found element; an empty range would indicate that the value was not found within the range. The following code implements this algorithm in Sparrow:

```
fun find(r: Range, value: @AnyType): typeOf(r)
        if isValid(r.front() == value) {
    while ( !r.isEmpty() && !(r.front() == value) )
        r.popFront();
    return r;
}
...
var r = find(v.all(), value);
if ( !r.isEmpty() ) cout << "Found!";
```

The reader should remark how easy it is to specify all the requirements associated with this algorithm. Declaring `r` as `Range` guarantees that we can iterate over it; the **`if`** clause guarantees that we can compare the given value to the elements of the range. No other constraint is required for this algorithm.

The following algorithm copies all the elements from the first range to the second range and returns the range of untouched elements from the second range; this time we will use the range operators defined above:

```
fun copy(r1, r2: Range): typeOf(r2) if isValid(*r2 = *r1) {
    while ( !!r1 && !!r2 ) {
        *r2 = *r1;
        ++r1;
        ++r2;
    }
    return r2;
}
```

Again, this function expresses all the required constraints directly into the code. Both arguments must be ranges and we must be able to copy the elements from `r1` to `r2`. Concise, easy, and safe.

Let us see now how we can traverse a bidirectional range in reverse order. For that, the easiest way is to simply *reverse the range*, as shown in listing 7.2. As can be seen, no elements are swapped, but, by using the `retro` function, the range is logically reversed. This does not induce any penalty on execution time or storage space.

```
class[initCtor] RetroRange(rangeType: Type)
        if BidirRange(#$rangeType) {
    using RetType      = rangeType.RetType;
    fun isEmpty        = range.isEmpty();
    fun front: RetType = range.back();
    fun popFront       { range.popBack(); }
    fun back: RetType  = range.front();
    fun popBack        { range.popFront(); }
    private var range: rangeType;
}
fun retro(range: BidirRange) = RetroRange(typeOf(range))(range);
...
for ( x = retro(v.all()) )
    cout << x << endl;
```

Listing 7.2: Traversing a range in reverse order

The `RetroRange` class expects a `BidirRange` type as a generic argument[7]. In turn, the `RetroRange` class satisfies the constrains of the `BidirRange` concept. The `[initCtor]` modifier instructs the compiler to automatically generate a construct that takes one parameter corresponding to the `range` class variable.

Similar to the approach of reversing a range, we defined range transformation operations, such as: take the first N elements from the range, take elements while a predicate is true, traverse a range using a stride, traverse a range in radial fashion, or cycle through a range several times. Any of these range operations are efficient as they only manipulate the way the access to the elements is done.

Sometimes it is very useful to create ranges that simply repeat a value. The `repeat` function from listing 7.3 creates a range by repeating a value indefinitely or for a given number of times.

As there is no restriction on the number of elements in a range, we can easily create ranges with an infinity of elements. Moreover, as ranges provide an indirection over the actual values in a collection of elements, there is no need to allocate memory for these elements—that would be the job of the containers. This way, the implementation of the first `repeat` does not need to allocate space for all the elements, and does not need to run intense computations to create the range. It will just copy the given value each time `popFront()` is called. This proves to be a powerful feature of ranges [Ale09].

[7]The **if** clause actually means `BidirRange(valueOfType(rangeType))` and checks whether the `rangeType` parameter satisfies the `BidirRange` concept

```
class[initCtor] RepeatRange(valueType: Type) {
    using RetType       = valueType;
    fun isEmpty         = false;
    fun front: RetType  = value;
    fun popFront        {}
    private var value: valueType;
}
fun repeat(value: AnyType) = RepeatRange(typeOf(value))(value);
fun repeat(value: AnyType, count: SizeType) = take(repeat(value), count);
...
for ( x = repeat("repetition is the mother of learning", 10) )
    cout << x << endl;
```

Listing 7.3: Repeating the same value to create a range

7.3.4 Numeric ranges

Let us consider the first ten natural numbers: $1, 2, 3, ...10$. As this is a sequence of elements and we have an order to access them one-by-one, we can create a range that will return these numbers. More generally, all the integers in a range $[n, m)$ can be represented by a generator range that is initialized with the values n and m.

In Sparrow, we can create numeric ranges with the following functions:

```
fun numericRange(start, end, step: Number) =
    NumericRangeWithStep(commonType(typeOf(start), typeOf(end)))(
    start, end, step);
fun numericRange(start, end: Number) = NumericRangeInc(
    commonType(typeOf(start), typeOf(end)))(start, end);
fun ..(start, end: Number) = NumericRangeInc(commonType(typeOf(
    start), typeOf(end)))(start, end);
fun ...(start, end: Number) = NumericRangeInc(commonType(typeOf(
    start), typeOf(end)))(start, end, true);
fun post_..(start: Number) = NumericRangeInc(typeOf(start))(
    start, NumericLimits(typeOf(start)).maxValue);
fun ../(range: Range, step: Number) = NumericRangeWithStep(range
    .RetType)(range, step);
```

These functions will create numeric ranges of any number type[8] within the given bounds, possibly with a given step. The type of the returned elements is the common type of the given start and end numbers. The interval is always closed on the start value (the start value is always returned by the range) but it can be open or closed for the last value (the last value may or may not be returned by the range). The `..` creates an open-end interval, while the `...` operator includes the last number in the range. The postfix, `..` operator creates

[8] `Numeric` is a concept that is satisfied by all the standard integer and floating point types

an infinite range starting with the given number. The last operator modifies an existing `NumericRange` range to specify its step. We take advantage here of Sparrow's ability to allow user-defined operators of any form (see previous chapter).

Like any other ranges, the values in these ranges can be traversed with a **for** loop. The following code shows how these range generator functions work:

```
fun printRange(r: Range) {
    for ( i = r )
        cout << i << ' ';
}
printRange( numericRange(1, 5) );       // 1 2 3 4
printRange( numericRange(1, 10, 2) );   // 1 3 5 7 9
printRange( 1..5 );                     // 1 2 3 4
printRange( 1...5 );                    // 1 2 3 4 5
printRange( 1..10../2 );                // 1 3 5 7 9
printRange( 3.. );                      // 3 4 5 ...
                                        // stops on overflow
```

This shows how simple it is to create user-friendly constructs in Sparrow. Not only is this an improvement in naturalness, it also shows that Sparrow is a flexible enough language that allows the user to construct powerful features that are easy to use.

7.3.5 Getting more functional

Continuing in the same direction, in Sparrow it is easy to create operations on ranges that take functions as parameters. As we shall see shortly, this resembles the type of functions that one would typically find in functional programming.

In the most general form, a *functor* is an entity for which we can apply the `()` operator (the call operator). Regular functions are just a particular case of this. A functor has a well defined type, and we can pass it as a parameter to generic functions.

As an example, let us consider the algorithm that finds the first element that satisfies a given predicate:

```
fun findIf(r: Range, pred: AnyType): typeOf(r)
        if typeOf(pred(*r)) == Bool {
    while ( !!r && !pred(*r) )
        ++r;
    return r;
}
```

This algorithm is very similar to the `find` algorithm presented in section 7.3.3, but instead of testing for value equality, it calls the predicate. The only requirement for the predicate is that it be callable with elements from the given range, and produce a `Bool`-like result.

This algorithm can be called with a normal function, a variable of a type that has the call operator defined, or even a lambda function, as shown in the following code:

```
fun isEven(x: Int) = x%2 == 0;
class IsOdd {
    fun ()(n: Int) = n%2 == 1;
}
var isOddVar: IsOdd;
findIf(1..10, \isEven);            // 2 3 4 ...
findIf(1..10, isOddVar);           // 1 2 3 ...
findIf(1..10, (fun n = n%2 == 0);  // 2 3 4 ...
```

Two of the most important functions that are omnipresent in functional languages are `filter` and `map`. The first function takes a collection of elements and returns a collection that contains only the elements that satisfy the given predicate[9]. The second function transforms the given collection into another collection by applying a transformation function to each element. Their Sparrow implementations are shown in listings 7.4 and 7.5. Instead of using these functions on collections, they operate on ranges.

Using these functions is easy, as expected:

```
printRange( filter(1..10, \isOdd) );      // 1 3 5 7 9
printRange( map(1..5, (fun n = n*2)) );  // 2 4 6 8 10
```

Taking advantage of the flexible operator system from Sparrow, we can also write the function calls as infix operators:

```
printRange( 1..10 filter \isOdd );      // 1 3 5 7 9
printRange( 1..5 map (fun n = n*2) );  // 2 4 6 8 10
```

Although this is a subjective matter, one might argue that this form is easier to write and understand than the previous one.

The user can further take advantage of the infix operator rules and combine multiple range operations into one expression:

```
printRange(1..10 filter \isOdd map (fun n=n*2)); // 2 6 10 14 18
```

Reading this expression from left to right also describes exactly the computations that take place: we have a range of numbers from 1 to 10 (inclusive), we use filter to keep only the odd numbers, and then we map to another range in which the values are doubled. We believe that most readers will find this code easy to read (if they are familiar with the `filter` and `map` semantics).

[9]adding a backslash before a function name tells the compiler we want to take the reference of the function instead of calling the function; we need to make this distinction because in Sparrow a function can be called without any parenthesis

```
class FilteredRange(rangeType, predType: Type)
        if Range(#$rangeType)
        && typeOf((#$predType)(#$rangeType front)) == Bool
{
    using RetType = -@rangeType.RetType;
    fun ctor(range: rangeType, pred: predType) {
        this.range ctor range;
        this.pred ctor pred;
        this.lastVal ctor;
        popUntilValid;
    }
    fun isEmpty          = range.isEmpty();
    fun front: RetType   = lastVal;
    fun popFront         { range.popFront(); popUntilValid; }
    private fun popUntilValid {
        while ( !range.isEmpty ) {
            lastVal = range.front();
            if ( pred(lastVal) )
                break;
            range.popFront();
        }
    }
    private var range: rangeType;
    private var pred: predType;
    private var lastVal: RetType;
}
fun filter(r: Range, pred: AnyType)
    = FilteredRange(typeOf(r), typeOf(pred))(r, pred);
```

Listing 7.4: Definition of a filter operation on ranges

7.3.6 A note on performance

The reader may wonder whether there are some execution costs associated with the usage of ranges and the algorithms defined on them. This concern is justified by the use of multiple abstractions in our code snippets (e.g., constructing a range of numbers, which is then filtered with a predicate to produce another range, which is then mapped by a closure to another range).

The next chapter presents some measurements which show that there are no costs associated with using these abstractions.

The main reason for not paying a performance penalty is that operations on ranges are, by design, performed lazily. For example, in the case of `map`, instead of transforming all the elements when calling the `map` function, we apply the transformation each time an element from the resulting range is accessed. This laziness even allows one to call a function like `map` on infinite ranges, like in the following example:

```
class TransformedRange(rangeType, funType: Type)
        if Range(#$rangeType)
        && isValid((#$funType)(#$rangeType front)) {
    using RetType = -@typeOf((#$funType)(#$rangeType front));
    fun ctor(range: rangeType, function: funType) {
        this.range ctor range;
        this.function ctor function;
        this.curVal ctor;
        this.hasValue ctor false;
    }
    fun isEmpty         = range.isEmpty();
    fun front: RetType {
        if ( !hasValue ) {
            curVal = function(range.front());
            hasValue = true;
        }
        return curVal;
    }
    fun popFront        { range.popFront(); hasValue = false; }
    private var range: rangeType;
    private var function: funType;
    private var curVal: RetType;
    private var hasValue: Bool;
}
fun map(r: Range, f: AnyType)
    = TransformedRange(typeOf(r), typeOf(f))(r, f);
```

Listing 7.5: Definition of a map operation on ranges

```
randomRange(1, 100) map (fun n=n*n)
```

Assuming that the `randomRange` function is a random number generator with values in the range $[1..100)$, the result will be a range that will indefinitely return random square numbers in range $[1..10000)$. The cost of obtaining a new such random square number is essentially just the cost of running the number generator.

Another factor that helps in generating fast code is the fact that Sparrow's compiler backend (LLVM based) will apply function inlining whenever possible. In our case, most of the functions presented here are very simple, so in the optimized binary they will be inlined.

Analyzing the content of the range helper classes presented above, we can easily conclude that ranges will use $O(1)$ memory space.

7.4 Streams example

To prove that Sparrow is flexible enough to generate easy to use and reason about code, we want to consider the implementation of a stream output framework. The goals for such a system would be:

- usable with Sparrow's standard types
- extensible with any user-defined type, if support is added
- make use of an operator syntax, similar to the stream output syntax from C++
- support for multiple targets: console, file, strings, user-defined output targets

We define a stream concept that requires that basic types can be written to the stream:

```
concept OutStream(x)
    if isValid(x.<<<(1L))
    && isValid(x.<<<(1.0))
    && isValid(x.<<<('c'))
    && isValid(x.<<<("string"))
    ;
```

This concept requires that any class that models it must implement the `<<<` operators on the standard numeric types, strings, and the `Char` type[10].

The following code presents an implementation of the `ConsoleOutputStream` concrete class that is able to write to the console:

```
class ConsoleOutputStream {
    fun <<<(x: Number)    { Impl.write(x); }
    fun <<<(x: Char)      { Impl.write(x); }
    fun <<<(x: Bool)      { this <<< (x ? "true" : "false"); }
    fun <<<(x: StringRef) { Impl.write(x)); }
    // This stream supports flush
    fun flush             { Impl.flushOutput(); }
}
var cout: ConsoleOutputStream;
```

Here, `Impl.write` is a series of overloaded functions that can write numeric values and string values to the console; these can be implemented in terms of the `printf` function from C. `Impl.flushOutput` flushes the standard output stream (implemented using `fflush(stdout)`).

Although not shown here, creating streams that write to files, a memory buffer, a string, or any user-defined structure is easy: the programmer only has to provide the functions to write the primitive types.

[10]The implementer may not define all the functions corresponding to all the standard types, by taking advantage of some implicit conversions. For example writing a `Short` value to the stream can be done by calling the function that writes a `Long`

To actually write into a generic stream we use the following two specializations of the << operator:

```
// If we define an ">>" operator in the type itself, make sure
// we can support the standard "<<" notation
fun << (s: @OutStream, x: @AnyType): typeOf(s)
        if isValid(x.>>(s)) {
    x.>>(s);
    return s;
}
// If no ">>" operator is defined in the class, try to place the
// input element in the stream. The stream class must define a
// "<<<" operator for this element
fun << (s: @OutStream, x: @AnyType): typeOf(s)
        if !isValid(x.>>(s)) && isValid(s.<<<(x)) {
    s.<<<(x);
    return s;
}
```

This is the most important functionality that we need in order to make the streams work as required. If we call the << operator with a user-defined type, then this type must define the >> operator inside; otherwise the stream must directly support writing that type into it.

Please note that we use the <<< inside streams so that it does not collide with the << operator that we generally use; whenever the compiler sees an `a << b` expression it will search for the operator among the members of the class of `a`; if it finds an operator with the same name, it will stop the name searching algorithm; therefore the externally defined << operators will never be called. See chapter 6 for more details on the name search algorithm for operators.

For completeness we will define two new helpful abstractions: writing newlines to the streams and flushing the streams. The following code shows their implementation:

```
class EndLineHelperClass {
    fun >>(os: @OutStream)  { os << '\n'; }
}
class FlushHelperClass {
    fun >>(os: @OutStream) {
        if[ct] ( isValid(os.flush()) )
            os.flush();
    }
}
var endl: EndLineHelperClass;
var flush: FlushHelperClass;
```

We believe that by now the reader is accustomed with the ease with which powerful features can be created in Sparrow, and is not surprised to see such a small code.

The `endl` variable, when it is passed to a stream, simply writes the new-line character (and does not flush the stream as in C++). The `flush` variable will try to flush the stream if the steam has a `flush` function defined. For the streams that do not have this defined, it does nothing. We use a compile-time **`if`** structure to perform the check at compile-time.

As previously seen, using streams in Sparrow is also easy:

```
class[initCtor] MyPair {
    var first, second: Int;
    fun >>(os: @OutStream) {
        os << '<' << first << ", " << second << '>';
    }
}
cout << "some string" << endl;             // one line, no flush
cout << "another string" << endl << flush; // with flush
cout << "pair: " << MyPair(7, 12) << endl; // "A pair: <7, 12>\n"
```

7.5 Advanced topics

7.5.1 The definition of AnyType

Although we have mentioned `AnyType` several times so far, we have never defined it. `AnyType` is actually a concept, defined in the standard library, and not in the Sparrow language itself. It can be defined in the following way:

```
concept AnyType(x) if true;
```

Similar to a generic function which is always valid without an **`if`** clause, we can omit the **`if true`** part completely:

```
concept AnyType(x); // Always true
```

Typically programming languages define basic constructs in the languages themselves, and not as library features. Again, adding to the flexibility of the Sparrow language, a basic building-block of the generic programming is defined in the library—additional confirmation that Sparrow is *designed for growth* [Ste99].

7.5.2 On types modeling concepts

One may find the way model checking is performed to be very peculiar. Instead of having a dedicated syntax for specifying concepts, with methods to specify the existence of various variables, functions, or types, we have a simple syntax with a single compile-time expression that resembles more closely function definition. Instead of providing language support for checking for the existence of variables, functions, and types, we choose not to do it. We rely on compile-time expressions to express all the constraints.

We started from the definition of a concept as a *predicate on types* [Sie05; DS06; Jon92]. If this definition fully specifies what a concept is, then concepts are just compile-time boolean expressions on types—with hyper-metaprogramming support, they can easily be implemented in Sparrow. This is the most general form of a predicate. If we think of concepts as sets of types, then we can find a compile-time boolean function that can tell us whether a type is contained in the set of types represented by the concept.

In a language such as C++, we have abstractions for detecting if a particular type contains specified variables, functions, or types. Although Sparrow can easily provide such support (using Compiler API—see chapter 11), we preferred to use the duck-typing paradigm for concepts. Instead of testing if a type contains a function with a specific functionality, we test whether we can call that function with some parameters on that type.

Let us take a simple example. We want to define a concept that is modeled by all types that have standard construction and destruction, copy construction, and equality defined. In Sparrow we would write:

```
concept Regular(x)
    if isValid(x.ctor())         // Default construction
    && isValid(x.ctor(x))        // Copy construction
    && isValid(x.dtor())         // Destruction
    && typeOf(x == x) == Bool    // Equality is defined
    ;
```

For concept checking, it is not required for methods to be explicitly defined. Tests are performed to simply determine whether they can be called in the specified ways. For example, the following type models the `Regular` concept, even though it does not have a copy constructor:

```
class[noDefault] NullType {
    fun ctor {}
    fun[convert] ctor(other: AnyType) {}
    fun dtor {}
    fun ==(other: NullType) = true;
    fun ==(other: AnyType) = false;
}
```

This class does not define a proper copy constructor. Still, an object of this type can be constructed from another object of the same type, and therefore it satisfies the `Regular` concept. Although one might argue that it is safer to check the full signature of the members, we believe that this approach is more flexible. The user of the concept does not really care how a particular class was defined, the most important part is how a particular class can be used.

We can do this with variables and type definitions inside classes. The following listing exemplifies this:

```
concept TestVarType(x)
    if isValid(x.varName1 = 5)     // Testing a var by assignment
    && typeOf(x.varName2) == Int   // Testing a var by type
    && typeOf(x.typeName) == Type // Testing for type
```

7.5.3 Concepts and axioms

Sutton and Stroustrup's view on concepts are that they should be "constraints + axioms" [SS12b]. We have shown that Sparrow supports the most general method of specifying constraints (using compile-time expressions), but we did not introduce any special form for expressing axioms.

Before discussing how axioms can be encoded in Sparrow, we need to mention how they are proposed to be handled in the upcoming C++ standard. In Sutton and Stroustrup's words: [SS12b]

> Axioms state semantic requirements on types that should not be statically evaluated. An axiom is an invariant that is assumed to hold (as opposed required to be checked) for types that meet a concept.

As the compiler is not required to do anything for axioms, we do not have any constraints on how to introduce them into the language. Therefore, we only need to invent a convention for expressing axioms. Listing 7.6 presents a possible implementation of concepts with axioms.

Note that in order to support axioms we did not change the syntax or the semantics of the language in any way, and we defined axioms with the same expressiveness as in C++. Sparrow is powerful enough to let the user define such abstractions. If at some point we would like to implement inside the compiler semantic reasoning on top of axioms (e.g., special optimizations), we can easily interpret these axioms.

Using a library similar to Haskell's QuickCheck [CH11] one can create unit-tests that will actually check if the axioms are fulfilled for the user's types.

Yet another feature that shows that simple language rules such as the ones in Sparrow can increase flexibility in the programming language.

7.5.4 Compile-time generics

The way the type system is designed, there is a problem with creating functions that operate on types. Using a `Type` parameter in a compile-time function yields an error. This is because the body is compiled before any value is bound to the parameter, and compiling any structure that uses the parameter will try to evaluate the type stored in the parameter.

To solve this problem, we can use the instantiation mechanism of generics. We cannot use generics directly because compile-time functions cannot

```
concept Ordered(x)
    if typeOf(x < x) == Bool   // Main ordering operator
    && typeOf(x > x) == Bool   // Other operators can be inferred
    && typeOf(x <= x) == Bool
    && typeOf(x >= x) == Bool
    && axiom("lessIsOrdered")
    && axiom("greater")
    && axiom("lessEqual")
    && axiom("greaterEqual")
    ;
fun[ct] axiom(name: StringRef) = true; // No checks performed
fun[rtct] => (premise, conclusion: Bool) = !(premise) || conclusion;
fun lessIsOrdered(x, y, z: Ordered)
         = !(x < x)
        && ( (x<y) => !(y<x) )
        && ( (x<y) => !(y<x) )
        && ( (x<y && y<z) => (x<z) )
        && ( x<y || y<x || x==y )
        ;
fun greater(x,y: Ordered)      = (x>y) == (y<x);
fun lessEqual(x,y: Ordered)    = (x<=y) == !(y<x);
fun greaterEqual(x,y: Ordered) = (x>=y) == !(x<y);
```

Listing 7.6: Defining concepts with axioms

be generics (there is no good way of distinguishing between generic parameters and non-generic parameters). Therefore we created a small extension to generics for handling this case: *compile-time generics*. These are generics that only have meaning at compile-time and all of their parameters are generic parameters. With this mechanism we bind the values of the parameters before instantiating the generic, before compiling its body, making it possible to create functions that operate on types.

The following listing provides some examples of compile-time generics defined in Sparrow:

```
fun[ctGeneric] isRef(t: Type)
                = 0<numRef(t);
fun[ctGeneric] addRef(t: Type)
                = changeRefCount(t, numRef(t)+1);
fun[ctGeneric] removeRef(t: Type)
                = ife(isRef(t), changeRefCount(t, numRef(t)-1), t);
fun[ctGeneric] removeAllRef(t: Type)
                = changeRefCount(t, 0);
fun[ctGeneric] atLeastOneRef(t: Type): Type
                = ife(isRef(t), t, addRef(t));
fun[ctGeneric] pre_-@ (t: Type)
                = removeRef(t);
```

```
fun[ctGeneric] pre_!@ (t: Type)
                    = atLeastOneRef(t);
```

This is a flexible mechanism that can be used to create type operations. The last two operator definitions are used frequently in the Sparrow standard library to remove the references of generic types or to ensure that we have at least one reference to a type.

The `!@` operator was exemplified in listing 7.1 while presenting a possible `Vector` implementation; the following code exemplifies the use of the `-@` operator:

```
fun makePair(v1, v2: @AnyType)
    = Pair(-@typeOf(v1), -@typeOf(v2))(v1, v2);
```

7.5.5 Conditional compilation

One Sparrow feature that can be very useful in conjunction with generic programming is conditional compilation—the ability to include or exclude code based on some conditions known at compile-time. In Sparrow this can be achieved using an **`if`**`[ct]` structure, very similar to a classic **`if`** statement:

```
if[ct] ( ct-condition ) action1 else action2
```

When the compiler identifies this structure it compile-time evaluates the condition. If the condition evaluates to **`true`**, the whole structure is replaced with the first action; otherwise the second action is used (see also chapter 5). We have already seen an example of using this structure in section 7.4, while defining the `FlushHelperClass` class.

Similar to the compile-time **`if`** structure, Sparrow also provides compile-time versions of **`while`** and **`for`** statements. The bodies of these statements are repeated in the final program as long as the condition is **`true`**, or the iteration is running. An example of using the compile-time **`for`** structure is shown below:

```
for[ct] ( x = 1..5 )
    cout << "Printing instance " << ctEval(x) << endl;
// Equivalent with the following code:
// {
//      cout << "Printing instance " << 1 << endl;
//      cout << "Printing instance " << 2 << endl;
//      cout << "Printing instance " << 3 << endl;
//      cout << "Printing instance " << 4 << endl;
//      cout << "Printing instance " << 5 << endl;
// }
```

More examples of using this compile-time **`for`** statement can be found in chapters 9 and 10.

From the point of view of the compiler, there are only three structures (`if`, `while`, and `for`) that can be used in conjunction with the `[ct]` modifier. The modifiers prove to be a flexible mechanism for extending basic language features.

7.6 Evaluation

7.6.1 Comparison with other languages

Following Siek's work [Sie05; GJL+03] we want to compare Sparrow's ability for generic programming with other programming languages. The results are shown in table 7.1.

The following describes the comparison criteria, focusing on how Sparrow fulfills them.

Multi-type concepts refers to the ability of a concept to be parameterized on more than one type. Currently Sparrow does not support this feature, although it is planned for future addition.

Multiple constraints refer to the ability to place multiple constraints on a single generic parameter; for example, one may want to constrain a type to be both a `Range` and a `Swappable`. This feature is supported by Sparrow as the language supports arbitrarily complex expressions in `if` clauses and concept definitions.

Associated type access refers to the possibility of directly defining inner, associated types in generics, and accessing them from the outside. Sparrow allows the definition of these types via the `using` construct, and they can be accessed using the dot notation on a member of the type, or from the type object itself. Listing 7.1 provides examples on how one may introduce associated types with a `Vector` generic class.

Constraints on associated types can be easily set in Sparrow as part of `if` clauses and concept definitions, as Sparrow allows the user to put constraints on any observables that can be associated with a type.

Retroactive modeling denotes the ability to specify "type models concept" relations after the type and the concept are defined. Currently this feature is not supported in Sparrow.

Type aliases refer to the possibility of creating shorter names for types. This is available in Sparrow with the help of the `using` construct, as shown in listing 7.1.

Separate compilation indicates that generics are compiled (semantically checked) independently of their use; as opposed to the template instantiation mechanism from C++. Currently, Sparrow does not support separate compilation, as it would add overhead to the generated run-time code. We plan to add a form of separate compilation that would not decrease the efficiency.

Implicit instantiation refers to automatically deducing the types of the arguments of generics, even if the generic arguments are not explicitly provided. Sparrow supports implicit instantiation. As discussed in this section 7.2.5, the implicit instantiation rules in Sparrow are so simple that the user hardly needs to think about them.

Concept dispatching represents the ability of specializing generic functions based on the concepts that are modeled by the input arguments. Sparrow provides this feature, as presented in section 7.3.1.

No run-time overhead refers to the absence of overheads related to generic programming features in the generated run-time code; if there is no run-time overhead, a generic functions should have the same generated instructions as a non-generic function with the exact same body, but without any generic parameters. Following the C++ method of ad-hoc polymorphism, Sparrow supports no run-time overhead.

Constraints on type usage refer to the ability to impose constraints on usage patterns of a type rather than its structure, and to differentiate (specialize) generics based on this aspect. Not only is this supported, it actually appears to be the preferred method of specializing in Sparrow.

Specialization on user constants denotes the ability of specializing generics on user-defined compile-time constants (numbers, strings, or other data structures). Due to its hyper-metaprogramming abilities, Sparrow allows generic parameters of any compile-time type, and allows the user to specialize on these parameters.

Arbitrarily complex conditions refer to the complexity of the conditions that can be expressed for generic parameters—values, types, type properties, etc. In Sparrow, one can add constraints on any information that is available at compile-time on a generic parameter.

7.6.2 Conclusions

We presented in this case study how the generics feature works in Sparrow. We started from a small number of design principles, and then showed that

	C++	SML	Haskell	Java	C#	$\mathcal{G}$	Sparrow
Multi-type concepts	○	●	●	○	○	●	○
Multiple constraints	○	◐	●	●	●	●	●
Associated type access	●	●	◐	◐	◐	●	●
Constraints on assoc. types	○	●	●	◐	◐	●	●
Retroactive modeling	○	●	●	○	○	●	○
Type aliases	●	●	●	○	○	●	●
Separate compilation	○	●	●	●	●	●	○
Implicit instantiation	●	○	●	●	●	●	●
Concept dispatching	●	○	○	○	○	◐	●
No run-time overhead	●	○	○	○	○	○	●
Constraints on type usage	◐	○	○	○	○	○	●
Spec. on user constants	◐	○	○	○	○	○	●
Arbitrarily complex cond.	◐	○	○	○	○	○	●

Table 7.1: Comparison between programming languages with respect to generic programming.

these principles, without any complex rules, can generate a complete generic programming environment in Sparrow.

We have shown that generic programming in Sparrow is complex enough to model structures that are present in other similar languages with support for generic programming (e.g., C++, D, $\mathcal{G}$).

A unified list of parameters is shown to be a viable alternative to the way languages like C++ implement generic parameters. Not only is the syntax simplified, but so are the rules required for type inference. A typical user that reads the code barely needs to know that there are some underlying rules governing how values are bound to generics.

The `if` clauses provide a simple, yet powerful method of imposing constraints over generics. One can use them to impose type-safety constraints, to specialize some functionality for efficiency, or simply to limit the functionality of some generics. This feature is heavily based on the hyper-metaprogramming support that Sparrow offers; this way, by using hyper-metaprogramming for multiple language features, the language becomes more coherent, and therefore more natural.

We provided a simple implementation of the *concepts* feature, which has received a lot of attention in the C++ community in recent years [SR03; JWL03; DS06; GSJR06; Got06; TJ07; SS12a; SSR13]. Not only have we shown that it can be implemented in Sparrow, we have also shown that an implementation based on hyper-metaprogramming is simple and does not involve new language abstractions. The user can learn with very little effort all there is to know about concepts in a short time.

Overall, compared with the support for generic programming in other languages, Sparrow appears to offer many of the important language features that make generic programming usable in practice, and with only a small number of design principles and rules. This constitutes a perfect example of the *growing a language* principle[Ste99].

In conclusion, Sparrow is flexible enough to let the user define complex data structures and algorithms in a generic way, such that both naturalness and efficiency are maintained.

Some people forget it, but efficiency is always a (or the) main constraint in the design of any software.

R. Ierusalimschy

CHAPTER 8

General efficiency analysis for Sparrow

Sparrow aims to be a high-level language, but at the same time allow the users to create efficient programs. Users should express the program in a way that is closer to the way of their thinking and the compiler should be able to transform this code into efficient machine code.

In this chapter we want to test this hypothesis. We present four case studies, with different degrees of complexity. The first case study tests the performance of basic `for` loops, as they constitute a basis for complex algorithms. The next two case studies test the performance of two problems that can easily be expressed with ranges, comparing them to a hand-made implementation in C++. Finally, we present a case study on a complex problem (the Havlak loop recognition algorithm), comparing Sparrow with C++, Go, Java, and Scala on execution time, memory, code size, binary size, and compilation time—the case study is based on a benchmark proposed by Hundt [Hun11].

We show in this section that Sparrow allows writing efficient code (similar efficiency to C++) in a natural manner. A basic meaning of *natural* is less code than in other languages like C++ and Scala. Moreover, the term also involves the ability of the programmer to directly express the concepts used in the problem description.

For Sparrow to be in the same efficiency class as C++ (as required by the language design), the Sparrow code generator must be able to produce executable code that is similar to the one generated by a C++ compiler. The language must support the same basic abstractions (stack and heap variables, pointers, increased control over the memory layout, functions, basic control structures, etc.) and, moreover, adding abstractions on top of these must not incur unwanted costs. If this is the case, then an efficient code written in C++ can also be written in Sparrow along the same lines, and it will be in the same class of performance. On top of that, Sparrow encourages using certain types of abstractions that increase the naturalness without introducing execution penalties. Ranges are probably the best example in this respect.

In Sparrow, a range is an abstraction over a collection of elements (not necessarily stored in a container) that can be traversed in a particular order. The range class actually defines how this traversal must be done to actually access the elements. Because ranges are heavily utilized, they need to be very fast, similar to iteration code written by hand. The case studies in this section test the performance of ranges.

If the primitives are fast and one can create abstractions on top of these primitives without incurring run-time costs, then Sparrow programs will be generally fast.

8.1 Efficiency of the `for` structures

To iterate over the range of natural numbers from 0 to 100 (right-open interval), in C++ one would traditionally write something similar to the following code[1]:

```
for ( int i=0; i<100; ++i ) {
    cout << i << endl;
}
```

The above can be transformed into a code that uses a `while` loop:

```
int i=0;
while ( i<100 ) {
    cout << i << endl;
    ++i;
}
```

The main difference between the two versions is how the `break` and `continue` statements are handled; if they appear in a `while` statement the step instruction is not executed properly.

Still, from the perspective of the user, the two structures are quite similar. In both cases the user needs to specify an initial state of the iteration, a termination condition, and a step operation. None of these three operating concepts are directly related to the goal of *iterating over the range of natural*

[1]the 2011 version of the C++ standard [Cpp11; Str13] introduced the so called *range-based for loops*; they provide an easier way of expressing `for` loops, similar to what we have in Sparrow. However, the range support in C++11 is not as evolved as in Sparrow; for example there is no standard way to use the range-based for loop to iterate over the range of natural number from 0 to 100.

numbers from 0 to 100, they are instead low-level primitives. This can make the iteration structure less natural to the programmer.

In Sparrow, the same iteration is written as:

```
for ( i = 0..100 ) {
    cout << i << endl;
}
```

Here, the goal is directly translated into code: *iterate over* translates into **for** structure and *range of natural numbers from 0 to 100* translates into `0..100`. There are no low-level primitives for a simple iteration. Allowing the user to express its objectives directly into code increases the naturalness of the programming language. To this end, the **for** structure is just a trivial example.

In general, adding more abstraction layers to a system can decrease its efficiency. We must check if writing **for** structures in this manner is slower than writing them as in C++. We will analyze the general case, and conduct some performance tests to ensure proper efficiency.

A **for** `( <name>: <type> = <range> ) <action>` construct is transformed into a corresponding **while** loop (see chapter 5):

```
var rangeVar: Range = <range>;
while ( ! rangeVar.isEmpty() ; rangeVar.popFront() ) {
    var <name>: <type> = rangeVar.front();
    action;
}
```

If `<type>` is not present, the type `rangeType.RetType` is assumed (every range is supposed to define a type named `RetType`). If the user passes an expression that is not a `Range`, the first line yields an error. In Sparrow, the syntax of the **while** loop is extended to support an additional *step* action; this makes it similar to the **for** structure from C++.

In our case, by replacing the range with `1..100`, and then expanding the `isEmpty`, `popFront` and `front` methods, we obtain:

```
var rangeBegin = 0;
var rangeEnd = 100;
while ( ! (rangeBegin >= rangeEnd) ; ++rangeBegin ) {
    var i: Int = rangeBegin;
    cout << i << endl;
}
```

In terms of execution, this is very similar to the standard usage of **for** in C++. At least at a first glance, the Sparrow abstraction does not introduce any performance penalty.

Note that the Sparrow compiler relies on function inlining to avoid performance penalties. If one had to pay a performance cost for any trivial function call, then the performance of **for** structures would be affected.

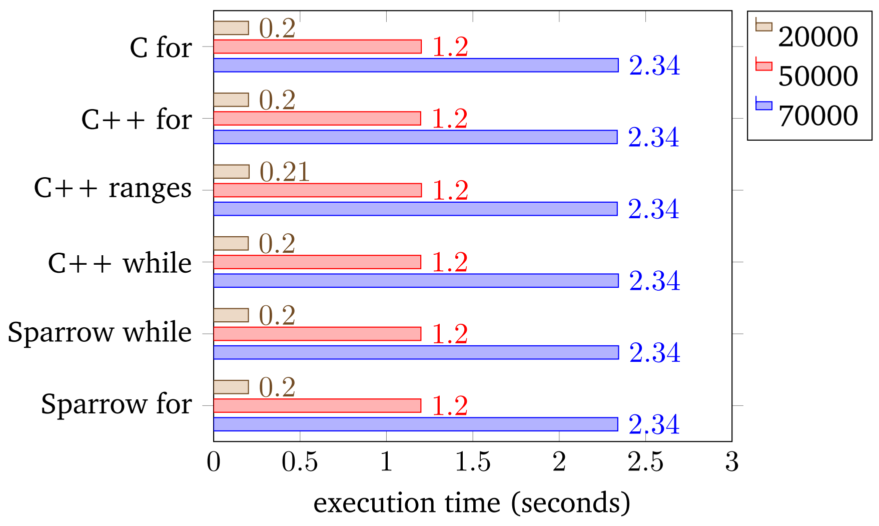


Figure 8.1: Performance of Sparrow **for** construct, compared with the performance of **for** and **while** in C++ and C, for three different values of parameter n. *C++ ranges* refers to a C++ program that uses the same range abstractions as Sparrow

To analyze the performance of the **for** structure in Sparrow, we use the following program:

```
var res: ULong = 0;
for ( i = 0..n )
    for ( j = 0..n )
        res += i*j;
```

If there is a penalty involved in the **for** loop, due to the quadratic nature of the algorithm, the problem will be reflected in the execution time for large values of n. We compared this program to similar versions written in C and C++; for completeness, we also compared variants of this program that contain **while** structures instead of **for** loops. Moreover we implemented a basic range solution in C++, very similar to the one in Sparrow. The details about the machine and the compilers used for this experiment can be found in Section 8.4.2.

The timing results for these programs are shown in fig. 8.1. All the versions practically yield the same results. This shows that Sparrow's **for** instruction is as fast as the **for** from C or C++. There is no performance penalty involved in the use of the **for** structure in Sparrow.

```
1  fun fib(n: UInt): UInt {
2      var p: UInt*UInt = 0 ~ 1;    // Last 2 entries
3      for ( i = 0..n )
4          p = p._2 ~ (p._1 + p._2);
5      return p._2;
6  }
7  fun isOdd x = x%2 != 0;
8  fun sqr n = n*n;
9  fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));
10
11 var res = 1...n map \fib filter \isOdd map \sqr sum;
```

Listing 8.1: Range manipulation with map and filter in Sparrow

8.2 Performance of more complex range operations

Having a **for** structure that is as fast as the one in C++ does not guarantee that more complex algorithms are not affected by performance penalties introduced by the language abstractions. This section analyzes more complex structures and algorithms, showing that Sparrow is not slower than C++, even though it uses higher abstractions.

One way of adding more complexity to the basic ranges is to use common operations like `map` and `filter` (presented in the previous chapter). To check the performance of these range operations we want to solve the following problem:

> Compute the sum of squares of all the odd numbers belonging to the first *n* Fibonacci numbers.

The Sparrow code that does this (including the auxiliary functions) is presented in listing 8.1.

The `sqr` and `sum` functions are implemented in the standard library, but we present them for completeness. Other than that, there is a function that computes the Fibonacci number with the given index, a function that tests whether a number is odd, and the actual computation required for the problem.

The reader can verify that we are using exactly the same concepts that are used in the problem description, but in reverse order: the first n Fibonacci numbers are represented by the `1...n` range mapped to corresponding Fibonacci numbers, then we filter for odd numbers, we square the results, and accumulate the sum of the resulting values. We did not use any other concepts in specifying the computation; consequently our computation is shorter than its description in plain English (ignoring the auxiliary functions).

Let us analyze the concepts expressed in the range operations from line 11. The `sum` operation is executed last over a range obtained by composing multiple ranges. For each range in this expression (numeric range, filtered range, and mapped range) the library offers 3 functions: `isEmpty`, `front`, and `popFront` which can be implemented in simple and efficient manners.

The `1...n` range is a basic range that gets expanded to an iteration similar to the one from languages like C++; we have shown already that this is as fast as it can be.

A mapped range will call the given function whenever the result is required. Special care is taken to call this function only once for each particular value, even if the caller invokes the `front` multiple times for the same value; there is some simple bookkeeping involved, but this is optimized away by the compiler. This means that the `fib` and the `sqr` functions will be called exactly `n` times.

A filtered range will augment the `popFront` function to skip the values that do not satisfy the given condition. This will invoke the given predicate (`isOdd`) for each value from the source range. This means that the `isOdd` function will also be called exactly `n` times for our computation.

The `foldLeft` operation inside the `sum` function will expand to a summation of all the elements from the given range, with no extra operations involved. This is very similar to the `accumulate` standard function from C++ [Cpp11; Jos12].

As the compiler is able to inline all the function calls and remove the bookkeeping involved with the mapped ranges, the generated code will be similar to the best implementation that can be done in languages like C++.

Listing 8.2 shows how the same algorithm would be written in C++. The computation is implemented in terms of lower-level primitives, and the main concepts present in the problem description do not appear in the code directly. There is only the invocation of Fibonacci number computation function that appears directly in the C++ code, but this is buried inside low-level logic that makes it hard for the reader to understand at a first glance what is going on[2].

We set out to measure the performance of the Sparrow program from listing 8.1, comparing it with the performance of the C++ program from listing 8.2. In addition, we also tested a version of the same algorithm implemented in C, and a version implemented in Sparrow that follows the same low-level coding we used in the C++ version. Again, the details about the machine and the compilers used for this experiment can be found in Section 8.4.2.

The results can be found in fig. 8.2. Again, the Sparrow execution time is equal to the execution times of the same program written in C++ and C. Sparrow allows expressing code in a concise and more natural manner, while having the same performance characteristics as C and C++.

[2]of course, programmers accustomed to C++ could decipher this code easily, but for more complex problems, it is likely that the C++ code may become harder to read

```cpp
int fib(int n) {
    int a = 0;
    int b = 1;
    for ( int i=0; i<n; ++i ) {
        int aOld = a;
        a = b;
        b += aOld;
    }
    return b;
}

int res = 0;
for ( int i=1; i<=n; ++i ) {
    int f = fib(i);
    if ( f % 2 != 0 ) {
        res += f*f;
    }
}
```

Listing 8.2: C++ equivalent of the code presented in listing 8.1

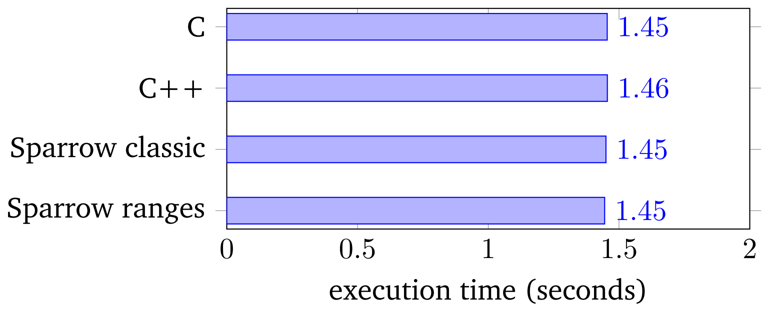


Figure 8.2: Execution time for the Fibonacci ranges problem (n=70000)

8.3 Lazy evaluation for ranges

In Sparrow, range operations use lazy evaluation; in this section, we illustrate how this design choice enables the generation of efficient code. For this we try to solve the following problem:

> What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

A Collatz sequence is a sequence generated from a starting value by repeatedly applying a certain transformation, and ending when the value 1 is reached. The transformation used is the following: if the number is even,

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
    (fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize takeWhile (fun s
    = s < 500) rootMeanSquare;
```

Listing 8.3: Computing root mean square of the first Collatz sequences; using infinite ranges

divide it by two; otherwise multiply it by three and add 1. For example, the Collatz sequence that starts with the number 13 is: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. This sequence has length 10.

The code describing the problem is rather compact: see listing 8.3. There is one simple function `nextCollatz` that, given a number, returns the next number in the Collatz sequence. There is another function `collatzSeq` that, given a number, returns a range with all the numbers in the corresponding Collatz sequence. To generate this range, we create a range that applies successively the `nextCollatz` function, computing values until `1` is encountered.

The `rootMeanSquare` function is implemented in the standard library; for completeness, we provide it below:

```
fun mean(r: Range): Double if Number(r.front()) {
    var sum: r.RetType = 0;
    var count = 0;
    for ( x = r ) {
        sum += x;
        ++count;
    }
    return Double(sum) / Double(count);
}
fun rootMeanSquare r = Math.sqrt(mean(r map \sqr));
```

The astute reader will immediately notice that we are using infinite ranges in this example; the `(1..)` expression creates an infinite range that starts with 1[3]. Also, the `generate1` function will create an infinite range. By design, Sparrow ranges are lazy, so the program will not loop indefinitely.

By using the `takeUntil` and `takeWhile` functions we can limit a range based on a predicate. This way, we reduce our ranges from infinite ranges to finite ranges[4].

If one understands the way these operate, then the actual computation associated with this problem is straightforward: we have an infinite range of all

[3]actually, the range will be bounded by the largest integer possible on the user machine; still, generalizing, because of the large end value, we can talk about infinite ranges

[4]we take for granted the fact that for any number `n` the associated Collatz sequence is finite

```
unsigned long long nextCollatz(unsigned long long n) {
    return n%2==0 ? n/2 : n*3+1;
}
unsigned int collatzLen(unsigned long long n) {
    unsigned int len = 1;
    while ( n > 1 ) {
        n = nextCollatz(n);
        ++len;
    }
    return len;
}

unsigned long long sum = 0;
int count = 0;
for ( unsigned int i=1; ; ++i ) {
    unsigned int len = collatzLen(i);
    if ( len >= n )
        break;
    // Main part of computing the root mean square
    sum += len*len;
    ++count;
}
double res = sqrt(double(sum) / double(count));
```

Listing 8.4: C++ equivalent of the code presented in listing 8.3

positive integers, we generate from it all the corresponding Collatz sequences, and for each such sequence we compute its length and we take values while the length is smaller than 500; for the resulting lengths we compute the root mean square.

Listing 8.4 presents the same algorithm as it would have been written in C++.

One issue with the C++ code is that the problem domain terms (like Collatz sequences and series of Collatz sequences) are obfuscated in the algorithm. Given the current problem, one cannot reuse part of the implementation for solving slightly related problems (e.g., what is the sum of the lengths of the selected Collatz sequences?). Moreover, in C++ it is not easy to extract the root-mean-square algorithm and place it in an external library so that it is easily usable[5]. This predicament usually leads to so-called *copy and paste programming* and increased code complexity.

We set up an experiment to determine the difference in performance of our Sparrow program compared to the C++ version. Besides the Sparrow ranges

[5]one would attempt to use iterators for this, but then it would need to encode the whole input as a pair of iterators—at this point it would be much harder to express the problem in these terms

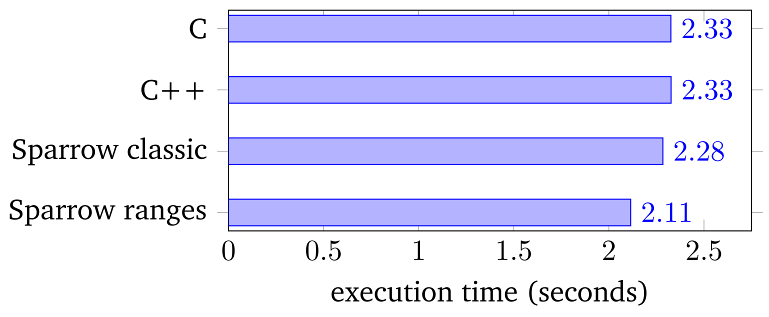


Figure 8.3: Execution time for the Collatz sequences problem

algorithm and the C++ algorithm, we also include in our test a version of the program written in C, and a version written in Sparrow, but using similar abstractions as in C++. The details about the machine and the compilers used for this experiment too can be found in Section 8.4.2.

The results of the measurements can be seen in fig. 8.3. Not only does the Sparrow algorithm compute the result in a finite amount of time, it also has the same performance characteristics as the C++ version. In fact, looking at the figure, the Sparrow ranges version is slightly faster than the other programs; this is due to the fact that, by arranging code in this manner, the optimizer is able to better identify possible optimizations. This is a local optimization improvement, and the increase in performance should not be generalized.

These measurements confirm once more that in Sparrow one can write code in a more natural manner, and that the code is translated into efficient machine code by the compiler.

8.4 Loop recognition in Sparrow

In this section we discuss the Sparrow implementation for the benchmark proposed by Robert Hundt [Hun11]. The benchmark involves a well defined algorithm (Havlak's loop recognition algorithm) to be implemented in several languages using the language idiomatic containers, looping constructs, and object allocation schemes. Implementing the algorithm in the same manner in different languages provides a fair comparison. In his paper Hundt considers the languages C++, Java, Go and Scala. We add Sparrow as another programming language to this benchmark.

The Sparrow version of the algorithm follows the specifics that are found in the other versions. As the benchmark requires, we do not attempt to use language-specific constructs to make the algorithm faster. We do not to rewrite parts of the algorithm to better fit Sparrow's special features. There are no special tweaks applied to make the Sparrow version faster.

8.4.1 Implementation notes

We present below the Sparrow features that were used in implementing this benchmark, discussing them on the same lines as Hundt in his article[Hun11]. The full Sparrow code for this benchmark can be found in appendix B.

Data structures

The data structures used in the Sparrow version are very similar to the ones that are used in the C++ version. There is however one notable exception: instead of using binary trees for maps and sets, we used hash tables. This is because the default maps and sets in Sparrow are based on hash tables; moreover, the Sparrow standard library does not implement maps and sets using binary trees at all. The main container types used in the algorithm are presented in the following code:

```
using NodeVector = UnionFindNode Vector;
using BasicBlockMap = Map(BasicBlock Ptr, Int);
using IntList = Int List;
using IntSet = Int Set;
using NodeList = (UnionFindNode Ptr) List;
using IntListVector = IntList Vector;
using IntSetVector = IntSet Vector;
using IntVector = Int Vector;
using CharVector = Byte Vector;
```

These `using` directives are similar to `typedef`s in C++. We used the operator notations as the code looks much cleaner without too many parenthesis—`Int List` is equivalent to `List(Int)`, where `List` is a class generic that we are instantiating with an `Int` parameter.

These container types can be optimized further, but we used the same containers as the C++ version to make a fair comparison.

Enumerations

The current version of Sparrow does not directly support enumerations. Therefore, we used name aliases for constants, as shown in the code below:

```
using BB_TOP             = Byte(0);   // uninitialized
using BB_NONHEADER       = Byte(1);   // a regular BB
using BB_REDUCIBLE       = Byte(2);   // reducible loop
using BB_SELF            = Byte(3);   // single BB loop
using BB_IRREDUCIBLE     = Byte(4);   // irreducible loop
using BB_DEAD            = Byte(5);   // a dead BB
using BB_LAST            = Byte(6);   // Sentinel
```

Iterating over data structures

In our implementation we used Sparrow's standard range iterations for most of the loop iterations, when this would not change the structure of the algorithm. In some places we changed the helper classes to return ranges instead of getting the underlying containers. The code below shows examples of such loops:

```
// Step a:
for ( bb = cfg.basicBlocksRange )
    number.insert(bb, kUnvisited);
...
// Step d:
for ( v = backPreds(w).all ) {...}
...
```

Type inference

Similar to Scala, Sparrow supports a basic form of type inference. This is mostly visible when declaring variables; the programmer does not need to specify the type of the variable, as it can be deduced from the variable initializer. The syntax of the variable declaration is just like in Scala[6]:

```
var lastId = current;
```

Member functions

Sparrow allows member functions to be declared inside classes, similar to C++. The constructors and destructors need to be declared manually (but with the fixed names `ctor` and `dtor`) as in C++ and unlike Scala. Just like in C++ member functions do not add any performance penalty. Similar to Java, in Sparrow there is no distinction between function declarations and function definitions; all member functions need to be defined inside the class. Here is the definition of `BasicBlockEdge`:

```
class BasicBlockEdge {
    fun ctor(cfg: @MaoCFG, fromName, toName: Int) {
        from ctor cfg.createNode(fromName);
        to   ctor cfg.createNode(toName);
        from->addOutEdge(to);
        to->addInEdge(from);
        cfg.addEdge(this);
    }
    var from, to: BasicBlock Ptr;
}
```

[6]type inference in Sparrow is inspired by Scala's type inference

Unlike the other languages, in Sparrow it is not common to define accessor methods for the inner members. Besides the fact that there is less code, the main reasons for this are the following:

1. Currently in Sparrow there is no `const` functionality; therefore, any variable that would be returned by a const reference in C++, would be returned as a normal reference in Sparrow. Having a reference to a variable is as unsafe as allowing direct access to that variable. Not yet having `const` in the language is a known language defect and will be addressed in the future.
2. When calling a function that does not require any parameters, the parentheses are not required as in C++. For example `lsg.killAll;` is a perfectly valid function call if `killAll` is a function without parameters. This means that, if we start the implementation by exposing the variables directly, one can afterwards encapsulate the variables by creating a function with the same name, without changing the callers.

8.4.2 Performance analysis

We used the same execution model that Hundt used in his paper [Hun11]. There is a small driver that is able to construct simple control flow graphs and then run the algorithms on them. In the first stage the driver constructs a simple control flow graph in the shape of a diamond with 4 nodes with a back-edge from the bottom to the top node; the loop recognition algorithm is run on this 15,000 times. In the next stage the driver constructs a large control flow graph containing 4-deep loop nests with a total of 76,000 loops; the loop recognition algorithm is run on this 50 times.

All the programs were run 7 times; the first round of execution was ignored. We measured both the CPU execution time and the memory usage (maximum resident set size). The values were then averaged, and the standard deviation was computed (just to make sure the resulting averages do not contain significant errors).

All the experiments were performed on a MacBook Pro Retina (late 2013), 2GHz Intel Core i7, 8GB RAM, Flash Storage, MacOS X 10.10.1. The C++ version was compiled with Apple's Clang version 6.0 (clang-600.0.56 - based on LLVM 3.5svn). The Go version was compiled with go version go1.3.1 darwin/amd64. The Java version was compiled with javac 1.8.0_25 and run with the `-server -Xss15500k` flags. The Scala version was compiled with Fast Scala compiler version 2.11.2 and run with the `-XX:+UseCompressedOops -server -Xss15500k` flags. The Sparrow version uses the latest development version (0.9.3) of the Sparrow reference compiler, using LLVM 3.5.

Execution time

The execution time measurements are presented in fig. 8.4.

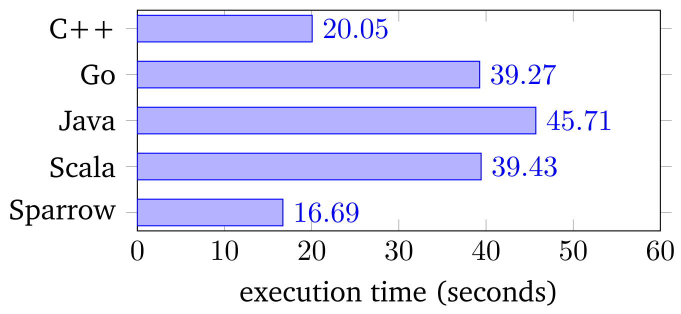


Figure 8.4: Execution time for the Loop recognition algorithm in various languages

As can be seen from the figure, the algorithm written in Sparrow is the fastest. The main reason for that is the use of hash tables to implement the standard maps instead of balanced trees (which are the norm in C++). Even if we try to improve the C++ implementation to use `unsorted_map` and `unsorted_set` (which rely on hash tables) the performance of the C++ program is still worse than the one in Sparrow.

We can conclude that Sparrow programs are in the same class of efficiency as programs written in C++ (even if for this test the Sparrow version is faster than the one from C++).

Code size

The code size can be one of the metrics used to measure the naturalness of the code. It is often the case that the smaller the code, the easier it is for the programmer to understand it. To measure the code size, we used the cloc tool [CLOC] version 1.62.pl. This counts the lines of code, ignoring comments and blank lines. The results are presented in the first row of table 8.1. The results are normalized based on the Scala version, as Scala is a language in which a lot of boilerplate code is reduced [Ode14; Ode11].

The Sparrow version is best in this category too. The required Sparrow code is smaller even than the Scala code. This means that Sparrow programs can be not only efficient, but also concise, letting the programmer focus only on the important aspects of the code.

Binary size

The size of the binary is also an important aspect of a program, especially in the modern trend of going mobile. The Sparrow version also performs well in this category, with an executable size that is 16% smaller than the C++ executable size. The only program that is smaller than the Sparrow program is the Java program. This is because for Java and Scala we use the dimensions

		C++	Go	Java	Scala	Sparrow
Code size	LOC	491	531	590	370	346
	norm	1.33 x	1.44 x	1.59 x	1.00 x	0.94 x
Binary size	bytes	25 248	1 955 040	13 248	48 246	21 416
	norm	1.00 x	77.43 x	0.52 x	1.91 x	0.84 x
Memory footprint	MB	165	450	983	699	195
	norm	1.00 x	2.73 x	5.96 x	4.24 x	1.18 x
Compilation time	sec	1.201	0.223	1.427	5.375	3.916
	norm	1.00 x	0.19 x	1.19 x	4.47 x	3.26 x

Table 8.1: Comparison between different versions of the loop recognition algorithm in terms of code size, binary size, compilation time, and memory footprint

of the `.jar` files, which are compressed forms of binaries (a zip algorithm is applied to the program byte code).

Memory footprint

It should not be necessary to convince the reader that the memory footprint is also an important characteristic of a program. We measured the maximum resident sets of the programs, and the results are also presented in table 8.1. Sparrow also performs well in this category, consuming only 18% more memory than the C++ version. This is also a consequence of using different default containers for maps and sets.

Compilation time

At this category Sparrow does not perform that well. The program compiles 2.66 times slower than the analogous C++ program. The main reason behind this is that the Sparrow compiler is not a production compiler, and it was not properly optimized. Still, the Scala compiler performed worse than the Sparrow compiler.

This benchmark shows that, for a relatively complex algorithm that intensively uses language default containers, loop constructs, and memory allocations, Sparrow is a language that is able to produce efficient binaries.

The Sparrow algorithm is faster than the one written in C++, the memory footprint is comparable to the C++ version, and the binary size is also smaller than the C++ program. All these benefits are provided while the number of lines of code required is smaller than for any other language in our benchmark. The only aspect where Sparrow falls short is the compilation time.

8.5 Conclusion

In this chapter we presented four case studies that assess the efficiency of Sparrow programs. We started from the study of the `for` structure performance, as the basic block of building algorithms. The Sparrow `for` loop is less verbose than a traditional C++ loop, more flexible, and has the same performance characteristics as the C++ `for` loop.

In the second case study we have shown that complex range operations can be easily expressed in Sparrow using range constructs and the infix/postfix operator syntax. If the problem can be formulated in terms of ranges, then one can directly express all of the key problem concepts directly in Sparrow code. Sometimes the problem solution in Sparrow can be even less verbose than the description of the problem in English. The performance of the Sparrow programs is similar to the performance of the C++ programs.

The third case study emphasizes an important property of Sparrow ranges: laziness. The given problem contains infinite ranges that, when properly assembled with range manipulation primitives, concisely represent the problem description, avoiding endless loops. Again, the performance of the Sparrow program is virtually identical to the performance of the C++ program.

The last case study aims to put the Sparrow language to the test for a more complex problem (loop recognition using Havlak's algorithm). The benchmark was initially proposed by Hundt [Hun11] for C++, Java, Go and Scala. We extended the benchmark with a Sparrow version of the algorithm. Under Hundt's constraints the Sparrow algorithm follows closely the implementations provided for other languages. Playing by the benchmark rules, we did not optimize the Sparrow version of the program by using data structures that are not standard in Sparrow, or by optimizing certain parts of the algorithm. As a result, the execution time of the Sparrow version was faster than the C++ version, the memory consumption was slightly higher than the C++ version, and the code size was smaller than for C++. On the other hand, the compilation time—which has not yet been a major concern in the development of Sparrow—is larger than for C++. Finally the number of lines of code required for the Sparrow version (while maintaining the main algorithm virtually unchanged) is smaller than for any other language in our benchmark.

As a general conclusion, one can say that Sparrow is in the same efficiency class as C++, but requires fewer lines of code. It lets programmers write efficient code while being concise.

You can't manage knowledge — nobody can. What you can do is to manage the environment in which knowledge can be created, discovered, captured, shared, distilled, validated, transferred, adopted, adapted and applied.

C. Collison, G. Parcell

CHAPTER 9

Moving computations to compile-time

Most compilers are able to use a variety of techniques in order to optimize the code that they are processing. One category of techniques involves identifying parts of the code for which there is enough information in order for them to be evaluated during compilation.

The extent to which such optimizations are possible depends on the language, the compiler implementation, and the characteristics of the target program. There are optimizations performed inside the compiler that are transparent to the user, but also optimizations at the level of the language itself, which require effort from the programmer's part. To achieve the latter, programming languages often employ the paradigm known as static metaprogramming [She01].

We explore the possibility of moving computations from run-time to compile-time in our hyper-metaprogramming framework. In this regard, there are two main concerns: efficiency and convenience. We attempt to determine if this technique is able to offer any significant improvements in performance for the run-time code, but also whether there are any trade-offs in terms of convenience.

We developed two case studies, which consist of practical applications of these techniques—minimal perfect hashing and regular expressions. With these case studies we intended to assess the benefits of moving computations from run-time to compile time by measuring the impact on the execution speed of the run-time code, and discussing the implications in terms of convenience. The obtained speedups are impressive.

9.1 Background

9.1.1 Hyper-metaprogramming

We defined hyper-metaprogramming as an extension of Turing-complete static metaprogramming (section 4.2). One can use hyper-metaprogramming to perform computations at compile-time. As hyper-metaprogramming imposes similar syntax, semantics, and abstractions, writing metaprograms requires the same amount of effort as writing traditional programs. Moreover, the hyper-metaprogramming requirements allow the same program to be written once and used both as a regular program and as a metaprogram, which offers the possibility of reusing run-time code as compile-time code. From a programmer's perspective, writing metaprograms becomes an easier task due to hyper-metaprogramming.

In addition, hyper-metaprogramming requires the programming language to support Turing-complete metaprogramming, and furthermore to be able to implement arbitrary data structures and algorithms at compile-time. In short, any computation that can be performed at run-time can be performed at compile-time as well.

If $[\![p]\!]$ is a run-time program that for a given input in produces the output $r = [\![p]\!][in]$, then we can translate the same program to compile-time, evaluate it, and obtain a semantically equivalent result:

$$\begin{aligned} &r^{ct} = \langle [\![p]\!][in^{ct}] \rangle_{ct} \\ &r^{ct} \equiv r \end{aligned} \tag{9.1}$$

After performing computations at compile-time, one typically wants to use the results of these computations at run-time. These results can be arbitrarily structured data, or code to be executed at run-time. Currently, hyper-metaprogramming does not impose any rules for moving the results from compile-time to run-time (code generation); different programming languages may implement different strategies to deal with this issue. We denote this by the $[\![tr]\!]$ function:

$$[\![tr]\!][r^{ct}] = r' \tag{9.2}$$

Taking eqs. (9.1) and (9.2) into account, and ignoring any possible differences that we might encounter between the run-time and compile-time execution environments, we have:

$$r' = r \tag{9.3}$$

In section 9.2 we describe how Sparrow implements these transformations from compile-time results to run-time code.

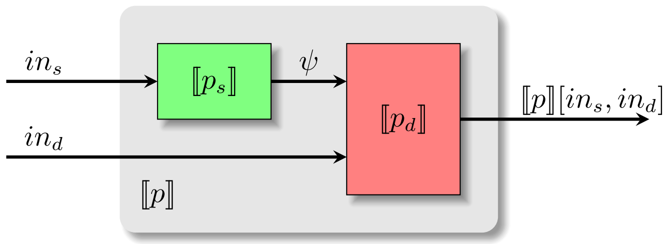


Figure 9.1: Original program, with both parts executed at run-time

It is important to note that hyper-metaprogramming does not impose a termination condition for the compilation process. If a metaprogram has bugs, it can cause the compiler to crash or behave incorrectly. Such drawbacks are commonly accepted in Turing complete run-time systems, therefore we also consider them acceptable in compile-time systems.

9.1.2 Moving computations from run-time to compile-time

As discussed in chapter 4, executing computations at compile-time also requires interpreter functionality from the compiler. Furthermore we typically have static parameters as well as dynamic parameters. This is a typical scenario for partial evaluation [Fut99; JGS93]. From a program evaluation perspective, we want to find a theoretical framework for moving computations from run-time to compile-time, and analyze its implications.

In a system that implements partial evaluation, a program[1] that depends on static parameters is transformed into a specialized version of the program. In this specialization, the static parameters are embedded into the program itself, which will only require dynamic parameters[2]:

$$[\![p]\!][in_s, in_d] = [\![p_{in_s}]\!][in_d] \tag{9.4}$$

If $[\![p]\!]$ takes static parameters, we have at least one possibility to divide the program into two parts $[\![p_s]\!]$ and $[\![p_d]\!]$ such that:

$$\begin{gathered}[\![p]\!][in_s, in_d] = [\![p_d]\!][\psi, in_d] \\ \text{where } \psi = [\![p_s]\!][in_s]\end{gathered} \tag{9.5}$$

Note that we can always find a trivial split given by $[\![p_d]\!] = [\![p]\!]$, $[\![p_s]\!] = \mathbb{1}$ and $\psi = in_s$. However, in many cases we can find a non-trivial $[\![p_s]\!]$ that actually does some useful processing on the static input. For the purpose of our argumentation, we assume that $[\![p_s]\!]$ performs the maximum possible amount of work.

[1]We use the term *program* also when referring to isolated computations and algorithms
[2]first Futamura projection [Fut99]

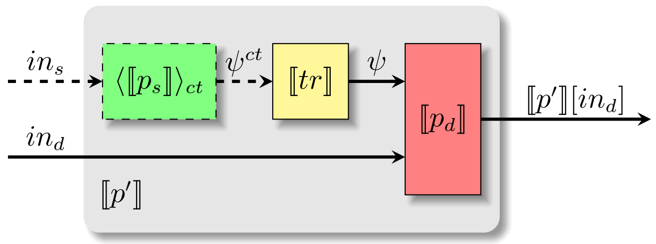


Figure 9.2: Specialized program. The dashed block and arrows are compile-time specific; they do not appear in the run-time program

Figure 9.1 shows a non-trivial split. In this division, $[\![p_s]\!]$ has only static inputs, whereas $[\![p_d]\!]$ only requires dynamic inputs and ψ, which is the result of $[\![p_s]\!][in_s]$.

To reduce the execution time of $[\![p]\!]$ we move the computations associated with $[\![p_s]\!]$ from run-time to compile-time; we produce a program $[\![p']\!] \equiv [\![p]\!]$ that does not contain the execution of $[\![p_s]\!]$ at run-time. As discussed in the previous section, hyper-metaprogramming can be used to execute the $[\![p_s]\!]$ program at compile-time without changing its syntax and semantics. We denote the result of the execution of $[\![p_s]\!]$ at compile-time by ψ^{ct}:

$$\psi^{ct} = \langle[\![p_s]\!][in_s]\rangle_{ct} \tag{9.6}$$

Note that ψ^{ct} is not the same as ψ from eq. (9.5). Although they both represent the result of the same computation, ψ^{ct} is available at compile-time, whereas ψ is only available at run-time. When the result of such a computation is a numeric value or a string, we can consider them to be the same, as in most programming languages there is no difference between numeric or string literals and their run-time counterparts. However, for some programs $[\![p_s]\!]$ may yield complex data structures (e.g. vectors, matrices, graphs of objects in memory) that cannot be easily moved from compile-time to run-time.

Still, no matter how complex the structure of ψ^{ct} is, we can always find a program $[\![tr]\!]$ that can transform the compile-time data to run-time. This program would simply have to generate run-time code that constructs the ψ value piece by piece from ψ^{ct}. We assume here that the programming language with support for hyper-metaprogramming has a method of doing this transformation:

$$\psi = [\![tr]\!][\psi^{ct}] \tag{9.7}$$

Consequently, using eqs. (9.5) to (9.7) we can transform our initial program $[\![p]\!]$ into:

$$\begin{aligned} [\![p']\!][in_s, in_d] &= [\![p_d]\!][\psi, in_d], \\ \psi &= [\![tr]\!][\psi^{ct}], \\ \psi^{ct} &= \langle[\![p_s]\!][in_s]\rangle_{ct} \end{aligned} \tag{9.8}$$

The structure of the specialized program $[\![p']\!]$ is depicted in fig. 9.2. The static part $[\![p_s]\!][in_s]$ is evaluated at compile-time, its result (ψ^{ct}) is translated into a form that can be used at run-time (ψ), and the latter is passed to the program $[\![p_d]\!]$.

If the language supports hyper-metaprogramming, and we are able to find a transformation $[\![tr]\!]$ that transforms ψ^{ct} into ψ while preserving the semantics, the new program is equivalent to the original program: $[\![p']\!] \equiv [\![p]\!]$.

We now analyze the effects of transforming $[\![p]\!]$ into $[\![p']\!]$ in terms of execution time. We denote by t_p, $t_{p'}$, t_{ps}, t_{p_d}, and t_{tr} the execution times for programs $[\![p]\!]$, $[\![p']\!]$, $[\![p_s]\!]$, $[\![p_d]\!]$, and $[\![tr]\!]$, respectively. We have $t_p = t_{ps} + t_{p_d}$ and $t_{p'} = t_{tr} + t_{p_d}$. In the execution of $[\![p']\!]$ we do not have the t_{ps} component anymore (it has been moved to compile-time), but we have a new component t_{tr}, corresponding to the translation of the static data ψ^{ct} to run-time data.

Let $\alpha = t_{ps}/t_p$ be the ratio of the static component to the original program, and $\beta = t_{tr}/t_{ps}$ the ratio of the ψ^{ct} to ψ transformation with respect to the generation time of ψ.

Theorem 1 *The speedup after transforming $[\![p]\!]$ to $[\![p']\!]$ is:*

$$su = \frac{t_p}{t_{p'}} = \frac{1}{1 - \alpha + \alpha\beta} \tag{9.9}$$

The proof follows immediately from the program structure[3].

If ψ has a simple structure (e.g. numbers, strings), then $\beta = 0$. Also, it can be safely assumed that for most problems transforming compile-time data into run-time data is much faster than actually generating the data, and thus $\beta \approx 0$. In this case, the speedup will be:

$$su \approx \frac{1}{1 - \alpha} \tag{9.10}$$

To exemplify, if $\alpha = 0.5$ (half of the program is dependent only on the static data) and $\beta = 0$ we obtain a speedup $su = 2$. This means that by moving the static computations from run-time to compile-time, the program will run twice as fast.

It is important to note that the transformation of $[\![p]\!]$ into $[\![p']\!]$ discussed here does not necessarily make $[\![p']\!]$ less convenient than $[\![p]\!]$ to write and use. Provided that hyper-metaprogramming is implemented in the language, the interface of the program can remain the same. We demonstrate this aspect in sections 9.2.1, 9.3 and 9.4.

[3]This is similar to Amdahl's law for parallel computing

9.2 Sparrow support

As discussed in chapter 4, in Sparrow, compile-time code is quasi-identical to run-time code. However, the programmer needs to specify which program constructs should be used at compile-time, and which should be used at run-time. From a syntactical point of view, the programmer needs to add a modifier—`[rt]`, `[ct]`, `[rtct]`, or `[autoct]`—when introducing declarations (classes, functions, and variables). The definitions marked with `[rt]` or `[ct]` can be used only at run-time or compile-time, respectively. A `[rtct]` modifier indicates that the definition can be used both at run-time and compile-time, depending on the context in which they are used. The `[autoct]` modifier is an extension of the `[rtct]` modifier for functions, which indicates that if all the arguments passed to a function are compile-time, then the function should be executed at compile-time, even if the function is called from a run-time context. The default mode is `[rt]`.

Apart from specifying the *evaluation mode* for certain declarations, the programmer can use classes, functions, variables, expressions, and control structures at compile-time, with no syntactic change from traditional run-time programming. More details on how Sparrow implements hyper-metaprogramming are presented in chapter 4.

To move the results of compile-time computations (data or code) to run-time, Sparrow defines some helper constructs: compile-time control structures, the `ctEval()` function, and compile-time to run-time constructors.

Structures like `if[ct]`, `while[ct]` and `for[ct]`—analogous to the traditional `if`, `while`, and `for` structures—represent convenient methods of including one or more code blocks in the run-time code, based on variables known at compile-time; these structures can be used both inside functions and outside of them, at the level of declarations. (see section 7.5.5)

A `ctEval(`*`ct-expr`*`)` function call makes sure that the given expression is compile-time evaluated at the point where the `ctEval` is type checked. If a compile-time expression is not enclosed in a `ctEval()` function call, it may be evaluated at a later stage by the compiler, typically when the actual backend code is generated. This may be too late, as the compile-time expression may have changed (e.g. an induction variable in a `for[ct]`).

To translate complex data structures (declared as `[rtct]`) from compile-time to run-time, Sparrow uses the so-called `ctorFromCt` constructors. These are called for data structures whose compile-time values need to be translated to run-time. The translation of basic types is defined by the language, and the rest of the types need to implement this constructor so that they can be moved from compile-time to run-time. Similarly to providing default constructors and copy constructors, Sparrow tries to automatically generate a `ctorFromCt` constructor, but the programmer can override this using his custom version.

A simple example of `ctorFromCt` for a Vector class is given in listing 9.1. The example also shows the usage of `for[ct]` and `ctEval`. The `for[ct]`

```
fun[rt] ctorFromCt(src: Vector ct) {
    this.ctor();
    this.reserve(ctEval( src size ));
    for[ct] ( el = src.all() )
        this.pushBack(ctEval(el));
}
```

Listing 9.1: Example of a `ctorFromCt` constructor

is used to generate a series of run-time calls to the `pushBack` method, passing different values obtained from the compile-time vector. The `ctEval()` function calls are needed here to indicate that the compiler must evaluate the expressions for each **for**`[ct]` iteration, making sure the right values are pushed into the run-time vector.

As explained in chapters 4 and 5, the current implementation of the Sparrow compiler implements two execution environments in a backend based on the LLVM framework [LLVM]: one for generating run-time code, and one for handling compile-time computations. Depending on the modifier, the compiler places each declaration in the corresponding execution environment; it may happen that the same code (if declared with `rtct` or `autoct`) is placed in both execution environments. The code that goes into the run-time execution environment is optimized and compiled just like in any traditional compiler; on the other hand, the code that goes into the compile-time execution environment is interpreted during the compilation process.

9.2.1 A simple example: the factorial

A common example used in literature to illustrate compile-time evaluation is the factorial [Por10]. In this section we look at the Sparrow implementation for this function, and reiterate how one can invoke it at compile-time.

The Sparrow recursive implementation for the factorial function is depicted in listing 9.2. The `fact` function is declared with the `[autoct]` modifier, informing the compiler that it should be available for use both at run-time and at compile-time. If the argument passed when calling `fact` is available at compile-time, the function execution will take place at compile-time. Apart from the `[autoct]` modifier, the programmer does not need to make any changes to the code in order to enable compile-time execution.

The `testFact` function calls both the run-time and the compile-time version of `fact`. In the call to `fact` at line 7, the function is given a run-time variable as an argument, thus the function call will be handled at run-time. At line 8, the argument is a compile-time constant; this makes the compiler treat the entire `fact(5)` function call as a compile-time expression, thus calling the factorial function at compile-time.

```
fun[autoct] fact(n: Int): Int {
    if ( n == 0 ) return 1;
    else return n*fact(n-1);
}
fun testFact() {
    var n = 5;          // run-time variable
    writeLn(fact(n));
    writeLn(fact(5));
}
```

Listing 9.2: Definition and usage of the factorial function in Sparrow

```
 1  define void @testFact() {
 2      %n = alloca i32
 3      call void @Int_ctor_Int(i32* %n, i32 5)
 4      %1 = load i32* %n
 5      %2 = call i32 @fact(i32 %1)
 6      call void @writeLnInt(i32 %2)
 7      call void @writeLnInt(i32 120)
 8      call void @Int_dtor(i32* %n)
 9      ret void
10  }
```

Listing 9.3: Generated LLVM code for the `testFact` function

The run-time code produced by the LLVM backend for the `testFact` function is listed in listing 9.3. It illustrates the outcome of the process that we described. At line 5 in the LLVM code, there is a run-time call to the `fact` function; its result is printed to the console at line 6. In contrast, there is no corresponding call for computing `fact(5)`; the compiler simply inserts the constant `120`—the result of the compile-time computation—into the call at line 7.

We can see that the same definition for the function is used in both cases in this example. In addition, the calls to `fact` themselves are indistinguishable from one another. In other words, run-time and compile-time evaluation are supported under a homogeneous interface. The `[autoct]` modifier is the only syntactical addition needed to enable this behavior. Despite its simplicity, the factorial function is able to exemplify how hyper-metaprogramming can facilitate this technique with minimal convenience costs.

In terms of performance, the entire factorial computation is moved to compile-time. The α factor defined in section 9.1 is 1. Because we do not have to do any operations to move integers from compile-time to run-time, the β factor is 0. Thus if we apply the speedup formula, we obtain $su = \infty$ for this trivial example.

9.3 Case study: minimal perfect hashing

A hash function h defined over a set of keys U is called a minimal perfect hash function if it maps the keys to the domain $\{0, ..., |U|-1\}$ and it is bijective on U ($h : U \to \{0, ..., |U| - 1\}$) [BBD09; CFDH92; BZ08]. Compared to a traditional hash function, it does not produce collisions ($k_1 \neq k_2 \Leftrightarrow h(k_1) \neq h(k_2)$), and hence does not require more storage than the actual size of U.

Unlike traditional hashing methods, constructing a minimal perfect hash function requires knowledge of all the keys at initialization, and the keys cannot change during its lifetime.

In cases in which the set of keys is fixed, it may be possible for it to be available at compile-time. For example, the GPERF generator [Sch00] constructs a perfect hash function for a list of known keywords, during the build process; the perfect hash function can then be used in compilers to test whether tokens are keywords or not. Even though creating the perfect hash function at compile-time has benefits (e.g. no run-time cost for creating the hash function, offline optimization of the hash function), this approach also poses some disadvantages, mainly concerning convenience: an external tool needs to be integrated in the build chain, the integration with the generated code is not always smooth, the external tool cannot be easily extended, etc.

We raise the issue of whether we can create a minimal perfect hashing implementation that has all the benefits of knowing the keys in advance, at compile-time, but does not compromise convenience—the algorithm should be able to work with both run-time and compile-time data, without changing its interface.

If such an algorithm is implemented, we need to check if we can obtain a speedup according to eq. (9.9) for building the hash function and then accessing it, without changing the way a programmer would use this structure.

9.3.1 Implementation details

We implemented a simplified version of the CHD algorithm [BBD09]; the compression step was not included, making the algorithm similar to the one presented in [CFDH92]. The CHD algorithm has a construction time of $O(n)$ and an access time of $O(1)$. In our implementation, the sorting step uses an $O(n \log n)$ algorithm, thus the overall construction time is $O(n \log n)$.

The public interface of the class that encapsulates the CHD algorithm is depicted in listing 9.4. Two constructors can be used to initialize the minimal perfect hash class, and two methods can be used to access the keys in the hash structure; in addition we have a `ctorFromCt` constructor for translating compile-time `MinPerfHash` objects to run-time.

The class can be used both at run-time and at compile-time, as it is declared using the `[rtct]` modifier. The `ctorFromCt` constructor ensures that we can safely convert compile-time instances of this class to run-time instances. Seeing

```
class[rtct] MinPerfHash {
    fun ctor(keys: @Vector(String));
    fun ctor(keys: @Vector(String), extraSpace: Double);
    fun[rt] ctorFromCt(src: MinPerfHash ct);
    fun search(key: String): UInt;
    fun ()(key: String) = search(key);
}
```

Listing 9.4: Interface for the `MinPerfHash` class

```
fun mph(keys: @Vector(String)): MinPerfHash {
      var hash = MinPerfHash(keys);
      return hash;
}
fun mph(keys: @Vector(String) ct): MinPerfHash {
      var[ct] hashCt = MinPerfHash(keys);
      var hash = hashCt;
      return hash;
}
```

Listing 9.5: Specializations for the function that builds a minimal perfect hash function

as how the class only contains an array of integers and two numbers, we could have used the automatically generated `ctorFromCt` constructor; however it turns out that a hand-crafted implementation of this constructor is faster in terms of compilation time than the generated constructor (see section 9.5 for more details).

Given a set of keys (run-time or compile-time) we have two functions that generate the corresponding `MinPefHash` object; these functions are depicted in listing 9.5. The compile-time version of the function first creates a compile-time variable for the minimal perfect hash, then a corresponding run-time variable, which is returned.

It is important to note that the programmer should use the same syntax for creating a minimal perfect hashing structure for both compile-time keys and run-time keys. This way, there is no change in convenience from user's perspective. Also from a convenience standpoint, there are minimal costs for the implementer of the minimal perfect hashing class: the `MinPerfHash` needs to be declared as `[rtct]`, an optional `ctorFromCt` constructor can be added, and also an additional overload to easily create `MinPerfHash` objects.

Therefore, by having hyper-metaprogramming support, we can move the building of the hash function at compile-time without losing convenience.

9.3.2 Experimental setup

The experiments were performed on a 3.5 GHz Intel Core i7 3770K machine, with 16 GB of RAM, running Ubuntu 13.10. The backend for the Sparrow compiler uses LLVM version 3.2, and the C/C++ programs were compiled using Clang version 3.2. The compilers were called with the -O3 optimization flag.

For this case study, we considered three implementations that perform minimal perfect hashing, and measured their execution time for various problem sizes. We tested a reference CMPH implementation [CMPH; BBD09], a Sparrow version that builds the hash function at run-time, and a Sparrow version that builds the hash function at compile-time. We compared the Sparrow run-time version to the CMPH implementation to make sure that there are no major differences in performance due to our simplified CHD algorithm. Our main concern was the difference in performance between the two Sparrow versions.

Each of these programs builds a hash function starting from a set of keys, and subsequently performs a series of run-time operations on the resulting structure. These operations consist of iterating through the keys, checking whether there are any duplicates (using a bitset structure), and also whether the result of the hashing exceeds the expected limit.

The keys used to build the hash functions were provided in a text file consisting of over 260,000 English words. We varied the problem size by selecting a different number of words for each run. Thanks to hyper-metaprogramming, we were able to read the keys from the external file at compile-time in the Sparrow CT version.

For all of the tests we used a load factor of 99% [BBD09].

To be able to verify the theoretical framework, we first computed the ratio between the time required to build the hash function and the time required to access it, for the Sparrow RT program. We varied the number of times the second step was executed, and we observed the differences in terms of total execution time. From this ratio, one can easily derive the α factor (see section 9.1.2). We considered the β factor to be in the range $\beta \in [0..0.1]$.

9.3.3 Results and discussion

The measured execution times are plotted as functions of the number of keys in fig. 9.3. Although the CMPH implementation performed better than the Sparrow RT version, we consider our implementation a reasonable one; it was not our goal to outperform the reference implementation.

When comparing the two Sparrow versions, we notice a significant difference in execution time. Whereas the execution time for the RT version increases rapidly with the problem size, the execution time for the CT version remains almost constant.

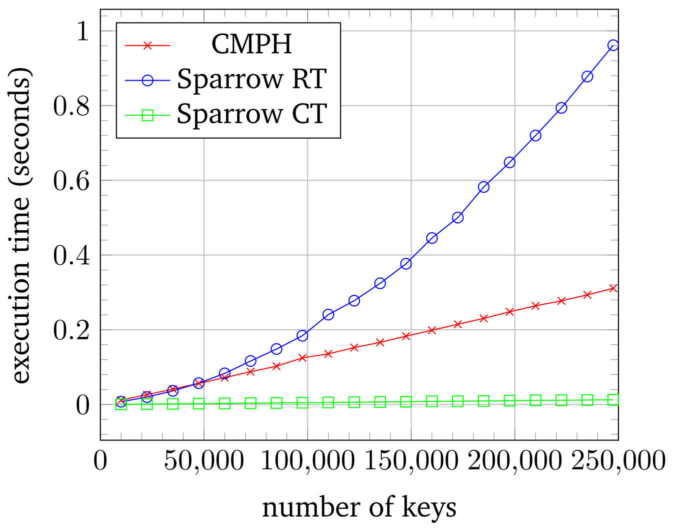


Figure 9.3: Execution times for the minimal perfect hashing implementations

This difference in performance is illustrated more clearly in fig. 9.4, where we plot the relative speedup, also as a function of the number of keys. The speedup between the CT and RT versions increases linearly with the problem size. Intuitively, the more computations we can move from run-time to compile-time, the greater the speedup will be.

After several measurements we determined that for 160,000 keys the ratio between the time required to build the hash and the time required to access it is 0.015, which makes $\alpha = 0.985$. According to eq. (9.9), if $\beta \in [0..0.1]$ we have a theoretical speedup $su \in [40..67]$. Indeed, if we compare the execution of the RT version to the CT version of the program, for 160,000 keys we obtained a speedup of 52.18. Therefore, the speedup obtained by the CT implementation over the RT implementation is in accordance to the predicted theoretical speedup.

This case study validates our assumption: we can produce significant execution speedup by moving the computations from run-time to compile-time, without compromising convenience.

One can also look at these benefits from an opposite perspective: it is possible to reduce the costs of configuration management and software development needed when using external offline tools, while maintaining a similar degree of performance. A language with hyper-metaprogramming support does not require external tools to move computations to compile-time.

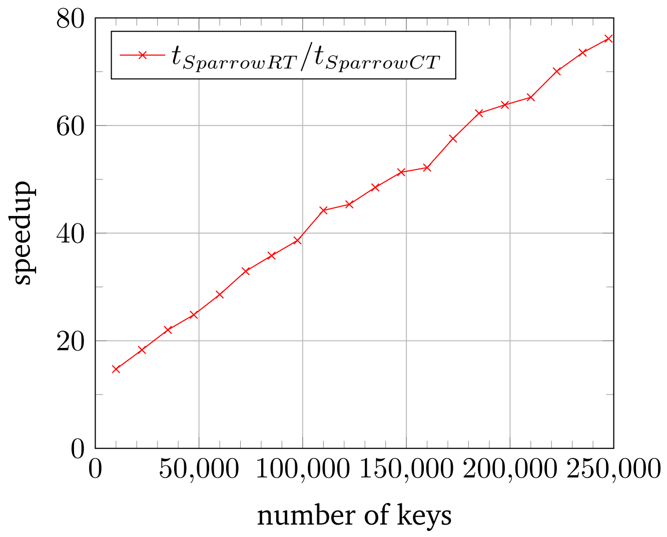


Figure 9.4: Speedup of Sparrow CT over Sparrow RT for minimal perfect hashing

9.4 Case study: regular expressions

Regular expressions [HMU07; Cox07] are a pervasive tool in any application that involves text processing. Consequently, any improvement in performance for regular expressions is likely to affect a significant number of programs.

There are two major steps in using a regular expression algorithm: building the automaton from a string representation, and matching strings against the constructed automaton. If $|r|$ is the length of the regex string and $|x|$ is the length of the test string, then the complexity of building an NFA is $O(|r|)$, and the complexity of testing a string is $O(|r| \times |x|)$. If we use a DFA algorithm, we have an $O(|x|)$ complexity for testing a string, but an $O(|r|^3)$ complexity for building the automaton in the typical case, and $O(|r|^2 2^{|r|})$ in the worst case [HMU07].

In practice, most often a regular expression is represented by a compile-time string literal. In such cases, the compiler can take advantage of the fact that the regular expression is available at compile-time.

As was the case with minimal perfect hashing, we raise the issue of whether we can move computations from run-time to compile-time in order to improve the execution speed of regular expressions, while maintaining the same degree of convenience that users expect. If we are able to move the construction of the automaton to compile-time, the obtained speedup must conform to eq. (9.9).

```
class[rtct] RegexAutomaton {
    fun ctor(re: String);
    fun[rt] ctorFromCt(src: RegexAutomaton ct);
    fun match(str: String): Bool;
    fun ()(str: String) = match(str);
}
fun regexMatch(re: String, str: String): Bool {
    var automaton = RegexAutomaton(re);
    return automaton.match(str);
}
fun regexMatch(re: String ct, str: String): Bool {
    var[ct] automatonCt = RegexAutomaton(re);
    var automaton = automatonCt;
    return automaton.match(str);
}
```

Listing 9.6: Interface for the class that represents a regular expression automaton and two wrapper functions that perform regex matching

9.4.1 Implementation details

We based our work on Cox's DFA bounded-memory implementation for regular expressions [Cox07]; it should be viewed as a proof-of-concept implementation, not an industrial strength solution. He offers a simplistic implementation that builds an NFA, and then incrementally constructs a DFA on top of it. His DFA is not built completely from the beginning; it is constructed incrementally as different strings are being tested.

Unlike Cox's implementation, we build the complete DFA before testing any strings; this gives us the possibility to move the entire automaton construction to compile-time, when the regular expression is available. For similar reasons, we used DFAs and not NFAs. While it is true that a DFA algorithm has a high complexity for building the automaton, in our implementation this cost has no impact at run-time. The only run-time cost is the one associated with testing the strings, which has $O(|x|)$ complexity, in contrast to an NFA algorithm's $O(|r|\times|x|)$. We designed and implemented our solution so that each automaton is built only once, but can be subsequently used anywhere in the application in an efficient manner.

The public interface for our regular expression automaton is presented in listing 9.6, along with two function overloads that wrap the automaton building and matching functionality. The `RegexAutomaton` class builds the entire automaton on construction. The two functions build the automaton from a regex string, and then use the given test string for matching; the automaton can be constructed both for run-time and compile-time strings.

Regular expression	Test string
$a?^n a^n$	a^n
$(a\|b)?^n (a\|b)^n z$	$a^n z$
$(a\|b)?^{2n} (a\|b)^n z$	$a^{2n} z$

Table 9.1: Formats of regular expressions and test strings

9.4.2 Experimental setup

The experiments for regular expressions were performed on the same platform as the minimal perfect hashing experiments (section 9.3.2).

Like in the previous case study, we tested three implementations: a reference implementation for regular expressions, a Sparrow run-time version, and a Sparrow compile-time version. As a reference we used Cox's memory-bounded DFA implementation [Cox07]. Again, our main concern was the difference in performance between the two Sparrow versions.

Each of these programs was tested in terms of execution time for three types of regular expressions, and for different lengths. The strings on which the regular expressions were tested were chosen to always be positive matches, in order to achieve maximal traversal of the automaton states. Table 9.1 lists the different types of regular expressions and the format of the strings on which they were used, as functions of length. We chose the same expressions that Cox used in his paper [Cox07]; the expressions are not used in the real world, but their construction is simple and we can easily reason about their complexity.

The Sparrow run-time version takes the regular expression as a run-time variable, whereas the compile-time version builds the entire DFA at compile-time from a string literal.

As in the minimal perfect hashing case study, we wanted to validate the theoretical framework. We ran the Sparrow run-time program several times with regular expression $(a|b)?^{2n}(a|b)^n z$, each time increasing the number of times we tested the string $a^{2n}z$ (one build of an automaton, several match tests). This allowed us to compute the α factor. Again, we considered β to be in the range $\beta \in [0..0.1]$.

9.4.3 Results and discussion

For the three regular expression types, the measured execution times are plotted as functions of n in figs. 9.5 to 9.7. In all three cases, the reference implementation performed better than the Sparrow RT version; this was expected, as we are building the full automaton from the beginning, whereas the reference implementation builds the states lazily, only when required.

Comparing the Sparrow CT version to the Sparrow RT version we observed an increase in speed in all three cases. This speedup is depicted in fig. 9.8 for

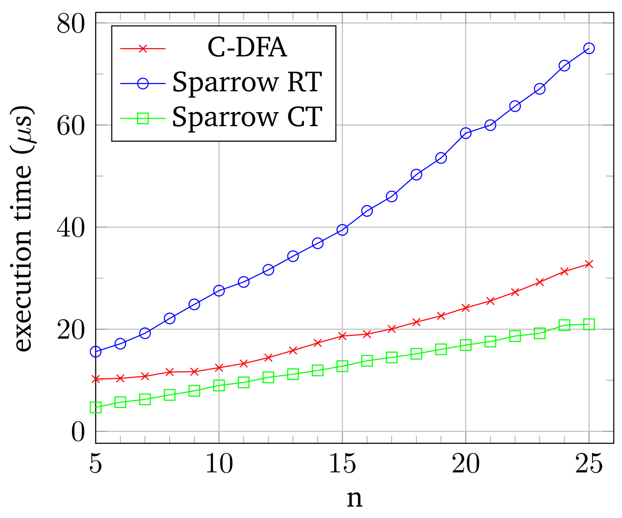


Figure 9.5: Execution times for the regular expression $a?^n a^n$

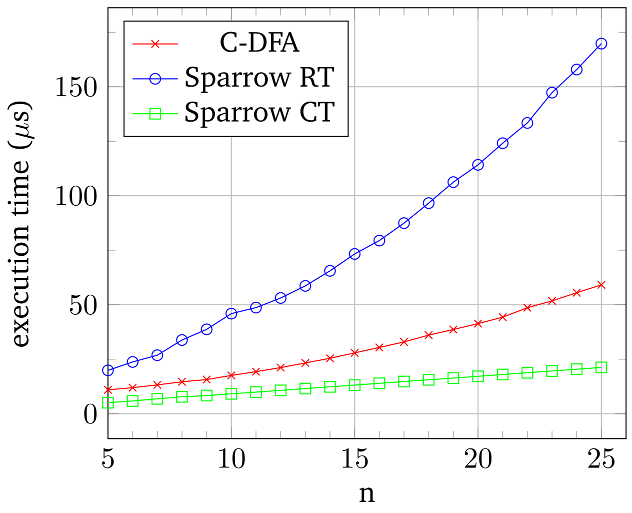


Figure 9.6: Execution times for the regular expression $(a|b)?^n(a|b)^n z$

all the regular expression types.

Again, the results show that the speedup increases with the complexity of the computations that are moved to compile-time.

One might argue that the regular expressions that we used are not realistic, thus making these speedup tests irrelevant. Indeed, these tests do not measure speedups for real-world scenarios, but we can assume that real-world regular expressions are more complex than the ones we used; considering the trends shown by our experiments, we would expect even larger speedups.

We measured for the run-time implementation the ratio between building

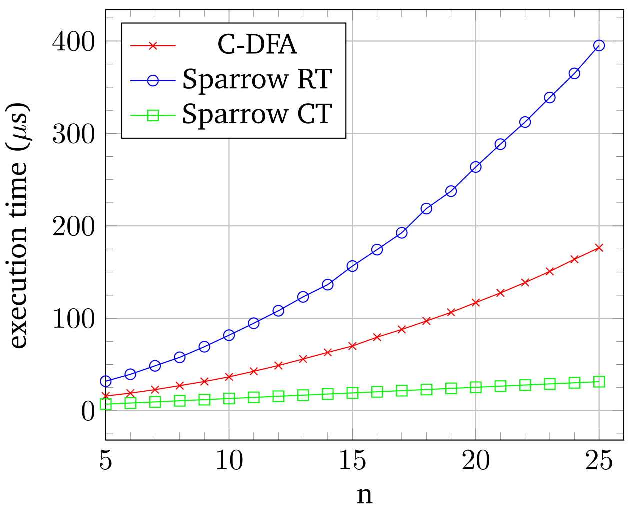


Figure 9.7: Execution times for the regular expression $(a|b)?^{2n}(a|b)^n z$

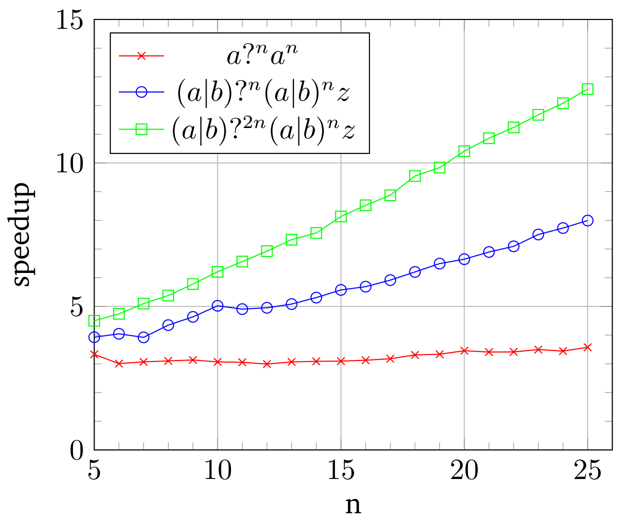


Figure 9.8: Speedup of Sparrow CT over Sparrow RT for the three regular expressions

the automaton for the $(a|b)?^{50}(a|b)^{25}z$ regular expression and for testing this automaton with the string $a^{50}z$; it is approximately $13 \cdot 10^3$ times slower to build this automaton than to do a match test on it. This yields an α factor of 0.9999. For $\beta \in [0..0.1]$, according to eq. (9.9) we should obtain a speedup in interval $su \in [10..13660]$ (in this case the α factor is so close to 1 that a small change in β produces very different results). Our measured speedup of the compile-time version over the run-time version for the given regular expression is 12.57, which is in the predicted range.

Our second case study shows that moving computations from run-time to compile-time is possible without compromising convenience. Seeing as how regular expressions are used in many applications, these performance improvements can be considered significant.

9.5 Secondary findings

Compilation time. The fact that the compiler must perform additional computations naturally implies that the compilation time increases. The compiler must contain a kind of embedded interpreter (the Sparrow compiler uses an LLVM JIT engine [LLVM]) which is used to execute compile-time code. In addition to the time that it takes the interpreter to actually execute the code, we can consider other factors: the time required to optimize and generate machine code for each new block created by the metaprograms, the number of times we need to invoke the compile-time execution engine, or the size of the compile-time code. In our minimal perfect hashing example, the compilation time increased by 13 seconds for 150,000 keys; for the regular expressions example, it increased by 0.6 seconds for building a DFA for the $(a|b)?^{50}(a|b)^{25}z$ expression. Solutions to decrease the compilation-time can be devised: optimize the compile-time code before executing it, reduce the number of times the compile-time execution environment is invoked, save computation results to files to avoid recomputing them on subsequent compilations, etc.

Executable size. The size of the resulting binary executables also tends to increase, in order to accommodate the additional code and data that are generated. It should be noted that not all of these results are preserved in the executable, only the ones that are actually used by the run-time code. In some cases, where complex computations produce results that have a simple structure, we may find that the executable size decreases; this can happen when parts of the code are no longer necessary at run-time, and are therefore removed. In our minimal perfect hashing example the size of the binary increased by approximately 43KB for every 50,000 keys.

Code generation can be slower than the actual computations. Somewhat unexpectedly, in some cases it is much slower to generate code for the results than to actually compute them. To illustrate this, let us consider the translation of the displacement table for minimal perfect hashing to run-time code; a basic implementation would resemble the one in listing 9.1. If we have 100,000 keys, the compiler needs to invoke the `ctEval` function 300,001 times and to generate 100,000 `pushBack` function calls; this is obviously very slow—in fact it is much slower than computing the entire hash table. For this particular case, it was more efficient to interpret the array of displacements as a memory region, translate that region to run-time as a string literal, and finally reinterpret it as an array.

9.6 Conclusions

In this chapter we explored the requirements and implications of employing hyper-metaprogramming with the aim of moving computations from run-time to compile-time. We showed that, without compromising convenience, it is possible to obtain significant speedups for programs that contain computations based on parameters which are available at compile-time.

Hyper-metaprograming imposes some restrictions on compile-time metaprogramming, and by doing so it alleviates the main problems associated with traditional static metaprogramming. As we emphasized on various occasions, writing metaprograms becomes an easier task from the programmer's perspective. It involves the same programming paradigm, data abstractions, and semantics as traditional programming; from a syntactical point of view, only a small set of modifiers is required. Furthermore, metaprograms can be employed to solve complex, real-world problems.

Sparrow was used as a supporting programming language that implements hyper-metaprogramming. The language proved to be a suitable working framework that allows the techniques presented in this chapter to be implemented and tested.

We presented two applications where we employed the technique of moving computations to compile-time. The speedup obtained for the two case studies makes moving computations from compile-time to run-time an appealing technique for improving the performance of programs.

If we understood something just one way, we would not understand it at all.

M. Minsky

Chapter 10
Compile-time execution efficiency

By now, the reader should be convinced that metaprogramming techniques can have a positive impact on the efficiency of the language (e.g., by moving computations to compile-time), and that they can help the language become more flexible and more natural (e.g., by creating a powerful framework for generic programming). There must be a downside too, right?

One of the bigger problem with compile-time metaprogramming is that it does not scale well in practice. Not only are the compilation processes slower, but the compilers typically limit the amount of computations one can perform. Such is the case with C++ template metaprogramming, where executing computations at compile-time is typically slow, and compilers have limits regarding the number of templates that can be instantiated, therefore reducing the number of computations that can actually be performed [Dub13; AG04; VJ02].

We described our hyper-metaprogramming as a Turing-complete static metaprogramming system that allows writing metaprograms with the same syntax, semantics, and idioms as traditional run-time programs. Still, is this true in practice? Can hyper-metaprogramming be used with ease to perform computations at compile-time, in a reasonable amount of time?

In this chapter we explore the feasibility of using hyper-metaprogramming to perform computations at compile-time. We use as a benchmark the N-Queens problem that Dubrov proposed for evaluating the template metaprogramming support of C++ compilers [Dub13]. We show that our metaprogramming system is very convenient for executing computations at compile-time, and it is also very fast—much faster than C++ template metaprogramming, and moreover as fast as a run-time execution environment.

10.1 Performing computations at compile-time in Sparrow

This section iterates over the most important Sparrow concepts that provide a seamless hyper-metaprogramming user-experience. Part of these concepts can also be seen in chapters 4, 5 and 9. Our goal here is to make this chapter as self-contained as possible, without requiring the reader to jump back and forth between the chapters.

In Sparrow, writing metaprograms is quasi-identical to writing regular run-time programs. The programmer just needs to specify which programming constructs should be used at compile-time, and which should be used at run-time—the *evaluation mode*. This distinction must be expressed whenever a definition (class, function, or variable) is introduced, with the use of so-called modifiers: `[rt]`, `[ct]`, `[rtct]`, and `[autoCt]`.

If a definition is intended exclusively for run-time, the `[rt]` modifier should be used. Similarly, when a definition is needed for compile-time only, the `[ct]` modifier should be used. If the same definition should be available both at run-time and at compile-time (as the last rule of hyper-metaprogramming requires), `[rtct]` or `[autoCt]` modifiers should be used. When a definition has the `[rtct]` modifier attached, the context in which the definition is used dictates the evaluation mode of the definition. The `[autoCt]` modifier is an extension of the `[rtct]` modifier, tailored for functions; if all the arguments passed to such a function are compile-time, then the function should be executed at compile-time, even if the function was called from a run-time context.

For example, if we declare a function `add(x,y: Int)` that adds two numbers as `[rtct]`, then, each time we call this function with compile-time arguments from a run-time algorithm, the operation is performed at run-time. However, if this function is declared as `[autoCt]`, the addition is performed at compile-time even inside run-time algorithms. This can move some of the effort from run-time to compile-time. On the other hand, we do not want functions that print something to the console to be executed at compile-time, even if we pass them compile-time string arguments.

By default, in Sparrow all definitions are `[rt]` if not otherwise specified.

To move the results of compile-time computations to run-time, Sparrow provides some compile-time control structures, a compile-time evaluation mechanism, and compile-time to run-time conversion constructors.

The **`if`**, **`while`**, and **`for`** structures with which the programmer is familiar have their compile-time counterparts **`if`**`[ct]`, **`while`**`[ct]`, and **`for`**`[ct]`, in order to be able to generate run-time code. These structures must only be provided with compile-time expressions. Depending on the values of these expressions, the compile-time control structures will generate zero, one, or more code blocks in the final code. It is possible to use these structures both inside functions and outside of them, at the level of definitions. The **`if`**`[ct]` is somewhat similar to the **`#if`** preprocessing command from C++, but it can use values produced by the compiler at later stages, for example as the result of a compile-time algorithm. The **`while`**`[ct]` and **`for`**`[ct]` have no C++ counterparts.

By default, the compiler does not try to eagerly evaluate compile-time expressions. In fact, as the evaluation can be costly, it tries to delay the moment of evaluation as much as possible. Still, there are cases in which the programmer may want to evaluate some compile-time expression earlier. For these cases, Sparrow offers a special function: a `ctEval(`*`ct-expr`*`)` function call makes sure that the given expression is evaluated at the point where the function call is type checked. These function calls are typically used in constructs that involve variables or **`while`**`[ct]` and **`for`**`[ct]` structures to capture the actual value of a variable when generating code, before the variable is changed.

To translate a value of a given type from compile-time to run-time, the compiler uses a special constructor, named `ctorFromCt`. Each time a compile-time value of type T (declared as `[rtct]`) needs to be moved to run-time, the compiler creates a new run-time object of type T, and calls the `ctorFromCt` constructor passing the compile-time value as an argument. Similarly to default constructors, copy constructors, and destructors, the compiler tries to generate this constructor automatically for each `[rtct]` class. The automatic generation may fail if the class contains references or sub-objects that do not have this constructor defined. However, the programmer may write his own version.

As a typical compiler backend has support for numeric and string literals, converting such values from compile-time to run-time is a simple case of generating such literals. In this case, Sparrow does not fundamentally differ from other programming languages. However, Sparrow needs to transfer to run-time complex compile-time objects, with multiple sub-objects, with possible pointers to other objects or to themselves, or with constraints that need to be satisfied (e.g, a sub-object must contain a pointer to itself). Most programming languages encounter this problem when serializing and deserializing objects to/from a stream. As previous experience shows us, serialization and deserialization require code to be executed in both operations. This is why Sparrow needs the compile-time to run-time conversion constructors. This is where all the operations required for moving the object state from compile-time to run-time, while preserving the object constraints, need to be placed.

An example of a `ctorFromCt` constructor for a `Vector` generic class is given in listing 10.1. Besides showing a compile-time to run-time conversion constructor, this example also illustrates usage examples for the `ctEval` function call and the **`for`**`[ct]` construct. The `ctEval` call is needed to capture

```
fun[rt] ctorFromCt(src: Vector ct) {
    this.ctor();
    this.reserve(ctEval( src size ));
    for[ct] ( el = src.all() )
        this.pushBack(ctEval(el));
}
```

Listing 10.1: Example of a `ctorFromCt` constructor

the compile-time value of the `src size` expression inside the function call, and not inside code generation after the compile-time values are no longer valid. The **for**`[ct]` construct has the effect of generating several calls to the `pushBack` function inside the `Vector` class, adding the right values to the run-time version of the object. As the number of `pushBack` function calls is the same as the number of values inside the compile-time object, the bigger the `Vector` is, the larger the generated code is.

The reader may have noticed that the type of the parameter of the constructor is a `Vector ct`. This expression is in fact a postfix operator call to the `ct` function. It takes a type (in our case `Vector`, for which the evaluation mode is `rtct`) and returns another type, the compile-time version of the given type. Such compile-time operations are commonplace in Sparrow, and can be employed with little effort.

This conversion constructor preserves `Vector` semantics and works in all cases, no matter how complex the type of the inner objects is. If the element type is a number, then a more efficient version can be implemented, by interpreting the vector data simply as a raw memory region, and then viewing it as a `String`. Such an approach will generate fewer instructions, both for compile-time and run-time, and thus is more efficient. Section 10.3.3 discusses a concrete example in which this technique can be used, and provides measurements to prove that it is actually efficient.

To implement hyper-metaprogramming, the Sparrow compiler employs two execution environments in the backend: one to generate run-time code, and the other to interpret compile-time metaprograms. The backend is implemented using the LLVM framework [LLVM; LV04]. Based on the evaluation mode of a definition, the Sparrow compiler puts the definition in the right execution environment. If the definition uses the `[rtct]` or `[autoCt]` modifiers, the definition will be placed in both execution environments.

For the compile-time execution environment the compiler employs a just-in-time interpreter provided by LLVM. To execute a function at compile-time, the compiler has to invoke the interpreter, pass the right values for arguments, and possibly retain the result of the function call. This way, the metaprograms are actually executed inside the compiler, with a backend similar to the one used for run-time code generation. As we will see, this implementation deeply

```
using PlacementType = Array(Int);
using SolutionsType = Vector(PlacementType);
fun[rtct] testQueens(sol: @PlacementType, k, y: Int): Bool {
    for ( i = 0 .. k )
        if ( y == sol(i) || k-i == Math.abs(y-sol(i)) )
            return false;
    return true;
}
fun[rtct] backtracking(curSol: @PlacementType, k, n: Int, res: @SolutionsType) {
    for ( y = 0 .. n ) {
        if ( testQueens(curSol, k, y) ) {
            curSol(k) = y;
            if ( k == n-1 )
                res.pushBack(curSol);
            else
                backtracking(curSol, k+1, n, res);
        }
    }
}
fun[rtct] nQueens(n: Int): SolutionsType {
    var res: SolutionsType;
    var curSol: PlacementType = n;
    backtracking(curSol, 0, n, res);
    return res;
}
```

Listing 10.2: N-Queens, complete Sparrow implementation

affects our measurements.

10.2 N-Queens in Sparrow

In order to illustrate the capabilities of hyper-metaprogramming, we implemented the N-Queens problem in Sparrow with the help of hyper-metaprogramming. This way, we have a comparison between hyper-metaprogramming in Sparrow and metaprogramming in C++.

Let us start by describing how the N-Queens problem can be implemented in Sparrow for run-time. The Sparrow code is depicted in listing 10.2. The code displays two **using** directives that act similarly to `typedef` from C++, there is a function to test if a configuration is valid, a function that performs the main backtracking part of the algorithm, and another function that wraps the algorithm, offering the user an easy interface to call the N-Queens algorithm.

The algorithm presented in listing 10.2 is a simple algorithm that works well for run-time. We need to see how we can transform this algorithm to also work at compile-time. The astute reader may have noticed in the code the `[rtct]` modifiers applied to the three functions, making them ready for both run-time and compile-time. Having these modifiers will make the entire

```
fun doTest {
    var[ct] solutionsCt = nQueens(4);
    prettyPrint(solutionsCt);
}
```

Listing 10.3: Invoking the N-Queens algorithm at compile-time

algorithm work at compile-time as well. There is no need for additional coding; the algorithm simply works at compile-time.

This shows how easy it is to write metaprograms in Sparrow. As the only extra effort from the programmer is to write the evaluation mode modifiers, we can conclude that writing metaprograms in Sparrow requires essentially the same effort as writing regular run-time programs.

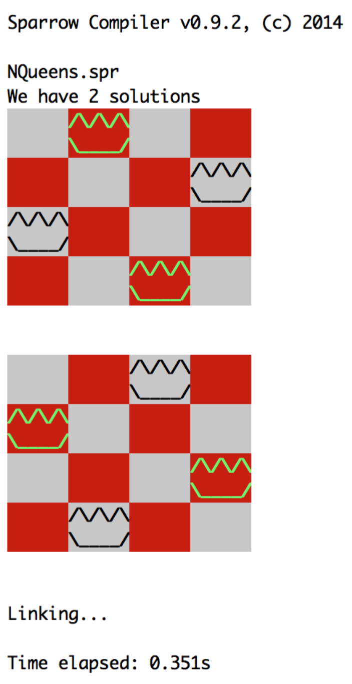
To execute the N-Queens algorithm at run-time, the call to the `nQueens` function should be performed inside a run-time context, and to use the algorithm at compile-time, we need to invoke the function in a compile-time context. Listing 10.3 presents a possible way to invoke the algorithm at compile-time. We create a compile-time variable that will get initialized with the result of calling the `nQueens` function. As the variable is compile-time, its initialization is performed in a compile-time context, therefore the `nQueens` function is invoked at compile-time.

Comparing this implementation to Dubrov's implementation of the same problem [Dub13], hyper-metaprogramming is clearly much easier to use than C++ template metaprogramming:

- our solution contains 25 lines of code; the C++ version contains about 200 lines of code for the actual algorithm
- the syntax of our code is the same as what run-time code would use; in C++ the syntax for the compile-time implementation is very different from the syntax used to write run-time code; the same applies to code semantics
- our N-Queens algorithm employs exactly the same code for both run-time and compile-time invocations; in C++ one must write different algorithms for run-time and for compile-time

This proves that in a language supporting hyper-metaprogramming, writing metaprograms that perform computations at compile-time is easier than in C++.

To test whether hyper-metaprogramming can perform more than simple numerical computations, we indulged ourselves in *having fun* with the results of the N-Queens algorithm. This way, we pretty-printed the solutions to the console (the actual compilation output) with colors. To draw the table, we read for each type of square (black or white, with or without queen) the characters to be displayed from an external file. Using ANSI colors in the input file, we were able to produce outputs similar to the one depicted in fig. 10.1. As can



```
Sparrow Compiler v0.9.2, (c) 2014

NQueens.spr
We have 2 solutions
```

```
Linking...

Time elapsed: 0.351s
```

Figure 10.1: The output for computing N-Queens for $N = 4$, with pretty print on a colored terminal (with ANSI escape codes)

be seen from the picture, these results are obtained at compile-time. (The `prettyPrint` function used to produce these results can be found in the listing from appendix C).

In C++ it is impossible to read external files, to format strings, or to write strings to the console (other than errors or warnings) at compile-time.

10.3 Compilation duration

This section features a series of experiments to test whether the compilation time increases excessively with the use of hyper-metaprogramming.

10.3.1 Hyper-metaprogramming vs. C++ metaprogramming

We use the N-Queens problem to compare the compilation time between a Sparrow version that uses hyper-metaprogramming and possible compile-time implementations in C++.

The code for the Sparrow version is the one presented in listing 10.2. The invocation of the algorithm is the one presented in listing 10.3, meaning that we will also be measuring the call to the `prettyPrint` function. This function will open an external file, read its content, and print the solutions nicely formatted, as shown in fig. 10.1.

For the C++ implementation, we test the exact N-Queens version that Dubrov presented [Dub13]. This version computes the solutions for the N-Queens problem with the help of template metaprogramming [AG04; VJ02], using Boost MPL library [GA]. The results of the algorithm are encoded in a type. Although the code for printing the solutions is generated at compile-time, this version does not format, nor print anything at compile-time; it also does not read files.

In addition to Dubrov's implementation, we also implemented a pseudo-solution for the N-Queens problem using the generalized constant expressions (in short `constexpr`) feature, introduced in C++ 11 [Cpp11]. This feature allows the compiler to perform computations at compile-time, with basic functions, without the help of templates.

With this feature, a C++ programmer can declare simple functions with the `constexpr` specifier, that can be executed at compile-time. If, when invoking the function, all the arguments are compile-time constants, the function will be evaluated at compile-time.

The 2011 C++ standard [Cpp11] imposes some restrictions on the `constexpr` function, limiting the functionality that can be implemented with this feature. Some of the most important limitations are:

- only simple objects can be used in conjunction with the `constexpr` feature; one cannot create dynamic structures like `vector` or `list` that contain memory allocations
- `constexpr` functions can only contain a return statement; no other computations, nor local declarations are permitted inside these functions, not even local variables
- no control structures are allowed inside `constexpr` functions

Although performing computations at compile-time is much easier with the help of the `constexpr` feature, currently only pure functions can be used with `constexpr`. Also, as memory allocations are not allowed, it is impossible to create structures to hold results of arbitrary length.

The 2014 version of the C++ standard (nicknamed C++ 14) [Cpp14] relaxed some of these restrictions, e.g., by allowing variables to be used inside functions, but memory allocations are still forbidden.

This means that we cannot implement with the help of `constexpr` feature (neither with C++ 11 nor C++ 14) a variant of the N-Queens problem that actually returns all the solutions of the problem for a given number of queens. Still, to be able to compare the compilation time using the `constexpr` feature, we implemented an algorithm that only returns the number of solutions and a single valid solution; the valid solution is encoded inside a 32-bit integer. The `constexpr` variant performs fewer computations than our hyper-metaprogramming version: it does not compute all the solutions, it does not store them, it does not print them (it does not generate any code for print-

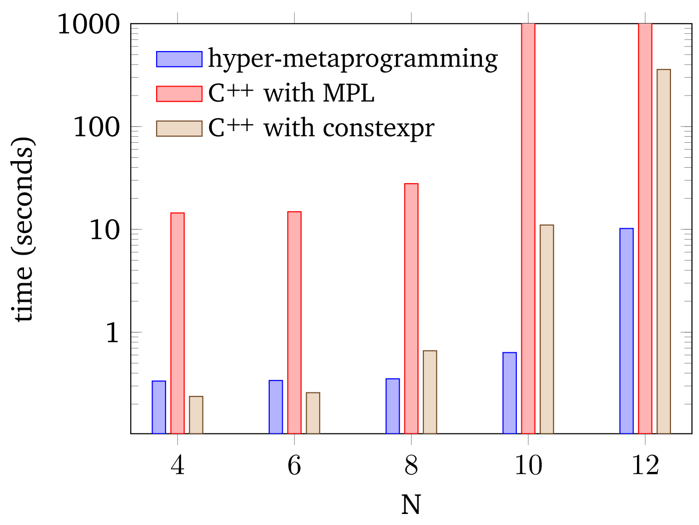


Figure 10.2: Compilation times for various N-Queens implementations

ing it), and it does not consult any files at compile-time. The listing for the `constexpr` version can be found in appendix D.

Figure 10.2 presents the compilation times for the N-Queens problem, for different numbers of queens, for the hyper-metaprogramming version in Sparrow and the two C++ versions. The hyper-metaprogramming version performs better than Dubrov's MPL implementation and the `constexpr` implementations.

The C++ MPL implementation proposed by Dubrov performs poorly in comparison with the rest of the implementations; for $N = 10$ and $N = 12$ the compilation was terminated by hand after 20 minutes, as this version cannot closely compete with the rest. It performs poorly even for the $N = 4$ case where only 2 solutions are possible from the space of 256 possibilities.

If the number of queens is small ($N = 4$ and $N = 6$), the amount of compile-time computations is limited for the hyper-metaprogramming and the `constexpr` cases. Therefore, the measurements roughly show the difference in compilation time between the Sparrow compiler and Clang C++ compiler. Here, the C++ compiler has a better start. Nonetheless, for larger values of N the compilation time of the `constexpr` version grows quicker than the compilation time of the hyper-metaprogramming version.

Compared to all the other metaprogramming versions, besides being faster, the hyper-metaprogramming version also does more: it computes all the solutions at compile-time, it loads an external file at compile-time, and it pretty-prints the results to the standard output at compile-time.

```
fun computeAtCtPrintAtRt {
    var[ct] solutionsCt = nQueens(4);
    var solutions = solutionsCt;
    prettyPrint(solutions);
}
```

Listing 10.4: Sparrow code for computing the N-Queens solutions at compile-time and printing the results at run-time

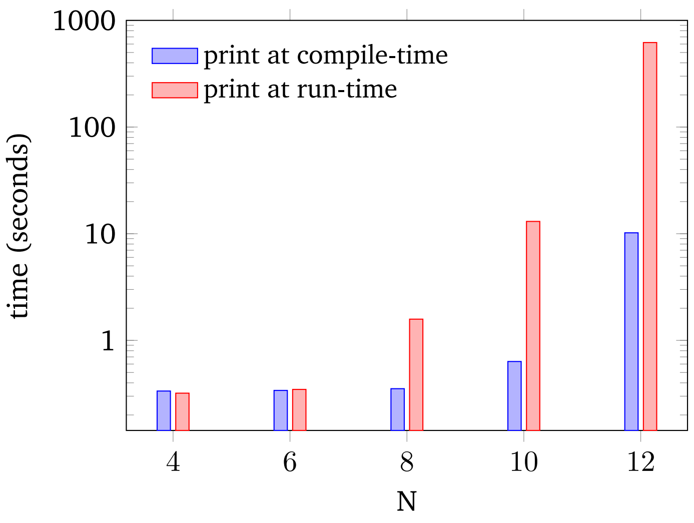


Figure 10.3: Compilation times for printing the N-Queens solutions only at compile-time, and only at run-time

10.3.2 Evaluation and generation costs

Intuitively, one might think that the compilation time of a program that uses hyper-metaprogramming consists of a base compilation time (the time to compile the program if it did not have any metaprograms), and the metaprogram execution time (how much time the compiler spends in executing the metaprogram). However, in certain situations there are other costs associated with hyper-metaprogramming, that can also be significant. Such are the costs involved in the evaluation of simple expressions at compile-time, and in generating code for run-time.

For example, let us slightly change our N-Queens implementation to print the solutions at run-time. Listing 10.4 shows the Sparrow code necessary for computing the solutions at compile-time and printing them at run-time.

Because printing at compile-time requires some extra effort at compile-time, we would expect the compilation time to be higher in the first case. However, it turns out that printing the solutions at run-time takes much more com-

```
fun[rt] ctorFromCt(src: Vector ct) {
    this.ctor();
    this.reserve(ctEval(src.size()));
    for[ct] ( el = src.all() )
        this.pushBack(ctEval(el));
}
fun[rt] ctorFromCt(src: Array ct) {
    var[ct] size = src size;
    this ctor size;
    for[ct] ( i = 0 .. size )
        at(ctEval(i)) = ctEval(src(i));
}
```

Listing 10.5: Default implementation of the `ctorFromCt` constructors for `Vector` and `Array`

pilation time than printing the solution at compile-time, as shown in fig. 10.3.

The justification is in the need to convert the solutions from compile-time data to run-time data. By default, to perform this conversion, the `ctorFromCt` constructor will be invoked for the `Vector` class (which is the type of the solutions), which will in turn call `ctorFromCt` for the `Array`. The default implementation for these constructors is shown in listing 10.5. Although these implementations are written in such a way as to accommodate all possible movements of data inside `Vector` and `Array`, they are not particularly efficient for our problem.

If we consider the generated code, for 10 queens and 724 solutions, the compiler will generate 11664 new lines of code, by a simple calculation. For each compile-time `Vector` or `Array` object the compiler has to generate a new `ctorFromCt` function; in total 725 such functions. This is because `ctorFromCt` is a generic function, not a regular function. In Sparrow, a generic function is similar to a function template in C++. For each function call, the compiler will create a new instance of the function, will semantically check it, and will generate code for it.

If we counted the number of times a compile-time expression is evaluated (see section 4.2 for more details), we would find that the compiler actually invokes the compile-time evaluation mechanism 43443 times, if we were converting all of the 724 solutions for 10 queens[1]. For each such evaluation, the compiler will generate a function inside the compile-time execution environ-

[1]There are two nested `for[ct]` loops. Inside a `for[ct]` structure with N iterations, the compiler invokes the evaluation mechanism $2+3N$ times just for the control structure, ignoring the body of the loop. (Once to evaluate the given range expression, $N+1$ times to check if the range is empty, N times to get the current value from the range and N times to move to the next value in the range). Inside `Vector`'s constructor we have one evaluation outside the loop and one inside the loop. Inside `Array`'s constructor—called from the `Vector`'s loop—we have one evaluation outside the loop and two evaluations inside the loop.

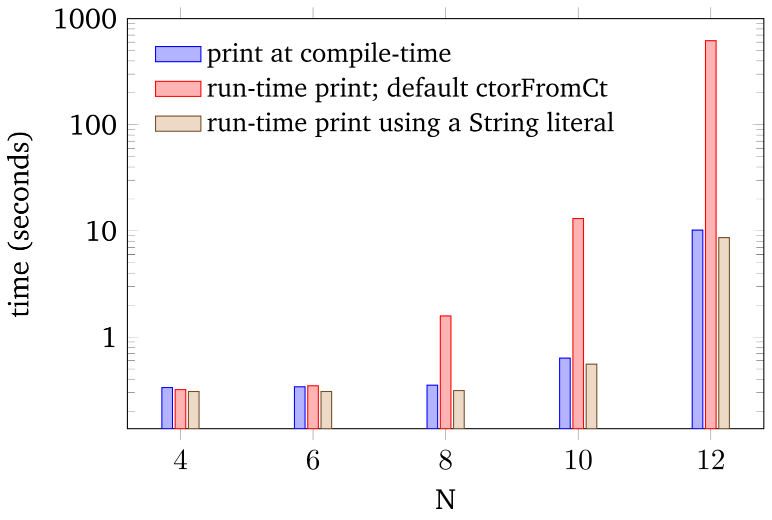


Figure 10.4: Compilation time for different solutions for printing the results; one prints the result at compile-time, one prints the result at run-time using the default `ctorFromCt`, and one prints the solution at run-time by applying the `String` transformation

ment, and then execute it using the internal interpreter. Even if the expressions to be evaluated are relatively trivial, taking into account the high number of evaluations that need to be performed, the reader can get a sense of how much extra effort is required from the compiler.

Luckily, there are ways to avoid this overhead. We can reduce both the code to be generated, and the number of required compile-time evaluations by applying techniques often used for serializing/deserializing objects. In our case, instead of generating code for a `Vector` of `Array`s, we can pack the solutions into a string. Converting only one `String` from compile-time to run-time is relatively cheap. We only need some extra work to pack the results into a single string at compile-time, and unpack the results from the string at run-time. If we encode a queen position into one char (byte), and use an additional chars to store the number of queens, we will use 7241 characters to store all the solutions for 10 queens.

The reader can find in appendix C the `toStringBuffer`, `fromStringBuffer`, and `toRt4` functions that convert the solutions from compile-time to run-time using a `String` literal. Analyzing this implementation shows that, except for the functions presented there, no new code is generated. Also, there is only one compile-time evaluation invoked: the one for the actual `String` literal.

We set up an experiment to measure whether the string workaround performs better in practice. The results are presented in fig. 10.4. Moving the data through a `String` literal makes it possible to decrease the compilation time below the time required to compile the version in which the solutions are

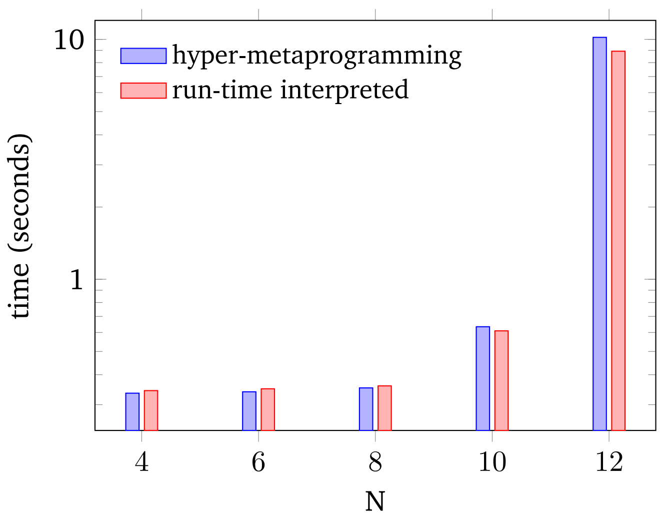


Figure 10.5: Compilation time of the N-Queens problem implemented in Sparrow using hyper-metaprogramming, compared with the execution time of the same problem at run-time

printed at compile-time.

We have shown here that in some cases, additional costs to move results of a metaprogram to run-time may be more substantial than the actual computation itself. However, these additional costs can be avoided by carefully implementing the translation from compile-time to run-time, and we have presented a technique that does that.

Generalizing the results obtained in this section, one can say that performing computations at compile-time is cheaper than code generation. This might explain why traditional metaprogramming systems that focus on code generation are slow. Using hyper-metaprogramming for performing computations at compile-time can speed-up the compilation time of metaprograms.

10.3.3 Hyper-metaprogramming vs. run-time execution

So far, we only measured the compilation time. But how does the compilation time compare to the time taken to execute the same algorithm at run-time?

We set up an experiment and compared the compilation time of the N-Queens problem in Sparrow running at compile-time with the execution time of the same problem, also implemented in Sparrow. Not only are the compared algorithms the same, but the source code for them is identical; we used the code from listing 10.2 in both cases, changing only the context from which the algorithm was called. As previously discussed, in Sparrow, this code reuse can be achieved by applying the `[rtct]` modifier to our functions.

The generated run-time version was the raw LLVM bitcode produced by generating the code; no optimizations were applied. To execute the code, we used the LLVM's `lli` interpreter. Note that this setting typically performs worse than actually optimizing the code and compiling it into a native executable. Nevertheless, we have chosen the raw bitcode version as it is similar to what the compiler uses for compile-time execution.

Figure 10.5 shows the execution speed of the N-Queens algorithm both at compile-time and at run-time. As can be seen from the picture, the differences in execution times are relatively small.

We expected to get different timing results for $N = 4$ because each version performs few N-Queens computations, and the time is spent on compiling the entire program (hyper-metaprogramming case) and initializing the interpreter (run-time case). However, by coincidence, we obtained similar results.

Ignoring the $N = 4$ starting point, it can be seen that the compilation time for the hyper-metaprogramming version grows in the same manner as the execution time for the run-time version. The run-time case is slightly faster for larger values of N, but the difference is not significant. Therefore, we can safely say that hyper-metaprogramming performs similarly to the run-time interpreted version of the same algorithm.

The Sparrow implementation that we used does not apply any optimizations for the compile-time code, even though this is theoretically possible. During a traditional compilation, the compile-time execution engine is invoked many times for operations that are cheap to perform. Optimizing such computations will take more than actually executing the not-optimized versions of the computations, making the optimization not feasible for such cases. As the compiler cannot easily determine which are the computations that can benefit from optimizations, it currently does not perform any optimizations. This can be improved in the future.

This experiment proves that there are no (substantial) abstraction costs for using hyper-metaprogramming to perform computations, compared to traditional run-time execution.

10.3.4 Experimental setup

All the experiments were performed on a MacBook Pro Retina (late 2013), 2GHz Intel Core i7, 8GB RAM, Flash Storage, MacOS X 10.9.1. The C++ versions (MPL and `constexpr`) were compiled with Apple's Clang version 5.0 (clang-500.2.79, based on LLVM 3.3svn). The hyper-metaprogramming version uses the latest 0.9.2 development version of the Sparrow reference compiler, using LLVM 3.3. The interpreter for the run-time version of N-Queens (`lli` program) belongs to the same LLVM 3.3.

The C++ programs were compiled with the flags `-std=c++11 -ftemplate -depth=4096`, without linking, producing just object files.

Similarly, the Sparrow hyper-metaprogramming and run-time versions were compiled with the `--simple-linking` flag, indicating to the compiler that it should produce an LLVM bitcode file as an output, without any optimizations and traditional linking.

To execute the run-time version, the LLVM's `lli` interpreter was used without any additional arguments.

For all versions, when performing measurements, the output of the compilation/execution was redirected to a file.

10.4 Conclusions

In this chapter we explored the implications of using hyper-metaprogramming for executing computations at compile-time in terms of ease of use and compile-time execution speed. In our analysis we used the N-Queens problem as the required computations increase exponentially, and also because we can easily compare our results with the findings of Dubrov [Dub13].

As a direct consequence of the hyper-metaprogramming definition, writing metaprograms is as easy as writing regular programs. There is no special syntax, no special semantics, and no special idioms to be used in order to write arbitrarily complex metaprograms. Moreover, one can use the same code for both run-time and compile-time. If a problem can be solved at run-time, then the same algorithm can be ported to compile-time with minimal effort (by simply adding the right modifiers). In particular, implementing the N-Queens problem at compile-time is trivial.

We want to stress the fact that hyper-metaprogramming can be used to perform arbitrary computations at compile-time, provided that the input data is available at compile-time. To illustrate this, not only did we implement the N-Queens problem, but we also showed that one can read from files at compile-time, format strings, and print the strings to the console using colors.

Throughout a series of experiments we also showed that hyper-metaprogramming, as it is implemented in Sparrow, is fast. Compared to C++ implementations (MPL and `constexpr`), our approach is able to achieve much faster compilation times. This is especially visible if there are large amounts of computations to be performed (large N values for our problem). Moreover, we compared the compile-time version of the N-Queens problem to a run-time version of the same problem; we used the LLVM bitcode interpreter to execute the run-time code. The experiments show that running the algorithm at compile-time takes roughly the same amount of time as executing the algorithm at run-time. Therefore, the abstraction costs for hyper-metaprogramming are low.

The chapter also explores some of the implications of moving results from compile-time to run-time. We showed that for large problems, moving the results from compile-time to run-time can be costly in terms of compilation time. In our example, it was cheaper to perform computations at compile-time than generating code for moving the same computation to run-time. We presented a technique one can employ to reduce the compilation time for moving results from compile-time to run-time.

All our experiments indicate that hyper-metaprogramming can be an appealing technique for performing computations at compile-time. It is easy, fast, and flexible.

If you give a person a fish, he can eat for a day. If you teach a person to fish, he can eat his whole life long. If you give a person tools, he can make a fishing pole—and lots of other tools! He can build a machine to crank out fishing poles. In this way he can help other persons to catch fish.

G. Steele

CHAPTER 11

Embedding other languages inside Sparrow

It is well known that the closer the concepts used in programming are to the programming domain, the easier the programming task becomes. In this regard, Domain-Specific Embedded Languages (DSELs) play an important role: they allow the programmer to work with abstractions specific to a domain in a more general-purpose language. This chapter presents an approach for embedding domain-specific languages (DSLs) [CE00] into a general-purpose host language, thus enriching the host language with the features of the embedded language. In this process the syntax of the embedded language is kept intact.

A Domain-Specific Embedded Language (DSEL) can be added into a host language in two main ways: by providing explicit compiler support, or by implementing the DSEL as a library feature. As extending compilers is often a cumbersome process, the second option is preferred whenever possible. The second option is also more flexible, as more than one DSEL can be added to the host language without requiring any change to the compiler. However, implementing DSELs purely as a library feature requires special support from the programming language: it must have good support for metaprogramming, and the compiler must allow applications to interact with its internal structures. In practice, a limited number of programming languages fulfill these requirements (e.g., Template Haskell [JS02], Racket [Fla12], MetaOCaml [CTHL03]).

The hyper-metaprogramming feature makes Sparrow a good candidate for a host language that allows the creation of DSELs inside it. We explore here the possibility of implementing DSELs in Sparrow, fully parsing the syntax of the embedded language, without changing the syntax of the Sparrow language.

As a proof-of-concept, we set out to embed a subset of the Prolog language [DM88] into the Sparrow programming language. One might argue that Prolog is a general purpose language and should not be considered domain-specific—still, for our purposes we want to embed a language that is rather complex, and whose semantics are very different from imperative languages.

Our presentation focuses on the actual process of embedding a language, and does not showcase a state-of-the-art implementation of Prolog. We are interested in finding the answer to the following question: how easy is it to use Sparrow's support for embedding a language, for a person that knows how to build a compiler for that specific language?

Not only do we achieve this in a simple and natural manner, but logic programs can actually be executed faster if they are translated into an imperative framework.

11.1 Overview and design

We want to be able to represent in Sparrow the Prolog code from listing 11.1. This is a simple code that contains some facts (`children`, `gender` predicates), some predicates that operate on the facts (`male`, `female`, `parent`, `father`, `mother`), and a recursive predicate (`ancestor`). Also, as an example of numeric computation, we would like to represent in Sparrow the Fibonacci code from listing 11.2. For the purpose of this thesis we will not deal with other Prolog features like lists, negations, or metalogic.

To add logic programming support in Sparrow, we want to be able to operate on different levels:

1. Using Sparrow syntax, we need to be able to write programs to be executed as if they were written in Prolog, using a logic execution engine.
2. We need to be able to pass an actual Prolog code in some kind of quotation mechanism in Sparrow source code; this code must then be translated to the equivalent Sparrow code and interpreted by the compiler.
3. An antiquotation mechanism [Mai07; JS02] must be implemented in the embedded Prolog code, so that it can interact with the rest of the Sparrow program.
4. Just like importing a Sparrow source file into an existing source file, we want to be able to import a Prolog file in a Sparrow source file. This would provide greater convenience to the user, as it can use existing Prolog source files directly with the Sparrow compiler.

On all of these levels, we need to have proper error reporting. If for example the user has a syntax error in the Prolog code, the compiler must report an error indicating the location of the error in the Prolog code; similarly, a semantic error needs to be associated with the location in the code from which it arises. The error messages need to be meaningful for the user, using terms

```
children(sam, mary).
children(denise, mary).
children(sam, frank).
children(denise, frank).
children(frank, garry).

gender(frank, male).
gender(sam, male).
gender(mary, female).
gender(denise, female).
gender(garry, male).

male(P) :- gender(P, male).
female(P) :- gender(P, female).

parent(P, C) :- children(C, P).

father(F, C) :- children(C, F), gender(F, male).
mother(M, C) :- children(C, M), gender(M, female).

ancestor(A, C) :- children(C, A).
ancestor(A, C) :- children(C, P), ancestor(A, P).
```

Listing 11.1: Example of Prolog code that we want to write in Sparrow

```
fib(0,0).
fib(1,1).
fib(X,Y) :-
    X2 is X-2, fib(X2, Y2),
    X1 is X-1, fib(X1, Y1),
    Y is Y1+Y2.
```

Listing 11.2: Fibonacci computation code in Prolog

appropriate for logic programming (as opposed to yielding Sparrow-specific messages).

Sparrow's hyper-metaprogramming appears flexible enough for us to impose a series of constraints in our process of implementing a domain specific embedded language in Sparrow:

- No logic programming feature should be implemented as a compiler feature
- After providing some general support for adding DSELs in the language, all the DSELs should be implemented entirely as library features
- In the process of adding support for DSELs we are not allowed to change the syntax of the Sparrow language beyond the extension points pro-

vided by the language/compiler (e.g., in contrast to Template Haskell [JS02] that changes the syntax of the original language to add support for quotation and antiquotation).

In order to allow a compile-time implementation of an arbitrary DSEL inside Sparrow, we need to make sure that the language and the compiler have the appropriate support. First we need to be able to perform computations at compile-time; hyper-metaprogramming is the language feature that provides just that. Then, we need to be able to inject code into the Sparrow compiler from a compile-time metaprogram.

We now turn our attention to the process of transforming the input code of the DSEL into something that the Sparrow compiler can process.

To be able to integrate a program written in language $\mathcal{L}$ into the Sparrow programming language, we need to apply at compile-time a function of the form:

$$compile_{\mathcal{L}} : String \to AST_{compiler} \tag{11.1}$$

This function takes as a parameter the input program to be interpreted, and generates the AST (abstract syntax tree) structure in the compiler used to represent this program. As with regular compilers, this function can be divided into two main components:

$$\begin{aligned} compile_{\mathcal{L}} &= generateCode_{\mathcal{L}} \circ parse_{\mathcal{L}} \\ parse_{\mathcal{L}} &: String \to IR_{\mathcal{L}} \\ generateCode_{\mathcal{L}} &: IR_{\mathcal{L}} \to AST_{compiler} \end{aligned} \tag{11.2}$$

The $parse_{\mathcal{L}}$ and $generateCode_{\mathcal{L}}$ functions correspond to the frontend and the backend parts of a traditional compiler, respectively. The $parse_{\mathcal{L}}$ function returns an intermediate representation of the source program. This intermediate representation is specific to the language $\mathcal{L}$. This function is also responsible for semantically checking the input code; if there is an error in the source program, it needs to be reported at this stage.

As compilers typically do, we can further split the $parse_{\mathcal{L}}$ function into several stages: tokenization, syntactic parsing, and semantic checking. However, for our current purposes, the way the parsing for each DSEL is implemented is irrelevant.

The intermediate representation resulting from $parse_{\mathcal{L}}$ is then taken by the code generator $generateCode_{\mathcal{L}}$, which will generate the corresponding Sparrow code by injecting it directly into the compiler.

We focus on the particularities of implementing these as a compile-time library feature in the Sparrow programming language instead of providing detailed information on how to build a compiler.

```
fun children(child, parent: @LStr)
     = child /=/ "Sam"    && parent /=/ "Mary"
    || child /=/ "Denise" && parent /=/ "Mary"
    || child /=/ "Sam"    && parent /=/ "Frank"
    || child /=/ "Denise" && parent /=/ "Frank"
    || child /=/ "Frank"  && parent /=/ "Gary"
    ;
fun gender(name, g: @LStr)
     = name /=/ "Frank"   && g /=/ "male"
    || name /=/ "Sam"     && g /=/ "male"
    || name /=/ "Mary"    && g /=/ "female"
    || name /=/ "Denise"  && g /=/ "female"
    || name /=/ "Gary"    && g /=/ "male"
    ;
fun male(name: @LStr)             = gender(name, "male");
fun female(name: @LStr)           = gender(name, "female");
fun parent(parent, child: @LStr) = children(child, parent);
fun father(name, child: @LStr)
     = children(child, name) && gender(name, "male");
fun mother(name, child: @LStr)
     = children(child, name) && gender(name, "female");
fun ancestor(a, c: @LStr): Predicate {
    var parent: LStr;
    return children(c, a)
        || children(c, parent) && rec(ancestor, a, parent);
}
```

Listing 11.3: Sparrow code corresponding to the Prolog code in listing 11.1

```
if ( father("frank", "denise")() )
    cout << "Frank is the father of Denise." << endl;
var a: LStr;
var pred = ancestor(a, "denise");
while ( pred() )
    cout << a.get() << " is an ancestor of Denise" << endl;
```

Listing 11.4: Calling logic code from Sparrow

11.2 Implementation details

11.2.1 Logic execution engine

If we want to embed logic programs into Sparrow, then for any supported logic program we must have an equivalent Sparrow program. We must be able to represent in Sparrow (using Sparrow syntax) all the concepts used in logic programming and we must also provide an execution engine that conforms with the execution model of logic programming.

Similar to the work of Naik [Nai10] for C++, we implemented a logic programming framework in Sparrow. We represent the Horn clauses of a logic program as functions that return predicates; these predicates implement the logic unification algorithm to test logic expressions and generate new values based on logic inference. If a predicate is called multiple times with one or more unbound variables, it can generate multiple output values; this is implemented using a basic coroutine mechanism.

The Sparrow code corresponding to the Prolog code from listing 11.1 can be found in listing 11.3. Listing 11.4 shows how these predicates can be invoked to test a certain proposition and to generate output values from a predicate.

The `||` operator is overloaded to implement logical disjunction, `&&` implements logical conjunction, and the user-defined `/=/` operator represents the equivalence/unification operation. The `LStr` type is a synonym for `LRef(String)`, and represents a logical reference to a string value; all logic programs should operate with such logical references.

Recursive behavior is implemented with the help of a special function `rec` that takes the name of the recursive function to be called and the list of arguments. This is needed to avoid infinite recursion while evaluating the function. Note that with this model, the function is called initially to generate a predicate, that can in turn be called later to actually perform the computations corresponding to the logic execution. As the values of the arguments are not bound/computed when generating the predicate, there is no way to stop infinite recursion. The `rec` predicate just delays the evaluation of the predicate until the execution time.

The most important concept here is the difference between calling a function to generate a predicate and the actual execution of the predicate that occurs at a later stage. If a user understands this difference, it will find that logic programming in Sparrow is straightforward, even when using Sparrow syntax. This is especially enhanced by Sparrow's support for custom user-defined operators.

11.2.2 Quotation mechanism; parsing Prolog code

According to the *no syntax change* rule (see section 11.1), we are not allowed to create a special delimiter for any DSEL. The embedded code needs to be passed using the existing syntax mechanisms of the language. The most appropriate syntax mechanism, available in all languages, are string literals. As in most languages, Sparrow literals are of the form `"some text"`[1].

Using string literals to encode DSELs has the inconvenience that in many programming languages one needs to escape certain characters, like `"` and

[1]Unlike many languages, Sparrow allows the string literals to be spread across multiple lines

`\`. Nowadays, languages (e.g. C++, D, Python) tend to provide raw string literals that do not recognize any escape characters inside them, and are more appropriate for encoding embedded programs. Sparrow has a similar string literal of the form `<{some text}>`. For the purpose of this thesis, to make the code more readable we will use only the first version of the string literal: `"some text"`.

Having a Prolog code represented as a string literal, triggering the parsing and compilation of Prolog code in Sparrow can be encoded as a simple compile-time function call:

```
compileProlog("...Prolog code...");
```

This is very similar to Template Haskell's splice syntactic form [JS02], but it does not require any syntax change to the language. Instead of writing something like `[compileProlog|...Prolog code...|]`, we convey the same information to the compiler in the form of a simple function call.

Because with hyper-metaprogramming executing compile-time functions is similar to executing regular run-time functions, parsing the Prolog code poses no special problems. We implemented a hand-made tokenizer that, given the range of characters from the input string, produces a range of tokens corresponding to the Prolog code. Afterwards, a parser takes a range of tokens and returns the Abstract Syntax Tree corresponding to the given code. As the grammar rules for Prolog are not very complex, we implemented the parser by hand, using a top-down, LL(1) approach [ALSU06]. Therefore, we have implemented the $parse_{\mathcal{L}}$ part, according to the terminology in section 11.1.

11.2.3 Compiler API

As discussed at the end of section 11.1, we need a method for injecting code into the Sparrow compiler. Instead of providing just one such function, we have implemented a series of functions that, when called at compile-time, can interact with the compiler. We called this set of functions the *Compiler API*.

Our goal was to provide to the Sparrow user the same functions that the compiler uses internally for creating, semantically checking, inspecting, and transforming AST nodes. This way, having exposed these functions, the user of the Sparrow language can manipulate the AST structures of a program just like the compiler does: the Sparrow user has very powerful metaprogramming capabilities through such an API.

Although we did not expose in the API all the compiler internal functions, we covered enough of them to allow one to inject complex code into the compiler. An excerpt from the Compiler API is presented in listing 11.5.

This API consists entirely of functions, identified by names annotated with the `native` modifier. The compiler will register custom functions with the corresponding names into the LLVM backend for the compile-time execution. Whenever an API function is called from within the language, the backend will

```
class[ct] CompilationContext {
    fun[static, native("$Meta.CompilationContext.current")]
        current(): CompilationContext;
    fun[native("$Meta.CompilationContext.evalMode")]
        evalMode: Int;
    fun[native("$Meta.CompilationContext.sourceCode")]
        sourceCode: SourceCode;
    ...
}
class[ct] AstNode {
    fun[native("$Meta.AstNode.location")]
        location: Location;
    fun[native("$Meta.AstNode.children")]
        children: Vector(AstNode);
    fun[native("$Meta.AstNode.context")]
        context: CompilationContext;
    fun[native("$Meta.AstNode.type")]
        type: AstType;
    fun[native("$Meta.AstNode.setContext")]
        setContext(context: CompilationContext);
    fun[native("$Meta.AstNode.computeType")]
        computeType;
    fun[native("$Meta.AstNode.semanticCheck")]
        semanticCheck;
    fun[native("$Meta.AstNode.hasError")]
        hasError: Bool;
    ...
}
fun[ct, native("$Meta.Feather.mkMemLoad")]
    mkMemLoad(l: @Location, e: AstNode): AstNode;
fun[ct, native("$Meta.Feather.mkIf")]
    mkIf(l: @Location, cond, thenC, elseC: AstNode): AstNode;
fun[ct, native("$Meta.Util.getParentDecl")]
    getParentDecl(ctx: CompilationContext): AstNode;
fun[ct, native("$Meta.Compiler.registerFrontendFun")]
    registerFrontendFun(ext, funName: StringRef): Bool;
...
```

Listing 11.5: Excerpt from the Compiler API as it appears in the Sparrow code

```
fun[ct] addFactorialFun(loc: Location, ctx: CompilationContext,
                        fName: StringRef): Bool {
  var idInt = mkIdentifier(loc, "Int");
  var idN = mkIdentifier(loc, "n");
  var one = mkIntLiteral(loc, 1);
  var ifCond = mkInfixOp(loc, "==", idN, mkIntLiteral(loc, 0));
  var ifThen = mkReturnStmt(loc, one);
  var recCall = mkFunApplication(loc, mkIdentifier(loc, fName),
      mkNodeList(loc, mkVector(mkInfixOp(loc,"-",idN,one))));
  var ifElse =mkReturnStmt(loc,mkInfixOp(loc,"*",idN,recCall));
  var body = mkLocalSpace(loc, mkVector(mkIf(loc, ifCond, ifThen
      , ifElse)));
  var factParam = mkSprParameter(loc, "n", idInt, nullNode);
  var factParams = mkNodeList(loc, mkVector(factParam), true);
  var factFun = mkSprFunction(loc, fName, factParams, idInt,
      body, nullNode);
  factFun.setContext(ctx);
  factFun.semanticCheck();
  return !factFun.hasError();
}
```

Listing 11.6: Example code that injects a Factorial function into the Sparrow compiler

invoke the corresponding implementation in the compiler. This way, one can use the compiler internals.

Injecting code into the compiler can be done by invoking the functions that create AST nodes. After the corresponding AST nodes are created, the user must set a context into which the AST structure needs to be inserted, and call `semanticCheck` on the root node—this will make sure that the entire AST structure is checked for correctness and will prepare it to be used by the compiler.

In listing 11.6 we present a code that injects the factorial function into the compiler, by using the Compiler API. The code has the disadvantage that it is verbose for a simple function like the factorial; however it constructs the AST nodes as the Sparrow compiler would, with all the associated information. This can be further improved using a library wrapper to allow the user to write less code. Also, the user can use the `lift` intrinsic to get the AST representation of Sparrow constructs.

Implementing this Compiler API is one of the most important steps in our process of embedding a language in Sparrow. It allows the library code to interact with the compiler, and inject into it the needed AST structures corresponding to the input source code.

Similar to the `addFactorialFun` function, we created a function that receives the Prolog intermediate representation, and generates the appropriate Sparrow AST representation inside the compiler. The function has the following signature:

```
fun[ct] genPrologCode(prologSystem: SparrowPrologSystem)
            : AstNode {...}
```

Setting the context of the resulting `AstNode` and semantically checking the node is performed by the caller.

This function corresponds to the $generateCode_{\mathcal{L}}$ function described in section 11.1. This represents the final component of our Prolog compiler implementation.

11.2.4 AST macros and the final form

Similar to Template Haskell [JS02] or MetaML [ST97] we implemented support for `lift` and `astEval`. A `lift` call takes a Sparrow expression and returns an `AstNode` object corresponding to the given code. Conversely, an `astEval` call will take an `AstNode` value as an argument and will replace itself with the indicated AST; the compiler will proceed as if the actual AST structure were in the place of the call.

We implemented `lift` and `astEval` as compiler intrinsics. The Sparrow compiler has basic support for adding intrinsics without changing the syntax of the language. Other examples of Sparrow intrinsics are: `sizeOf` (get the size of an expression), `ctEval` (evaluate an expression at compile-time), and `isValid` (test if an expression is valid, without issuing errors).

If we have a function `transform(node: AstNode): AstNode` that does some processing on a given AST node and returns another AST node, then the following construct is useful:

```
astEval( transform( lift( <expr> ) ) )
```

It provides support for transforming an existing Sparrow expression into some other Sparrow construct using a user-defined meta-function. To simplify the use of this construct, we implemented an extension to function declarations: *macro functions*. To define a macro function f, one needs to apply the `macro` modifier to the function declaration, and all the parameter types and the return type must be `AstNode`. When invoking f with the arguments e_1, ... e_n, the function application will be replaced with `astEval(` f(`lift(`e_1`)`, ..., `lift(`e_n`)`). Sparrow provides a relatively simple mechanism of extending declarations through the use of *modifiers*, without requiring any change in the syntax.

One of the advantages of using a macro instead of a normal function is that we can obtain all the properties of the AST nodes describing the parameters. For example, for each parameter we can obtain the location of the argument that was passed to the invocation and its compilation context.

With the help of macros, we can properly implement the `compileProlog` function:

```
fun[macro] compileProlog(sourceString: AstNode): AstNode {
  var loc = sourceString.location();
  if[ct] ( sourceString.nodeKind() != Kinds.nkSparrowExpLiteral )
    report("compileProlog should take a string literal", loc);
  else {
    sourceString.semanticCheck();
    var ctx = sourceString.context();
    var str: String = astEval(sourceString.clone());
    return handlePrologCode(str, loc, ctx);
  }
}
```

When invoking this macro the user is not required to pass the location and the compilation context of the Prolog code. They are acquired from the given argument. The way `compileProlog` is implemented requires the user to pass a string literal; for any other expression passed as argument an error is generated.

If this function is passed a string containing the code in listing 11.1 it will inject a code similar to the one in listing 11.3; the user can call the Prolog code in a manner similar to the one presented in listing 11.4.

Finally, the `handlePrologCode` function has the following implementation:

```
fun[ct] handlePrologCode(code: @String, location: Meta.Location,
        context: Meta.CompilationContext): Meta.AstNode {
    var errorReporter: CompilerErrorReporter;
    var lexer = mkLexer(code.all(), errorReporter, location);
    var parser = mkParser(lexer, errorReporter);
    var sps: SparrowPrologSystem = parser.parse();
    sps.semanticCheck();
    var res = genPrologCode(sps);
    res.setContext(context);
    return res;
}
```

The reader can see all the main components of the Prolog compiler in the code above.

Note that the macro mechanism described here is not actually needed to implement the main functionality of the embedded language compiler. Without macros, the user can simply call the `handlePrologCode` compile-time function, but in that case two additional parameters are required: location and compilation context. Macros serve as syntactic sugar and facilitate obtaining these two parameters from the given string literal passed as input.

11.2.5 Error handling

In the most basic form, an error message contains two parts: a textual description and a location to which the message refers. To be able to provide meaningful error messages we therefore need a method of generating errors with custom descriptions, and to track the location of every token in the input code. With these two elements, an implementer can create complex error reporting systems that are able to generate helpful and effective messages.

Sparrow provides several abstractions for handling location information. The `SourceCode` class represents a source code that is tracked by the compiler. The `Location` class defines a range of characters (line + column) in a source code. As the range of characters is represented by four integers (start line/column, end line/column) the manipulation of location information can be done easily.

In our case, we first need to store the location of the entire source code literal. Then, while tokenizing the Prolog code we also keep track of the location information of the individual tokens inside the string literal. By combining these two pieces of information we obtain an absolute location for each Prolog token that can be used for error reporting.

The compiler API defines several functions that, when invoked, generate user-defined errors, warning, and information messages. The following code exemplifies the use of some of these functions:

```
Meta.report("An error with custom text", loc);
Meta.report(Meta.Diagnostic(
    "This is a warning", loc, Meta.diagWarning));  // a warning
var[ct] diags: Vector(Meta.Diagnostic) = ...
Meta.report(diags);      // Report several diagnostic messages
Meta.raise();            // Stop compiling
```

By using these abstractions one can generate meaningful errors at the correct locations in the embedded code. For example, if we add `fib(sun, earth).` after the last line of the code from listing 11.2, with our implementation we obtain the following errors:

```
Fibonacci.pl(7:5-8) : ERROR : Type mismatch: cannot combine
        atom with number
> fib(sun, earth).
      ~~~
Fibonacci.pl(7:5-8) : See also atom expression:
> fib(sun, earth).
      ~~~
Fibonacci.pl(5:11-14) : See also number expression:
>     X1 is X-1, fib(X1, Y1),
            ~~~
```

These errors have the same form as the ones produced while compiling Sparrow code.

Even for our basic implementation the reported errors are context-aware. The complexity of the error reporting system can be extended indefinitely, and can be given access to complete contextual information; this is strictly a matter of compiler engineering.

11.2.6 Antiquotation

Antiquotation [Mai07; JS02] is the mechanism that allows embedded code to refer to code in the host language. In our case, we want Prolog code to be able to refer to Sparrow code. For example, if we have a Sparrow variable we want to be able to read its value; similarly we want to be able to invoke Sparrow functions inside Prolog for various computations.

The following code presents a trivial example of antiquotation in our implementation:

```
planet(P) :- orbits(P, sun).
planet2(P) :- orbits(P, $sunVar).
planet3(P) :- orbits(P, ${"sun"}).
planet4(P) :- orbits(P, ${getSun()}).
```

We use the `$` symbol to start an antiquotation. If there is a simple identifier after `$` then the identifier will be the antiquoted entity. An alternative is to have curly braces after `$` to precisely delimit the antiquoted entity.

When implementing antiquotation there are several major design decisions that need to be made. Do we want the same antiquotation syntax for every DSEL, or do we want each DSEL to have its own antiquotation mechanism? What kind of constructs from the host compiler should be referable in the DSELs? Who does the parsing of the antiquoted expression: the DSEL implementation or the host compiler?

To answer the first question, we opted to let the antiquotation syntax be defined by each DSEL implementation. If we were to have only one implementation of the antiquotation parsing in the host language, then we would not be sure that the syntax that we chose does not collide with the syntax of the inner language. For example, if we used the `$` sign to start an antiquotation as described above, then we could not implement a Perl DSEL, as the `$` notation would collide with Perl's syntax. Therefore we believe that implementing the antiquotation parsing in each DSEL is the best approach. The incurred cost is that we need to implement the antiquotation mechanism for each DSEL that we want to implement. However, with good software engineering practices we can factor out the common implementation of antiquotation as a stand-alone library feature so that we can use it in multiple DSELs.

Typically when we want to refer to Sparrow code from inside a DSEL, we want to be able to write expressions. However, as Sparrow is an imperative language, one can reason that not only expression, but also statements and declarations should be antiquotable inside DSELs. For example, one may want to have complete Sparrow functions in the embedded code. Despite losing generality, we only implemented the simplest case in which only expressions are antiquotable from the Prolog code, which makes more sense for this particular embedded language.

As previously stated, to parse the antiquoted entity we have two possibilities: implement the parsing in the DSEL or implement the parsing in the host language. We first started with a simpler implementation that does the parsing inside the DSEL. We only recognized variable names, literals, and simple function calls, and then at the code generation step we called the appropriate API function to create the correct kind of AST node. However this implementation turned out to be too limiting. As soon as we want to have more complex expressions with operators, parentheses, and nested function calls, the complexity of the implementation increases significantly. Therefore, in a second iteration, we exposed a compiler API function that is able to parse Sparrow expressions and generate the corresponding AST code, which is called by the DSEL implementation whenever an antiquoted term is encountered.

Although not completely generic, this antiquotation implementation can be used successfully for embedding Prolog code in Sparrow. The antiquotation mechanisms can be implemented in different ways for different embedded languages; we explored several alternatives for our Prolog proof of concept, making sure that Sparrow is capable of allowing antiquotation for an arbitrary embedded language.

11.2.7 Directly importing Prolog files

In addition to embedding Prolog code inside Sparrow, we want to allow the user to keep the Prolog code separated, and to invoke the Sparrow compiler to compile Prolog files. The compiler will then act as if it is natively capable of compiling Prolog source files.

In order to do that, we want to have the possibility of directly importing Prolog files from Sparrow code, similarly to importing other Sparrow files:

```
import "parents.pl";
```

To make this functionality available to the user, we implemented a mechanism through which one can register a compile-time function to handle imports of files with a certain extension. To import Prolog files, one needs to add the following code somewhere inside the program:

```
var[ct] regOk = registerFrontendFun(".pl", "handlePrologCode");
```

`registerFrontendFun` is an API function that will instruct the compiler to call the function with the name passed as the second argument to all the files that have the given extension. Here, `handlePrologCode` refers to the function described in section 11.2.4.

When the compiler tries to compile a file with the `.pl` extension, because it does not support it natively, it will search for all the registered functions in order to find a match; if it finds a function that can handle the extension, it will use it to compile the content of the file. First the compiler creates a location (in this case the location refers to the first line and column of the Prolog file) and a compilation context for the new file, it reads the content of the file, and then invokes `handlePrologCode`. As previously discussed, this function returns an `AstNode` corresponding to the content of the source file.

This mechanism ensures a seamless integration between Sparrow and Prolog code. From the user perspective, the Prolog code appears to be compiled natively.

11.3 Performance

We set out to test the execution performance of our Prolog DSEL implementation. We measured the execution time required to compute the Fibonacci sums and we compared the results with the SWI Prolog implementation. For the SWI Prolog implementation, we measured both the unoptimized and optimized compiled versions of the same program. In all three cases we used the code in listing 11.2. The Sparrow DSEL injects into the compiler a code that is similar to the one displayed in listing 11.7.

Our measurements were performed on a MacBook Pro Retina (late 2013), 2GHz Intel Core i7, 8GB RAM, Flash Storage, MacOS X 10.9.5. The Prolog DSEL implementation uses a modified version of the Sparrow reference compiler version 0.9.2, using LLVM 3.3. We used SWI Prolog version 6.6.0. The command line to compile the SWI Prolog versions of the program was: `swipl --stand_alone=true --goal="fib(`N`, X), write(X), halt." -L6g -o fib-swi.out -c Fibonacci.pl`, with the `-O` flag for the optimized version.

The results are shown in fig. 11.1.

The Sparrow DSEL implementation performed better than the unoptimized version of SWI Prolog, but worse than the optimized version. Although for the Sparrow version we used the standard LLVM optimizer flags, no logic programming specific optimizations were implemented. With this in mind, one needs to compare the Sparrow DSEL version with the unoptimized version of SWI Prolog. In this respect, our Prolog DSEL implementation outperforms the SWI Prolog implementation. The results obtained so far lead us to believe that a complete Prolog DSEL implementation, with multiple logic-specific optimizations added, would yield faster code than the one produced by popular Prolog implementations.

```
fun fib(p_1: @LInt, p_2: @LInt): Predicate {
    var l2_X2: LInt;
    var l2_Y2: LInt;
    var l2_X1: LInt;
    var l2_Y1: LInt;
    return p_1 /=/ 0 && p_2 /=/ 0
        || p_1 /=/ 1 && p_2 /=/ 1
        || l2_X2 /=/ (p_1 /-/ 2) && rec(fib, l2_X2, l2_Y2)
            && l2_X1 /=/ (p_1 /-/ 1) && rec(fib, l2_X1, l2_Y1)
            && p_2 /=/ (l2_Y1 /+/ l2_Y2)
        ;
}
```

Listing 11.7: What the code to be injected into the compiler for the Fibonacci code presented in listing 11.2 would look like

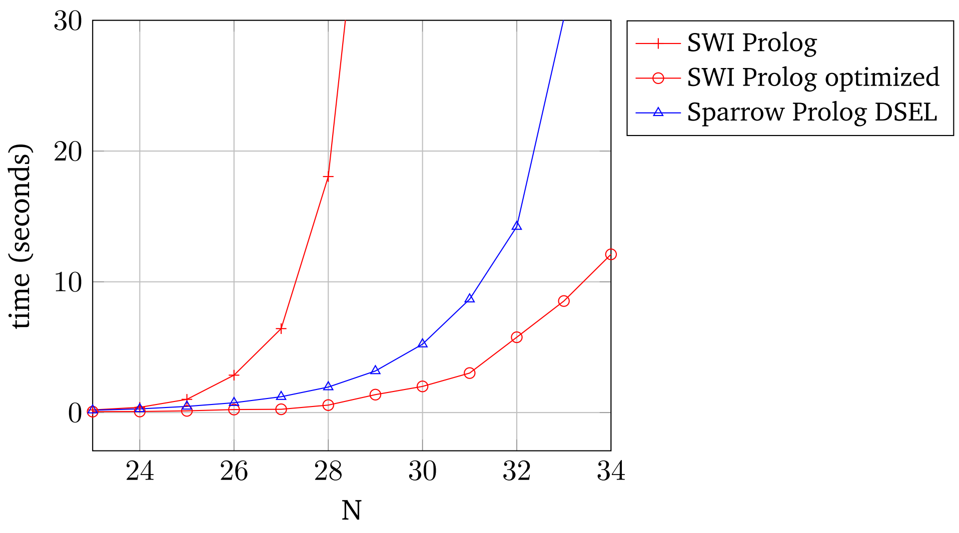


Figure 11.1: Execution times for the classic Fibonacci algorithm

```
fun fib(n: Int): Int {
    if ( n == 0 )       return 0;
    else if ( n == 1 ) return 1;
    else               return fib(n - 1) + fib(n - 2);
}
```

Listing 11.8: Imperative version of the Fibonacci algorithm (using recursion)

11.3.1 Generating native code

Although the code injected into the compiler (see listing 11.7) for our Fibonacci example (listing 11.2) looks somewhat similar to a regular recursive function, its execution is fundamentally different. As explained in section 11.2.1, when `fib` is called, no computation is actually performed; rather, a predicate is generated, that when invoked will actually perform the computations. When `rec` is actually invoked it will create a copy of the `fib` predicate and then invoke it. This way, while executing the fibonacci algorithm we will create a large number of predicates; each predicate will create its own set of variables. This way, most of the execution time is spent in copying the predicates and allocating memory for the logical variables.

With this in mind, we set out to find a method of transforming the generated code into a version that is closer to the traditional execution model in imperative programming (similar to the code presented in listing 11.8. This version would have stack variables instead of logic variables (which are implemented as shared pointers over optional values), regular function calls instead of creating predicate objects, assignment instead of unification, and regular arithmetic.

Before transforming our generated code, we need to check the conditions in which such a conversion is possible. As the reader might expect, not all logic algorithms have a straightforward imperative form. For our transformation to work, the criteria include, but are not limited to:

- all parameters must be either input-only or output-only
- the algorithm should never write into an input parameter
- the algorithm should never read from an output parameter
- the right-hand side of an **`is`** expression always evaluates to a value
- all the local variables can only be used once in an **`is`** expression
- there are no calls to other predicates except recursive calls to the same predicate

The Fibonacci code from listing 11.2 satisfies these conditions. The first parameter is an input parameter, and the second parameter is an output parameter; one never writes in the first parameter and never reads from the second. All the local parameters appear only once in the right-hand side of an **`is`** expression, and the right-hand side of all the **`is`** expressions are always entities that can evaluate to variables. Finally, the function only calls itself.

Note however that these conditions do not guarantee a perfect isomorphism between logic programs that satisfy them and the corresponding imperative versions. For example, in Prolog one can call the `fib` function with inverse input/output parameters like this: `fib(X, 1).` Although this call does not make much sense, it is perfectly legal in Prolog; the code from listing 11.8 however cannot be used this way. Therefore, the user should be careful when applying this transformation.

After checking the conditions for an arbitrary logic algorithm, we implemented a simple transformation that works with the subset of Prolog currently supported. The main points involved are summarized below:

- a native function is generated for the predicate, with the following transformations:
  - all the parameters and local variables will use native types instead of logical references
  - the input parameters are received by value, the output parameters are received by reference
  - **`is`** expressions between an input parameter and a value will become regular equality checks
  - **`is`** expressions between output parameters or local variables and expressions will become assignments
  - logical conjunction and disjunction operators will become simple boolean operators
- a predicate class is generated, wrapping the native function
- the logic function that returns the wrapper predicate is created

We implemented a version of this transformation that works with our Prolog DSEL; with this in place, our system is able to generate imperative code for logic algorithms that satisfy the previously mentioned conditions. For the Fibonacci code from listing 11.2 our system injects into the compiler the code from listing 11.9. The boolean-style of the code is preserved, but this time all the operations are native. The `=/` operator implements the assignment and returns **`true`**, making it possible to chain assignments in a logical expression.

With this transformation in place, we measured again the performance of our implementation. The results are presented in fig. 11.2. The Sparrow version that has the Fibonacci computed within a native function performs much better than the other implementations. In addition to SWI Prolog, we also compared the execution time with a program written in C++ that computes the Fibonacci sums in a recursive manner (similar to the code from listing 11.8). The C++ version is marginally better (by approximately 1.2%) than our version.[2]

[2]Interestingly, the difference in performance does not come from the fact that we have a wrapper around `fib_native`, but from the fact that we are returning the result as a reference—this way, we require some extra mem-stores and mem-loads that the C++ version does not require.

```
fun fib_native(p_1: Int, p_2: @Int): Bool {
    var l2_X2: Int = Uninitialized();
    var l2_Y2: Int = Uninitialized();
    var l2_X1: Int = Uninitialized();
    var l2_Y1: Int = Uninitialized();
    return p_1 == 0 && p_2 =/ 0
        || p_1 == 1 && p_2 =/ 1
        || l2_X2 =/ (p_1 - 2) && fib_native(l2_X2, l2_Y2)
            && l2_X1 =/ (p_1 - 1) && fib_native(l2_X1, l2_Y1)
            && p_2 =/ (l2_Y1 + l2_Y2)
        ;
}
class fib_pred_wrapper {
    fun ctor(p_1: @LInt, p_2: @LInt) {
        this.p_1 ctor p_1;
        this.p_2 ctor p_2;
    }
    fun (): Bool {
        if ( p_1.isNull() )
            return false;
        p_2 = Int();
        return fib_native(p_1.get(), p_2.get());
    }
    private var p_1: LInt;
    private var p_2: LInt;
}
fun fib(p_1: @LInt, p_2: @LInt): Predicate = fib_pred_wrapper(p_1, p_2);
```

Listing 11.9: What the code to be injected into compiler for the Fibonacci code presented in listing 11.2 would look like, using an imperative version of the algorithm

Not all logic algorithms can be transformed into imperative programs in this way, but those who can may yield very efficient native code. The entire process, from parsing Prolog code to generating native code, is implemented as a library feature, meaning that users can implement their own optimization steps without changing the compiler. For example, the Fibonacci code generation can be further improved by replacing recursion with a `while` loop.

11.3.2 Compilation time and memory usage

Although compiled code can increase the performance of the generated program, the user has to pay the cost of compilation. The total compilation time for our optimized Fibonacci example (Prolog code + test code) was approximately 1.97 seconds. From that, the time taken by the actual handling of the Prolog code was 0.48 seconds; the linking part takes about 0.57 seconds; the

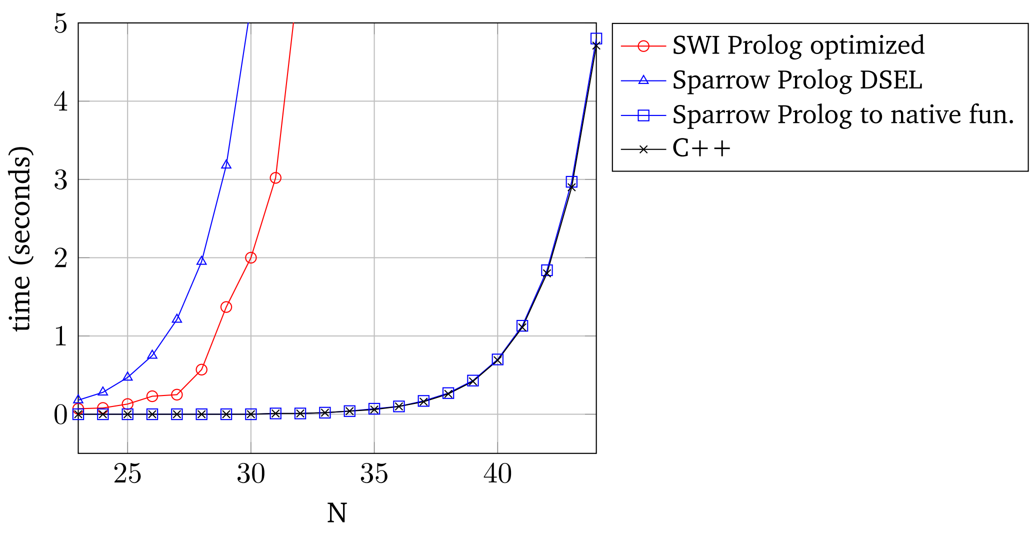


Figure 11.2: Execution times for the classic Fibonacci algorithm, optimized versions

rest (about 0.92 seconds) is taken up by the actual compilation of the Sparrow code involved. This can be correlated to the findings from chapter 10, that hyper-metaprogramming computations can be executed fast, but the overall compilation processes can be slow.

The maximum memory required by our running programs can be seen in fig. 11.3. The maximum memory needed grows exponentially for both our executing engine and the SWI implementation. Our execution engine takes more memory than the SWI Prolog implementation. The native codes generated by our optimization and the C++ native implementation consume very small amounts of memory that practically do not grow with the input size.

11.4 Conclusions

In this chapter we presented an approach for adding DSELs in the Sparrow programming language entirely as a library feature. We implemented a subset of the Prolog language as a DSEL, thus introducing logic programming into an imperative language. Although we changed the compiler to support a tighter integration with the Sparrow code, adding the DSEL did not require any specific change to the compiler; in particular, the compiler is not aware at any time that it is dealing with logic programming. Moreover, we did not change the syntax of the Sparrow programming language to add support for DSELs.

Even though we implemented only a subset of Prolog, our DSEL implementation can be considered complete. Our solution is based on an execution

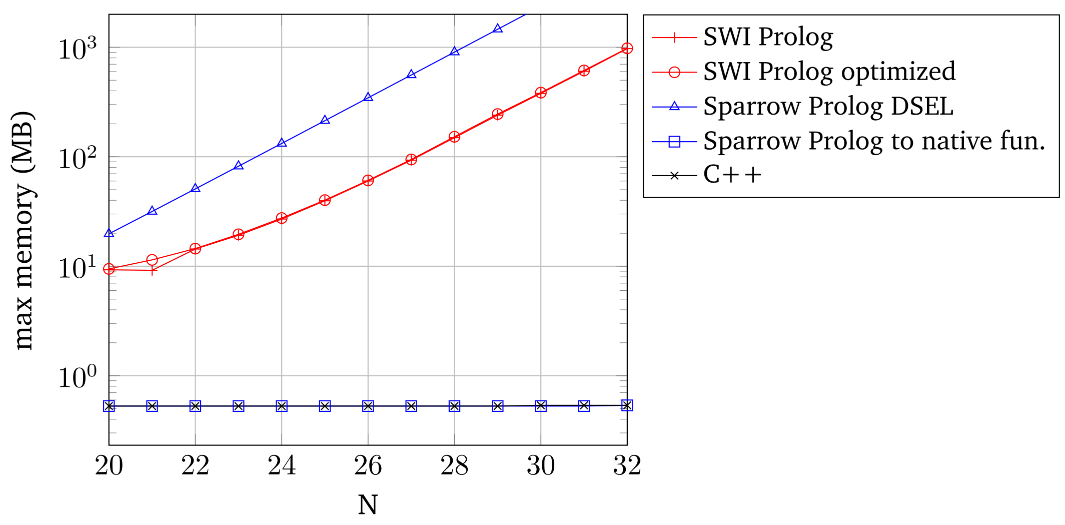


Figure 11.3: Maximum memory utilized

engine that is capable of running logic programs. The Prolog code is passed to a compile-time function that parses it and generates the corresponding AST program structures in the Sparrow compiler. This function is actually a macro-function, which constitutes a convenient way of providing context information to the DSEL metaprogram. A set of compiler intrinsics, called Compiler API, can interact with the AST, facilitating code generation and complex error reporting. Various antiquotation mechanisms can also be implemented at compile-time, allowing the embedded code to interact with the host Sparrow program. Apart from specifying Prolog code in Sparrow source files, we also provide support for importing Prolog source files directly.

We evaluated the performance of Prolog programs embedded into Sparrow, comparing it to the performance of the same code compiled with SWI Prolog. Our initial implementation, which does not contain any logic-specific optimizations, outperforms the unoptimized version generated by SWI Prolog. We also implemented an optimization that tries, whenever possible, to generate native imperative code from the Prolog specification; the code generated in this manner can be as fast as a native C++ implementation.

Implementing a Prolog DSEL in Sparrow turned out to be an easy task, thanks to the powerful hyper-metaprogramming support that allows writing metaprograms in the same manner as regular run-time programs. The ease of implementation, the natural integration of Prolog into Sparrow, and the performance results lead us to believe that the approach described in this chapter is well suited for building DSELs entirely as a library feature.

Allowing the creation of DSELs inside Sparrow increases the overall naturalness of the language. The users can work in the programming paradigm best suited for the programming task that needs to be accomplished.

PART III

Epilogue

I am tomorrow, or some future day, what I establish today. I am today what I established yesterday or some previous day.

J. Joyce

Chapter 12 Future work

The Sparrow programming language has come a long way; almost 10 years from the initial idea. Starting from *a better C++* and *make metaprogramming a feature easy to use* we evolved into creating a programming language that can be at the same time flexible, efficient, and natural. However, there is still a great deal of work ahead of us. We want to further advance the state of the art by improving our language and compiler. This chapter describes the main next steps we can undertake to create an even better programming language.

12.1 Improving the current compiler

During the creation of the Sparrow programming language and of the reference compiler, our main goal was to implement the most important Sparrow features, that would account for the most important use-cases. Even if this approach allowed us to showcase major Sparrow functionalities rather quickly, we neglected some of the features that other programming languages offer.

One example of a feature that is not fully implemented is subtype polymorphism [AC96; CW85; Pie02]. Apparently, in our Sparrow programs the need for subtype polymorphism is not as stringent as initially thought. In the cases in which dynamic behavior was needed we emulated this by using function objects; a function object can be considered to be a minimal interface with a single method.

Without subtype polymorphism, class inheritance is not particularly useful. Although the compiler has support for it, it acts more like a *composition* relationship than a *subsumption* relationship.

Other features that would be very welcome in the language are: exceptions, constant/immutable data, move semantics, better access control, atomic operations, co-routines, enumerations, and `switch` statements. Of course, improving and extending the standard library is always desired.

The current version of the compiler aimed at producing efficient code for the common use-cases; creating objects, using variables, calling functions, using basic control flows are as fast as in languages like C++ or C — there are no hidden costs. However, more can be done to improve efficiency when handling corner-cases. For example, we can implement move semantics, NRVO (Named Return Value Optimization)[1], data alignment and padding, atomic operations, or platform specific optimizations.

The naturalness can be further improved by creating user-friendly constructs in the language. Enumerations and pattern matching are two features that we find very useful. Adding these in a typical programming language would increase language complexity; whenever possible, we would like to add features like these through language extensions.

From an extensibility point of view, we would like to allow the user to define language features like enumerations and `switch` statements; the macro support may need to be extended to handle the introduction of these constructs in the language in a simple fashion. Also, allowing the user to define custom modifiers could prove to be a powerful method to allow extending the language.

12.2 Micro-compilers

Hyper-metaprogramming is a fundamental feature in the design of the Sparrow programming language. However, if we look carefully at the definition of hyper-metaprogramming, this feature is not bound in any way to a programming language. We can consider this a feature of the compiler more than a feature of the language. If hyper-metaprogramming were a feature more basic than the language itself, we pondered whether it makes sense to build a *micro-compiler* that is language agnostic, and only implements hyper-metaprogramming; this compiler could then be extended with compile-time *modules* to support various programming languages (e.g., Sparrow, C++, C).

If we were to construct a micro-compiler, we would do the following: First we would create a backend capable of handling hyper-metaprogramming requirements; this would be based on LLVM [LLVM] for its rich set of capabilities. For this backend we need support for executing arbitrary functions at compile-time. Once we have this in place, we can *boot-strap* the compiler to add more functionality to it. One of the first things to add to this micro-compiler is the ability to load *compiler modules*; these would be shared libraries that can implement various functionalities for the compiler. For example, once boot-strapped,

[1]currently the Sparrow compiler only implements the basic *Return Value Optimization*

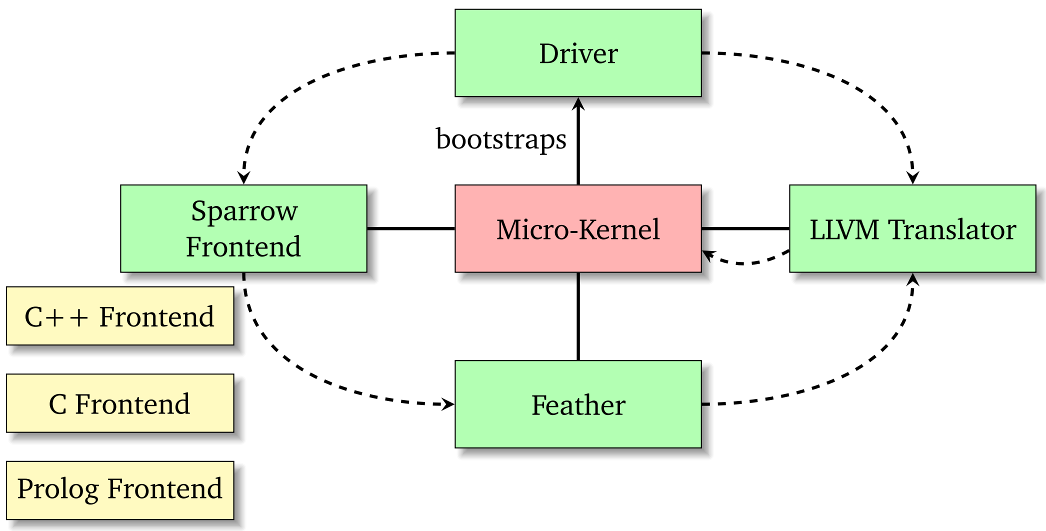


Figure 12.1: The architecture of micro-compiler

the compiler-driver could be one of the loaded modules; then the driver would recognize the fact that the user wants to compile a Sparrow program, and load the corresponding module that would know how to interpret the Sparrow code.

The architecture of such a compiler is presented in fig. 12.1. Our *Nest* component (see section 5.1) would become the Micro-Kernel component. The main difference is that the Micro-Kernel component needs to incorporate the compile-time execution environment (for symmetry reasons, it probably needs to include the run-time execution environment too). This component should have the ability to load external modules and to *plug-in* methods from those modules into the compile-time backend.

The Driver module should be loaded first. This checks how the compiler was invoked, and creates the compiler flows required for the compilation process. It will ask the micro-kernel to load other modules if they are needed. For example, if one needs to compile a Sparrow program, the driver needs to load the Sparrow Frontend, Feather, and LLVM Translator modules. If the input program is a C program, the C Frontend module would be loaded.

So far, the modules that we have presented handle major compiler functionalities (frontend, middleware, and backend). However we can easily imagine modules that just extend the compilation processes. For example, one could create a module to implement enumerations for the language. This allows extending the language in an easier and more flexible way.

Implementing a feature as a compiler module rather than creating compile-time code in the language (as we did in chapter 11) has the advantage of being faster. One can optimize a module before it is loaded by the compiler, whereas a typical metaprogram code is not optimized on the fly by the compile-time execution engine.

We envision a shared repository of modules (e.g., similar to a Linux package repository), where the users of the compiler can choose which modules to use. If creating a module is simple enough, the number of modules in this repository can grow dramatically, making it much easier to add new functionality to compilers.

We expect that implementing micro-compilers would have the same impact on the world of programming languages that micro-kernels had on the world of operating systems [TB14].

12.3 More functional Sparrow

At the time of writing this thesis, almost ten years have passed since the inception of Sparrow. What initially started as *we need to build an object-oriented language*, has changed over the years to *object-oriented is not a must*. As time passed, we empirically found that object-oriented principles are not perfectly aligned with Sparrow principles. The main reason is that object-oriented programming imposes unneeded complexity in the programming language, that, in some contexts, can damage naturalness and flexibility.

Below we attempt to provide some arguments that support our perspective.

Sparrow emphasizes the use of generic programming to produce short, easy-to-understand code that is also efficient. However, we know from C++ experience [Jos12; VJ02; Str13] that generics do not mix well with object-oriented features. Firstly, a C++ template method cannot be virtual, thus one cannot have subtype polymorphism with generics. Secondly, there is a tension between placing operations inside a class (a typical object-oriented approach), and placing them outside a class (a typical generic programming approach). For example, the architecture of the STL [Jos12; MS89] makes the distinction between three important components: containers, iterators, and algorithms. The containers are data structures that implement little functionality; most of the functionality is implemented by the algorithms; the algorithms are decoupled by the containers through the use of iterators. This approach allows the user to write significantly less code than a traditional object-oriented approach[2], and the code is more efficient.

Moreover, including functions into classes is not a direct consequence of object-oriented programming principles. Below there is a brief analysis of these principles, showing how they can be implemented in languages that are not object-oriented:

- **Encapsulation** (packaging data and functions). This is achieved easily by using packages (namespaces in C++, signatures in ML).

[2]if we have M algorithms and N data structures, the amount of code is $O(M + N)$ for generic programming and $O(M \cdot N)$ for a typical object-oriented approach [Sie05; Jos12]

- **Information hiding**. Again, packages provide a better way of handling this than classes.
- **Decoupling**. This is present in the vast majority of languages that support *structured programming*; the best software engineering practices advise using *modules* to decouple between different parts of a system. The best way one can implement these modules is by using packages. In addition to creating these modules, one needs to provide a method for interacting between modules; these are the so-called module interfaces. The interfaces can be implemented as collections of functions and data structures exposed by the module.
- **Polymorphism**. As an alternative to the subtype polymorphism used by object-oriented languages one can use *parametric polymorphism* (e.g., as used by generic programming) or *ad-hoc polymorphism* (e.g., function overloading)
- **Inheritance**. This is not a language feature or design principle as it does not entail any benefits on its own. In an object-oriented programming language like C++ there are voices that argue that inheritance is not that useful and can be modeled by object composition; for example see item 34 (Prefer composition to inheritance) from *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* [SA04] and the *Inheritance Is The Base Class of Evil* presentation [Par14].
- **Dynamic dispatch**. Although object-oriented languages provide an easy method of implementing dynamic dispatch (e.g., virtual functions in C++), other languages have different strategies. Languages like C provide function pointers to handle this, while functional languages provide closures to implement dynamic behavior. Essentially, a closure or a function pointer is an interface with a single method, and any object-oriented interface can be decomposed into smaller, one-function interfaces.

Functional languages can offer the same abstraction level as object-oriented languages. For example, in languages like Haskell and ML, packages are properly implemented to support encapsulation, information hiding, and decoupling. Parametric polymorphism is widely used to model polymorphic behavior. Closures can be seen as one-method interfaces. Functions represent the basic tool for building programs in these languages. Typically the implementation of a function in a functional language is simpler than in a language like C++.

Let us consider creating a method in C++. First, we note that calling this function requires an additional syntax. If `obj` is an object for which we want to call the `doStuff` method, then the user must write `obj.doStuf()`, instead of `doStuff(obj)` as one would write for regular functions. The compiler will create internally a **`this`** parameter and transform the method call into a function call. Inside the method, accessing the current object is done by using a special syntax: the **`this`** keyword; accessing base classes is also done by a special construct—in constructors the name of the class can be used, but inside the method a cast from **`this`** to the actual class needs to be performed. If we have a binary operation for a custom type (e.g., plus operator for a `Complex` class) one cannot write a method that is symmetric with respect to the two arguments (as the plus operation is); the left-hand side parameter is passed as **`this`**, and the right-hand side parameter is passed as a regular parameter. In addition, the semantics of this method do not follow the implicit conversion rules that apply to a plus operation on integers[3].

Moving forward, we need to introduce more syntactic changes to methods compared to functions. First, some methods are *class methods*, as they do not require an active object; in C++ they are marked as **`static`**. To distinguish between dynamic dispatch and static dispatch, the C++ language requires the user to annotate with **`virtual`** the methods for which dynamic dispatch needs to be applied. Abstract methods (the ones that need to be overriden in the derived classes) can be specified with yet another syntax element: appending `= 0` at the end of the method declaration. The 2011 version of the C++ standard adds new keywords for better specifying user's intent: `override` and `final`. For the automatically generated member functions, as of C++ 2011, the user can also use two new keywords: `default` and `delete`.

Because sometimes the objects on which we call methods can be constants, the C++ standard allows the user to specify **`const`** when declaring a method; similarly, for volatile objects, the user can also add the **`volatile`** attribute to the method declaration. With the introduction of move semantics in C++ 2011, the user can also specify the ref-qualification when declaring the function: `&` or `&&`.

As the reader can see, C++ has a large amount of language complexity for methods. Most of the presented special syntaxes are related only to the handling of the **`this`** parameter. Functional languages, on the other hand, do not suffer from the same problems. The functions are always outside classes, and there is no **`this`** parameter. All the function parameters behave the same.

Sparrow could benefit a lot from this language simplicity.

In addition to simplifying the language, functional languages usually employ two techniques that would also be very useful in Sparrow. The first one is data immutability. Even if Sparrow will not aim for absolute purity like Haskell, making data immutable by default can help in increasing the naturalness and in some cases even the efficiency of the language. As an immediate consequence, the user will find that it is much easier to reason on immutable data, knowing that nobody is changing the data while the user is working with it. The compiler can also employ a new set of optimizations if it knows that there are no aliases to the data that can change its content. Still, probably the biggest benefit of data immutability would be in multi-threading. Concurrent algorithms are easier to write and reason about if they only operate

[3]there is no implicit conversion of the **`this`** parameter, but implicit conversion can happen for the right-hand side parameter; this means that in an operation like `a + b` the `a` argument is not treated similarly to the `b` argument

on immutable data; as no mutexes need to be held, there will be no inherent deadlocks for this model.

Another important feature of functional languages that would be welcome in Sparrow is pattern matching. For some problems, pattern matching can greatly simplify the code that needs to be written. A good example of a domain in which this helps is the construction of compilers; for instance, Sestoft [Ses12] provides F# implementations for major programming language concepts which are short and simple.

We are also considering simplified type definitions, discriminated unions, currying, and migrating from ad-hoc generics to parametric polymorphism.

Eventually, this road would lead us to the creation of a programming language that can be called functional, but has the efficiency level of C++.

12.4 Concept-oriented programming

We defined *concepts* as being *predicates on types* (see chapter 7). In other words, they are sets of types. For example, the `Number` Sparrow concept can be seen as the set of types that contains: `Int`, `UInt`, `Long`, `Double`, etc.

This means that for every data type we can envision a concept that generalizes only that type. In this case, for function parameters, we can completely replace data types with concepts. The actual data types become merely implementation details—indicating to the compiler the layout of the data.

We define *concept-oriented programming* as the style of programming that emphasizes the use of concepts for function parameters instead of concrete types. In this style, regular functions are replaced by generic functions. We therefore lift the functions from a concrete implementation to a more abstract, general method of computation.

In a typical program, one would use type erasure or conversions (implicit or explicit) to satisfy the requirements of the called functions. For example, if a function takes as parameter a vector of integers, but we have a vector of short integers, the user needs a manual conversion. If all the functions are generics, these conversions and type erasure are not necessary; the compiler will know more about the *actual* types that the programmer is using. Reducing type erasure and the number of required conversions will only help it to generate more efficient code. For example, partial specialization can be applied to choose the best algorithms for implementing a specific sub-problem.

Concept-oriented programming would allow the programmer to write more general programs, and the compiler to generate more efficient code.

For this technique to work in practice we need to move our compiler towards using ad-hoc generics to a lesser degree. Our generics also need to have separate compilation. Otherwise, compiling simple problems would take a relatively long time.

12.5 Sparrow 2.0

At the moment of writing this section, we already have a rough plan to take the Sparrow programming language to the next level: *Sparrow 2.0*. According to our plans, the language would become more functional (while retaining its imperative features for efficiency), and the internal compiler architecture would migrate towards the micro-compiler scheme.

Because writing programs in Sparrow can be easier than writing them in C++, while maintaining the same efficiency level, our first action would be to rewrite the compiler itself in Sparrow. We want to have a more complex program written in Sparrow. This would be a sign of language maturity. Having implemented the language in Sparrow, we can start to simplify the language by moving towards a more functional style. As a side effect, the Sparrow compiler can also be simplified. Afterwards, the current compiler architecture can be simplified even further by completely switching to a micro-compiler approach.

At the end of this process we hope to achieve a simple language for which a compiler can easily be implemented. This language will be more natural, more flexible, while also maintaining the same high level of efficiency.

We shall not cease from exploration, and the end of all our exploring will be to arrive where we started and know the place for the first time.

T. S. Eliot

Chapter 13 Conclusions

This thesis presented the design and implementation of the Sparrow programming language. We started from the design principles of the language, presented hyper-metaprogramming as the central feature of the language, provided a sketch of the translation processes that are present in the Sparrow compiler, and finally we walked the reader through a series of case-studies to analyze and evaluate Sparrow. It remains for this chapter to analyze whether Sparrow has reached its goal of being a language that integrates efficiency, flexibility, and naturalness.

After a brief introduction to the Sparrow programming language (chapter 3) we presented in chapter 4 the design of Sparrow. The design focuses on the three main principles of the language, and how they can be integrated into the same language. We presented hyper-metaprogramming as the language feature that may provide the *glue* between these principles. We then presented the reduction mechanism, which gives Sparrow the ability to translate complex structures into simpler ones. Hyper-metaprogramming and the reduction mechanism provide the required support for implementing all the other Sparrow features, in accordance with the stated principles.

Chapter 5 provides a sketch of translation from a Sparrow program, to the corresponding LLVM code. This chapter actually describes how our reference compiler was implemented. There are three major implications resulting from this chapter. Firstly, we have a semi-formal method of defining the Sparrow programming language. Secondly, we provide a set of instructions on how to build a compiler for the language, showing that Sparrow is not just a theoretic language. The last implication that can be extracted from this chapter is the efficiency of Sparrow; looking carefully at the translation rules that we have stated, one can notice that there is no place in the compiler in which unneeded code is generated. This means that the user of the language will not pay for what he does not use.

The second part of the thesis presented six case studies on Sparrow, with the aim of analyzing the language with respect to all three principles. Instead of focusing on a single principle per case study, we preferred to analyze each study from as many points of view as possible. For each feature that we presented we tried to assess naturalness, flexibility, and efficiency.

Chapter 6 presented the operators feature of Sparrow. We have shown that this feature is flexible enough to allow the programmer to create his own operators, and customize their precedence and associativity. All of these are performed at library level; the configuration of operators is done through compile-time function calls. We have shown that this scheme is easy to use and more flexible than in other languages. Moreover, because operator calls are just function calls, there is no efficiency penalty of using this scheme.

In chapter 7 we presented the generic programming feature from Sparrow. We have a simple syntax for defining and using generics, and yet it is powerful enough to model structures that are present in other similar languages with support for generic programming (e.g., C++, D, $\mathcal{G}$). In addition to allowing parameterized declarations, Sparrow also supports **`if`** clauses as a method of imposing instantiation conditions, and, as a generalization, *concepts*. Concepts are predicates on types, but they can also be viewed as representing sets of types. With concepts, one can parameterize algorithms on sets of types, thus greatly increasing the expressiveness of generic programming in Sparrow. Unlike the proposed feature for C++ [SS12a], Sparrow concepts are easy to define and understand, yet more powerful. This chapter has shown that simple rules can generate features that are easy to use, yet flexible enough to be useful in many contexts.

The efficiency of the generics feature was measured in chapter 8, which also provided a general analysis of efficiency in Sparrow. We presented a series of tests in increasing order of complexity. We have shown that Sparrow is at least as efficient as C++. Besides presenting the efficiency measurements, this chapter has also shown that writing algorithms in Sparrow is easy. Using ranges one can express problems in a reduced code size (sometimes even shorter than the problem description in English), and yet the compiler will generate efficient code. The last case study in this chapter compared Sparrow to C++, Go, Java, and Scala[1]. Sparrow ranked highest in terms of execution time and program size; the only aspect in which Sparrow fell somewhat short was the compilation time. This chapter has clearly shown that Sparrow can be simultaneously natural, flexible, and efficient.

Moving forward on the efficiency aspect, chapter 9 presented a novel technique of moving computations from run-time to compile-time. This technique

[1]the Havlak loop recognition algorithm was implemented in a similar manner in all the languages

uses hyper-metaprogramming in conjunction with partial evaluation [JGS93; BJE88] to create a specialized version of the final program that will run faster. We have shown that, if one has programs that can be specialized on compile-time arguments, one can obtain significant speedups by moving some of the computations to compile-time. After developing a theoretical framework for computing this speedup, we measured it in two case-studies: one involving minimal perfect hashing, and another involving regular expressions. As a result, we obtained speedups of approximately 80 for minimal perfect hashing, and over 12 for regular expressions. These speedups were obtained without sacrificing convenience: the user would still use the same interface for interacting with the minimal perfect hashing and regular expression components. This is another example in which the language is flexible enough that it allows speeding up programs while maintaining the naturalness level.

Chapter 10 analyzes the implications of using hyper-metaprogramming inside programs. Considering how template metaprogramming works in C++ [Dub13], one would expect that executing computations at compile-time would be slow, cumbersome, and impractical. Our chapter shows that by using hyper-metaprogramming, metaprograms are written as easily as regular programs, and they are executed in a similar manner. Our studies showed that one can run complex algorithms that allocate memory on the heap, read data from files, interact with the operating system, and print information to the console output while compiling. Moreover, the execution speed for these algorithms is much better than that of C++ metaprograms. Actually, executing computations at compile-time in Sparrow is comparable to executing them at run-time in an interpreted environment. We have shown that generating code can be more time consuming than executing complex algorithms at compile-time.

As our final case study, chapter 11 has shown how one can embed a programming language inside Sparrow. We have used as an example a subset of the Prolog language, because it represents a totally different programming paradigm (logic programming), and all the important concepts are not directly mappable to Sparrow concepts. As a result, the user can write Prolog programs directly in Sparrow, using Prolog syntax; moreover, one can *import* Prolog files, just as one would import other Sparrow source files. We have shown that adding an embedded language to Sparrow, and also a new programming paradigm, does not require any changes to the compiler: it can be implemented solely as a library feature. Adding a new language makes heavy use of the hyper-metaprogramming functionality; one can implement a parser, a semantic analyzer, and a code generator as metaprograms to be executed at compile-time in Sparrow. The code generator can use the *Compiler API*, which internally will use exactly the same mechanisms that the Sparrow compiler uses for code generation. Using hyper-metaprogramming, we can *open up the compiler*, adding the possibility to extend it, and thus extending the language—a sign of increased flexibility. Having the possibility to add new domain-specific languages and programming paradigms in Sparrow can only increase the naturalness of the language, as the programmer can choose the most appropriate language/paradigm to solve a particular sub-problem. Moreover, on the efficiency side, we have shown that in certain conditions, one can take advantage of Sparrow's compiler and optimizer to generate efficient code out of a logic specification—we essentially transformed logic programs into imperative programs.

Although the Sparrow programming language fulfills to a high degree its initial goals, we can improve it even further. Chapter 12 presented some of the ideas for building a language that is even more powerful.

Overall we provided evidence to support the claim that we have achieved our initial goals:

1. we have successfully built the Sparrow programming language, and a reference compiler, to satisfy our design principles
2. Sparrow is efficient
3. Sparrow is flexible
4. Sparrow is natural (easy to use)
5. hyper-metaprogramming can be implemented in a programming language, and can be useful
6. hyper-metaprogramming is feasible in practice; there are no major limitations, as in the case of the C++ template system
7. one can move computations to compile-time to speed up programs
8. one can design languages with high degrees of efficiency, flexibility, and naturalness; hyper-metaprogramming can be used to *glue* these qualities together

By creating Sparrow, we have created an efficient, flexible, and natural programming language, that fulfills all the goals we have set for our research.

13.1 Main contributions

Theoretical:

- A new method of constructing programming languages that can integrate flexibility, efficiency and naturalness.
- A theoretical framework for constructing hyper-metaprogramming, an innovative form of compile-time metaprogramming that is easy to use, flexible and fast.
- By making hyper-metaprogramming a central feature of the Sparrow language, we create a novel technique of constructing programming languages. With this, we also change the status of compile-time metaprogramming from an *esoteric* feature to a potentially fundamental one.
- An original method of improving execution times by moving parts of the computations to compile-time. We described both a theoretical framework and some applications.

Applicative:

- A new programming language, Sparrow, that integrates flexibility, efficiency and naturalness. Sparrow is at least as efficient as C++, more flexible and more natural than most of the mature programming languages (e.g., C++, Scala, Java).
- Advances in the way a programming language can define operators. We proposed a method of constructing an operators system that is more flexible than those found in existing programming languages, while still being natural and fast.
- Contributions in the field of generic programming. We devised a method of constructing a generic programming model in programming languages that is simple, easy to use, efficient and powerful.
- Proposed an innovative method of constructing *concepts* that is both simple and powerful.
- Contributions to the way one can represent efficient computations in a natural manner. Our method is based on generic programming and a library feature called *ranges*, and lets the user implement problems in simple expressions, using the terms found in the problem description, while maintaining the same efficiency as C++.
- An original method of improving execution times by moving parts of the computations to compile-time. We described both a theoretical framework and some applications. The resulting speedups are impressive: more than 12x for regular expressions and 80x for minimal perfect hashing.
- A comprehensive analysis of the costs implied by hyper-metaprogramming. Our results show that executing algorithms at compile-time is easy and fast; almost as fast as executing them at run-time.
- A novel technique of embedding domain-specific programming languages into a host programming language, using hyper-metaprogramming. This technique allows the implementation to be done purely at library level, without changing the compiler of the host language; at the same time the user will use the syntax of the embedded language, and the integration is seamless (e.g., homogeneous error reporting, anti-quotation for interacting between languages). The generated code is also efficient.
- Allowing the user to extend a language with new domain-specific languages and with new programming paradigms also constitutes a major accomplishment in increasing the naturalness of programming languages; the user is able to express computations in the paradigm or language that best fits the problem.

Conferences and Publications

- Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “General Efficiency Analysis for the Sparrow Programming Language”. In: *20th International Conference on Control Systems and Computer Science (CSCS)*. 2015. Best paper award.
  - parts of this paper can be found in chapter 8
- Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “Flexible Operators In Sparrow”. In: *Proceedings of IAC-EIaT 2014*. 2014.
  - parts of this paper can be found in chapter 6
- Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “Moving Computations from Run-time to Compile-time: Hyper-metaprogramming in Practice”. In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. 2014.
  - parts of this paper can be found in chapter 9
- Lucian Radu Teodorescu and Rodica Potolea. “Compiler Design for Hyper- metaprogramming”. In: *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2013.
  - parts of this paper can be found in chapter 4
- Lucian Radu Teodorescu and Rodica Potolea. “Metaprogramming can be Easy, Fast and Fun: A Plea for Hyper-Metaprogramming”. In: *ACAM Journal of Automation Computers and Applied Mathematics* 21 (2012).
  - parts of this paper can be found in chapter 10
- Lucian Radu Teodorescu, Alin Suciu, and Rodica Potolea. “Sparrow: Towards a New Multi-Paradigm Language”. In: *Analele Universitatii de Vest din Timisoara* 48.3 (2011).
  - parts of this paper can be found in chapter 4
- Lucian Radu Teodorescu and Rodica Potolea. “Sparrow: a language for flexibility, efficiency and naturalness”. In: *ACM SIGPLAN conference on Programming language design and implementation—poster session*. 2009.

Awaiting publication

- Lucian Radu Teodorescu and Rodica Potolea. “Adding Logic to Imperative Programming: A Sparrow experience”.
- Lucian Radu Teodorescu and Rodica Potolea. “Generic programming in

Sparrow".

Scientific articles not related to thesis domain

- Lucian Radu Teodorescu, Răzvan Boldizsar, Mihai Ordean, Melania Duma, Laura Deteșan and Mihaela Ordean. "Part of Speech Tagging for Romanian Text-to-Speech System". In: *Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*. 2011.
- Andrei Șaupe, Lucian Radu Teodorescu, Mihai Alexandru Ordean, Răzvan Boldizsar, Mihaela Ordean and Gheorghe Cosmin Silaghi. "Efficient parsing of romanian language for text-to-speech purposes". In: *Proceedings of the 12th International Conference on Text, Speech and Dialogue*. 2009.

Listings

List of Figures

List of Tables

References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AG04] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[AJR+01] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. *STAPL: An Adaptive, Generic Parallel C++ Library*. 2001.

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing, 2001.

[Ale09] Andrei Alexandrescu. *Iterators Must Go*. 2009.

[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 2010.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[AOP+98] N. I. Adams, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand, H. Abelson, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, and E. Kohlbecker. *Revised5 report on the algorithmic language scheme*. Tech. rep. 9. 1998.

[App97] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, 1997.

[Aus99] Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing, 1999.

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. "Hash, Displace, and Compress". In: *Algorithms-ESA*. Vol. 5757. 2009.

[BDQ98] Federico Bassetti, Kei Davis, and Dan Quinlan. "C++ expression templates performance issues in scientific computing". In: *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*. 1998.

[BG] Fulgham Brent and Isaac Gouy. *The Computer Language Benchmarks Game*. URL: `http://benchmarksgame.alioth.debian.org` (visited on 03/01/2015).

[BG96] Thomas J. Bergin Jr. and Richard G. Gibson Jr. *History of programming languages II*. ACM, 1996.

[Bis] *GNU Bison*. URL: `https://www.gnu.org/software/bison/` (visited on 03/01/2015).

[BJ95] Frederick P. Brooks Jr. *The mythical man-month (anniversary ed.)* Addison-Wesley Longman Publishing, 1995.

[BJE88] D. Bjorner, Neil D. Jones, and A.P. Ershov, eds. *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark, 18-24 Oct., 1987*. Elsevier Science Inc., 1988.

[Bru02] Kim B. Bruce. *Foundations of object-oriented languages: types and semantics*. MIT Press, 2002.

[BW09] Federico Biancuzzi and Shane Warden. *Masterminds of Programming*. O'Reilly Media, 2009.

[BZ08] Fabiano Cupertino Botelho and Nivio Ziviani. "Near-optimal space perfect hashing algorithms". In: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management - CIKM '07*. 2008.

[C11] *ISO/IEC 9899:2011: Programming languages – C*. Tech. rep. 2011.

[Car96] Luca Cardelli. "Type Systems". In: *ACM Comput. Surv.* 28.1 (1996).

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CFDH92] Qi Fan Chen, Edward A. Fox, Amjad M. Daoud, and Lenwood S. Heath. "Practical minimal perfect hash functions for large databases". In: *Communications of the ACM* 35 (1992).

[CH11] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 46.4 (2011).

[CLOC] Al Danial. *CLOC: Count lines of code*. URL: `http://cloc.sourceforge.net` (visited on 03/01/2015).

[CMPH] Davi Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. *C Minimal Perfect Hashing Library*. URL: `http://cmph.sourceforge.net` (visited on 03/01/2015).

[Cos00] Eugeniu Coseriu. *Lessons of General Linguistics [Lecții de lingvistică generală]*. ARC Publishing House, 2000.

[COST04] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. "DSL Implementation in MetaOCaml, Template Haskell, and C++". In: *Domain-Specific Program Generation* 3016 (2004).

[Cox07] Russ Cox. *Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...)* 2007. URL: `http://swtch.com/~rsc/regexp/regexp1.html`.

[Cpp03] *ISO/IEC 14882:2003: Programming languages – C++*. Tech. rep. 2003.

[Cpp11] *ISO/IEC 14882:2011: Programming languages – C++*. Tech. rep. 2011.

[Cpp14] *ISO/IEC 14882:2014: Programming languages – C++*. Tech. rep. 2014.

[Cpp98] *ISO/IEC 14882:1998: Programming languages – C++*. Tech. rep. 1998.

[CT11] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2011.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. "Implementing multi-stage languages using ASTs, Gensym, and reflection". In: *Proceedings of the 2nd international conference on Generative programming and component engineering*. 2003.

[CW85] Luca Cardelli and Peter Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys* 17.4 (1985).

[Dla] *The D programming language*. URL: `http://dlang.org`.

[DM88] Saumya K. Debray and Prateek Mishra. "Denotational and operational semantics for prolog". In: *The Journal of Logic Programming* 5.1 (1988).

[DS06] Gabriel Dos Reis and Bjarne Stroustrup. "Specifying C++ concepts". In: *33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ISO/IEC. 2006.

[DS10] Gabriel Dos Reis and Bjarne Stroustrup. "General constant expressions for system programming languages". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. 2010.

[DS93] Charles Donnelly and Richard Stallman. *Bison: the Yacc-compatible parser generator*. Free Software Foundation, 1993.

[Dub13] Denis V. Dubrov. "N Queens Problem: a Metaprogramming Stress Test for the Compiler". In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*. 2013.

[FGK+98] Andreas Fabri, Geert-jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schonherr. "On the design of CGAL a computational geometry algorithms library". In: *Software-Practice and Experience* 30 (1998).

[Fin96] Raphael A. Finkel. *Advanced programming language design*. 1996.

[Fla12] Matthew Flatt. "Creating languages in Racket". In: *Communications of the ACM* 55 (2012).

[Fle] *flex: The Fast Lexical Analyzer*. URL: `http://flex.sourceforge.net` (visited on 03/01/2015).

[Fut99] Yoshihiko Futamura. "Partial Evaluation Of Computation Process - An Approach To a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12.4 (1999).

[GA] Aleksey Gurtovoy and David Abrahams. *Boost MPL Library*. URL: `http://www.boost.org/libs/mpl/doc/index.html` (visited on 03/01/2015).

[Gcc] *GCC, the GNU Compiler Collection*. URL: `https://gcc.gnu.org` (visited on 03/01/2015).

[Gib03] Jeremy Gibbons. “Patterns in Datatype-Generic Programming”. In: *Multiparadigm Programming*. Vol. 27. 2003.

[GIL] *Boost Generic Image Library*. URL: `http://www.boost.org/doc/libs/release/libs/gil/` (visited on 03/01/2015).

[GJK+05] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. “Generic Programming and High-Performance Libraries”. In: *International Journal of Parallel Programming* 33.2 (2005).

[GJL+03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. “A comparative study of language support for generic programming”. In: *SIGPLAN Notices* 38.11 (2003).

[GJL+07] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. “An Extended Comparative Study of Language Support for Generic Programming”. In: *Journal of Functional Programming* 17.2 (2007).

[GJLB00] Dick Grune, C. Jacobs, Koen Langendoen, and Henri Bal. *Modern Compiler Design*. John Wiley & Sons, 2000.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. 2014.

[Glu09] Robert Gluck. “Is there a fourth Futamura projection?” In: *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. 2009.

[Gol06] Lois Goldthwaite. *Technical report on C++ performance*. Tech. rep. 2006.

[Got06] Peter Gottschling. *Fundamental Algebraic Concepts in Concept-Enabled C++*. Tech. rep. TR638. Indiana University, 2006.

[GP09] Jeremy Gibbons and Ross Paterson. “Parametric Datatype-Genericity”. In: *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming - WGP ’09*. 2009.

[GSJR06] Douglas Gregor, Bjarne Stroustrup, Jaakko Järvi, and Gabriel Dos Reis. “Concepts: Linguistic support for generic programming in C++”. In: *SIGPLAN Notices*. Indiana University, Texas A&M University, Rice University. 2006.

[GWd07] Jeremy Gibbons, Meng Wang, and Bruno César dos Santos Oliveira. “Generic and Indexed Programming”. In: *Trends in Functional Programming*. 2007.

[GWJ08] Peter Gottschling, David S. Wise, and Adwait Joshi. “Generic Support of Algorithmic and Structural Recursion for Scientific Computing”. In: *The International Journal of Parallel, Emergent and Distributed Systems* (2008).

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. “Type classes in Haskell”. In: *ACM Transactions on Programming Languages and Systems* 18.2 (1996).

[HL07] Ralf Hinze and A. Löh. "Generic programming, now!" In: *Datatype-Generic Programming* (2007).

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

[HPP09] Mary Hall, David Padua, and Keshav Pingali. "Compiler research: the next 50 years". In: *Commun. ACM* 52.2 (2009).

[Hun11] Robert Hundt. "Loop recognition in C++/Java/Go/Scala". In: *Proceedings of Scala Days*. 2011.

[JGS93] Neil D. Jones, Carsten Krogh Gomard, and Peter Sestof. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[JGW+06] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. "Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++". In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. 2006.

[JLPR08] Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. "Libraries for generic programming in Haskell". In: *Proceedings of the 6th international conference on Advanced functional programming*. 2008.

[JMS07] Jaakko Järvi, Matthew A. Marcus, and Jacob N. Smith. "Library composition and adaptation using C++ concepts". In: *GPCE'07: Generative Programming and Component Engineering*. 2007.

[Joh75] Stephen C. Johnson. *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975.

[Jon92] Mark P. Jones. "Qualified types: theory and practice". PhD thesis. Oxford University Computing Laboratory, 1992.

[Jon96] Neil D. Jones. "An introduction to partial evaluation". In: *ACM Computing Surveys (CSUR)* 28.3 (1996).

[Jos12] Nicolai M. Josuttis. *The C++ standard library: a tutorial and reference (second edition)*. Addison-Wesley, 2012.

[JS02] Simon Peyton Jones and Tim Sheard. "Template meta-programming for Haskell". In: *SIGPLAN Not.* 37.12 (2002).

[JWL03] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. "Concept-Controlled Polymorphism". In: *GPCE'03: Generative Programming and Component Engineering*. Indiana University. 2003.

[KMA88] Aaron Kershenbaum, David Musser, and Stepanov Alexander. *Higher order imperative programming*. Tech. rep. 1988.

[Koe12] Andrew Koenig. *A Personal Note About Argument-Dependent Lookup*. 2012. URL: `http://www.drdobbs.com/cpp/a-personal-note-about-argument-dependent/232901443`.

[KRE88] B.W. Kernighan, D.M. Ritchie, and P. Ejeklint. *The C programming language*. 1988.

[Lat02] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". MA thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[Lat11] Chris Lattner. "LLVM and Clang: Advancing compiler technology". In: *Free and Open Source Developers' European Meeting, Brussels, Belgium*. 2011.

[Lev09] John Levine. *Flex & Bison*. O'Reilly Media, 2009.

[LL02] Lie-Quan Lee and Andrew Lumsdaine. "Generic programming for high performance scientific applications". In: *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. 2002.

[LLVM] *The LLVM Compiler Infrastructure*. URL: `http://www.llvm.org` (visited on 03/01/2015).

[LM97] Andrew Lumsdaine and Brian C Mccandless. "Parallel Extensions to the Matrix Template Library". In: *Parallel Processing for Scientific Computing*. 1997.

[LP03] Ralf Lämmel and Simon Peyton Jones. "Scrap your boilerplate: a practical approach to generic programming". In: *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation*. Vol. 38. 3. 2003.

[LPJ05] Ralf Lämmel and Simon Peyton-Jones. "Scrap your boilerplate with class: extensible generic functions". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)* (2005).

[LS75] M.E. Lesk and E. Schmidt. *Lex – A lexical analyzer generator*. Tech. rep. 1975.

[LV04] Chris Lattner and Adve Vikram. *The LLVM Compiler Framework and Infrastructure*. 2004.

[MaDJL10] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. "A generic deriving mechanism for Haskell". In: *Proceedings of the third ACM Haskell symposium on Haskell - Haskell '10*. Vol. 45. 11. 2010.

[Mai07] Geoffrey Mainland. "Why it's nice to be quoted". In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop - Haskell '07*. 2007.

[Mak09] Ronald Mak. *Writing Compilers and Interpreters: A Software Engineering Approach*. Wiley Publishing, 2009.

[Mar10] Simon Marlow. *Haskell 2010 language report*. Tech. rep. 2010.

[Mey01] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Longman Ltd., 2001.

[Mey14a] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 2014.

[Mey14b] Scott Meyers. *Type Deduction and Why You Care (presentation)*. 2014. URL: `http://channel9.msdn.com/events/CPP/C-PP-Con-2014/Type-Deduction-and-Why-You-Care`.

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 375.3 (1978).

[MS89] David R. Musser and Alexander A. Stepanov. “Generic Programming”. In: *ISAAC ’88: Proceedings of the International Symposium ISSAC’88 on Symbolic and Algebraic Computation*. 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT press, 1997.

[Nai10] Roshan Naik. *Castor 1.1 Reference Manual*. Tech. rep. 2010.

[Ode11] Martin Odersky. *The Scala Language Specification v2.9*. Tech. rep. 2011.

[Ode14] Martin Odersky. *Scala By Example*. Tech. rep. 2014.

[Par] Terence Parr. *ANTLR*. URL: `http://www.antlr.org` (visited on 03/01/2015).

[Par10] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2010.

[Par14] Sean Parent. *Inheritance Is the Base Class of Evil (presentation)*. 2014. URL: `http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil`.

[PF11] Terence Parr and Kathleen Fisher. “LL(*): the foundation of the ANTLR parser generator”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011.

[Pho] *Phoenix Compiler and Shared Source Common Language Infrastructure*. URL: `http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx` (visited on 03/01/2015).

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PLMS00] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, 2000.

[Por10] Zoltán Porkoláb. “Functional Programming with C++ Template Metaprograms”. In: *Proceedings of the Third summer school conference on Central European functional programming school*. 2010.

[Rac] *Racket: A programmable programming language*. URL: `http://racket-lang.org` (visited on 03/01/2015).

[Rey09] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 2009.

[RH07] Barbara Ryder and Brent Hailpern. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007.

[RJ05] Gabriel Dos Reis and Jaakko Jarvi. “What is Generic Programming?” In: *Proceedings of the First International Workshop on Library-Centric Software Design*. 2005.

[RO12] Tiark Rompf and Martin Odersky. “Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs”. In: *Communications of the ACM* (2012).

[RSA+13] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, Hyoukjoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. "Optimizing Data Structures in High-Level Programs". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*. 2013.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004.

[SC11] Amin Shali and William R. Cook. "Hybrid partial evaluation". In: *ACM SIGPLAN Notices* 46 (2011).

[Sch00] Douglas C. Schmidt. "GPERF: A Perfect Hash Function Generator". In: *More C++ gems*. 2000.

[Sco09] Michael L. Scott. *Programming Language Pragmatics (3rd edition)*. Morgan Kaufmann Publishers, 2009.

[Seb12] Robert W. Sebesta. *Concepts of Programming Languages (10th edition)*. 2012.

[Ses12] Peter Sestoft. *Programming language concepts*. Vol. 50. Springer Science & Business Media, 2012.

[SGB13] A.K. Sujeeth, Austin Gibbons, and K.J. Brown. "Forge: generating a high performance DSL implementation from a declarative specification". In: *Proceedings of the 12th international conference on Generative programming: concepts & experiences* (2013).

[She01] Tim Sheard. "Accomplishments and Research Challenges in Meta-Programming". In: *Proceedings of the 2nd international conference on Semantics, applications, and implementation of program generation*. 2001.

[SHL+12] Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith, and ... *The F# 3.0 Language Specification*. Tech. rep. 2012.

[Sie05] Jeremy G. Siek. "A Language for Generic Programming". PhD thesis. Indiana University, 2005.

[Sie12] Jeremy G. Siek. "The c++0x "concepts" effort". In: *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*. 2012.

[SL00] Jeremy Siek and Andrew Lumsdaine. "Concept checking: Binding parametric polymorphism in C++". In: *In First Workshop on C++ Template Programming*. 2000.

[SL05] Jeremy Siek and Andrew Lumsdaine. "Essential language support for generic programming". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005.

[SL94] Alexander Stepanov and Meng Lee. *The standard template library*. Tech. rep. WG21/N0482, ISO Programming Language C++ Project, 1994.

[SL98] Jeremy G. Siek and Andrew Lumsdaine. “The matrix template library: A generic programming approach to high performance numerical linear algebra”. In: *In International Symposium on Computing in Object-Oriented Parallel Environments*. 1998.

[SLL01] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Pearson Education, 2001.

[SM09] Alexander A. Stepanov and Paul McJones. *Elements of programming*. Addison-Wesley Professional, 2009.

[Sof15] TIOBE Software. *TIOBE Index for February 2015*. 2015. URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (visited on 03/01/2015).

[SR03] Bjarne Stroustrup and Gabriel Dos Reis. *Concepts - Design choices for template argument checking*. Tech. rep. N1522=03-0105. ISO/IEC, 2003.

[SR14] Alexander A. Stepanov and Daniel E. Rose. *From Mathematics to Generic Programming*. Pearson Education, 2014.

[SS12a] Bjarne Stroustrup and Andrew Sutton. *A Concept Design for the STL*. Tech. rep. 2012.

[SS12b] Andrew Sutton and Bjarne Stroustrup. “Design of concept libraries for c++”. In: *Proceedings of the 4th international conference on Software Language Engineering*. 2012.

[SSR13] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. *Concepts Lite*. Tech. rep. 2013.

[ST97] Tim Sheard and Walid Taha. “Multi-stage programming with explicit annotations”. In: *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. Vol. 32. 1997.

[Ste99] Guy Steele. “Growing a language”. In: *Higher-Order and Symbolic Computation* 12.3 (1999).

[Str07] Bjarne Stroustrup. “Evolving a language in and for the real world: C++ 1991-2006”. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007).

[Str13] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[Str67] Christopher Strachey. “Fundamental concepts in programming languages”. In: *Lecture notes for the International Summer School in Computer Programming* (1967).

[Str94] Bjarne Stroustrup. “The design and evolution of C++”. In: (1994).

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, 2014.

[TDP14a] Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “Flexible Operators In Sparrow”. In: *Proceedings of IAC-EIaT 2014*. 2014.

[TDP14b] Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “Moving Computations from Run-time to Compile-time: Hyper-metaprogramming in Practice”. In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. 2014.

[TJ07] Xiaolong Tang and Jaakko Järvi. “Concept-Based Optimization”. In: *Proceedings of the 2007 Symposium on Library-Centric Software Design*. 2007.

[TJ10] Xiaolong Tang and Jaakko Järvi. “Generic flow-sensitive optimizing transformations in C++ with concepts”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. 2010.

[TP12] Lucian Radu Teodorescu and Rodica Potolea. “Metaprogramming can be Easy, Fast and Fun: A Plea for Hyper-Metaprogramming”. In: *ACAM Journal of Automation Computers and Applied Mathematics* 21 (2012).

[TP13] Lucian Radu Teodorescu and Rodica Potolea. “Compiler Design for Hyper-metaprogramming”. In: *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2013.

[Tra08] Laurence Tratt. “Domain Specific Language Implementation via Compile-Time Meta-Programming”. In: *ACM Transactions on Programming Languages and Systems* V.6 (2008).

[TSP11] Lucian Radu Teodorescu, Alin Suciu, and Rodica Potolea. “Sparrow: Towards a New Multi-Paradigm Language”. In: *Analele Universitatii de Vest din Timisoara* 48.3 (2011).

[Unr94] Erwin Unruh. *Prime number computation*. Distributed in the ANSI X3J16-94-0075/ISO WG21426 meeting. 1994.

[Vel03] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. 2003.

[Vel95a] Todd L. Veldhuizen. “Expression Templates”. In: *C++ Report* 7 (1995).

[Vel95b] Todd L. Veldhuizen. “Using C++ template metaprograms”. In: *C++ Report* 7.4 (1995).

[Vel98] Todd L. Veldhuizen. “Arrays in Blitz++”. In: *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE’98)*. 1998.

[Vel99] Todd L. Veldhuizen. “C++ templates as partial evaluation”. In: *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 1999.

[VG98] Todd L. Veldhuizen and Dennis Gannon. “Active libraries: Rethinking the roles of compilers and libraries”. In: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO’98)*. 1998.

[VJ02] David Vandevoorde and Nicolai M Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing, 2002.

[VT11] Naveneetha Vasudevan and Laurence Tratt. “Comparative study of DSL tools”. In: *Electronic Notes in Theoretical Computer Science* 264 (2011).

[WB89] Philip Wadler and Stephen Blott. "How to make ad-hoc polymorphism less ad hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989.

[Wex78] Richard L. Wexelblat. *History of programming languages*. ACM, 1978.

[Wir05] Niklaus Wirth. *Compiler Construction*. November. Addison-Wesley Reading, 2005.

[Wir77] Niklaus Wirth. "What can we do about the unnecessary diversity of notation for syntactic definitions?" In: *Communications of the ACM* 20.11 (1977).

[WRI+10] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. "Mint: Java Multi-stage Programming Using Weak Separability". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2010.

APPENDICES

Appendix A

The grammar of Sparrow

The lexical syntax of Sparrow is given by the following EBNF grammar:

```
Letter   ::= 'a'-'z' | 'A'-'Z' | '_'
Digit    ::= '0'-'9'
OpChar   ::= '~' | '!' | '@' | '#' | '$' | '%' | '^' | '&' | '-' | '+'
           | '=' | '|' | '\' | ':' | '<' | '>' | '?' | '/' | '*'
Oper          ::= OpChar {OpChar}[1]
                | (OpChar | '.') '.' {OpChar | '.'}
Id            ::= Letter {Letter | Digit} ['_' Oper]

Number        ::= (DecimalNum | BinNum | OctalNum | HexNum) ['l'|'L']
DecimalNum    ::= '1'-'9' {'0'-'9' | '_'} ['u'|'U']
                | '0'
BinNum        ::= '0' ('b'|'B') ('0'|'1') {'0'|'1'|'_'}
OctalNum      ::= '0' '0'-'7' {'0'-'7' | '_'}
HexNum        ::= '0' ('x'|'X') HexDigit {HexDigit | '_'}
HexDigit      ::= '0'-'9' | 'a'-'f' | 'A'-'F'
FloatingNum   ::= {Digit} '.' Digit {Digit} [ExpPart] [FloatSuffix]
                | Digit {Digit} ExpPart [FloatSuffix]
                | Digit {Digit} FloatSuffix
ExpPart       ::= ('e'|'E') ['+'|'-'] Digit {Digit}
FloatSuffix   ::= ('f'|'F'|'d'|'D')

Character     ::= "'" [Escape | AnyChar] "'"
String        ::= '"' [Escape | AnyChar]* '"'
                | '<' '{' AnyChar* '}' '>'
Escape        ::= '\' ['r'|'n'|'b'|'f'|'t'|'\' | "'" | '"']
                | '\' '0'-'7'
                | '\' '0'-'7' '0'-'7'
```

[1]in our implementation, the `Oper` lexical rule will not match =; this is needed to distinguish between expressions that contains the equal sign, and the ones that do not

```
                 | '\' '0'-'3' '0'-'7' '0'-'7'
                 | '\' ['x'|'X'] HexDigit HexDigit?

  Symbols  ::= '{' | '}' | '[' | ']' | '(' | ')' | ':' | ';'
             | ',' | '.' | '`' | '='
  Keywords ::= "break" | "catch" | "class" | "concept" | "continue"
             | "fun" | "if" | "else" | "false" | "finally" | "for"
             | "friend" | "import" | "null" | "package" | "private"
             | "protected" | "public" | "return" | "this" | "throw"
             | "true" | "try" | "using" | "var" | "while"

  LineComment   ::= '/' '/' AnyChar* '\n'
  MultiComment  ::= '/' '*' AnyChar* '*' '/'
  Whitespace    ::= ' ' | '\t' | '\r'
  Newline       ::= '\n'
```

The context-free syntax of Sparrow is given by the following EBNF grammar:

```
  ProgramFile      ::= [PackageTopDecl] {Import} {TopLevel}

  PackageTopDecl   ::= 'package' QualifiedId ';'
  Import           ::= 'import' QualifiedIdStar ';'
                     | 'import' String ';'
  QualifiedId      ::= Id {'.' Id}
  QualifiedIdStar  ::= QualifiedId {'.' '*'}

  TopLevel    ::= Declaration
                | IfStmt
                | Expr ';'
  Declaration ::= [AccessSpec] UsingDecl
                | [AccessSpec] PackageDecl
                | [AccessSpec] ClassDecl
                | [AccessSpec] ConceptDecl
                | [AccessSpec] VarDecl
                | [AccessSpec] FunDecl
  AccessSpec ::= 'private' | 'protected' | 'public'
  Mods ::= '[' Expr {',' Expr} ']'
  UsingDecl  ::= 'using' [Mods] QualifiedIdStar ';'
               | 'using' [Mods] Id '=' Expr ';'
  PackageDecl ::= 'package' [Mods] Id '{' {TopLevel} '}'
  ClassDecl ::= 'class' [Mods] Id [Params] [':' ExprList] IfClause
                '{' {TopLevel} '}'
  ConceptDecl ::= 'concept' [Mods] Id '(' Id [Type] ')' IfClause ';'
  VarDecl    ::= 'var' [Mods] IdList [Type] ['=' Expr] ';'
             | 'var' [Mods] IdList '=' Expr ';'
  FunDecl    ::= 'fun' [Mods] FunName [Params] [Type] IfClause FunBody
             | 'fun' [Mods] FunName [Params] [Type] '=' Expr IfClause';
               '
  IdList     ::= Id {',' Id}
  Type       ::= ':' ExprNE
  FunName    ::= IdOrOperator | '(' ')'
```

```
FunBody   ::= '{' {Stmt} '}' | ';'
Params    ::= '(' [Formal {',' Formal}] ')'
            | IdList
Formal    ::= IdList [Type] ['=' Expr]
IfClause ::= ['if' Expr]

Operator        ::= Oper | '='
OperatorNE      ::= Oper
IdOrOperator    ::= Id | Operator
IdOrOperatorNE ::= Id | OperatorNE

ExprList        ::= Expr {',' Expr}
Expr            ::= PostfixExpr
ExprNE          ::= PostfixExprNE

PostfixExpr    ::= InfixExpr [IdOrOperator]
InfixExpr      ::= PrefixExpr
                 | InfixExpr IdOrOperator InfixExpr
PrefixExpr     ::= SimpleExpr
                 | Operator PrefixExpr
                 | '`' Id '`' PrefixExpr
SimpleExpr     ::= SimpleExpr '(' [ExprList] ')'
                 | SimpleExpr '.' IdOrOperator
                 | SimpleExpr '.' '(' ')'
                 | BasicExpr
PostfixExprNE  ::= InfixExprNE [IdOrOperatorNE]
InfixExprNE    ::= PrefixExprNE
                 | InfixExprNE IdOrOperatorNE InfixExprNE
PrefixExprNE   ::= SimpleExprNE
                 | OperatorNE PrefixExpr
                 | '`' Id '`' PrefixExprNE
SimpleExprNE   ::= SimpleExprNE '(' [ExprList] ')'
                 | SimpleExprNE '.' IdOrOperatorNE
                 | SimpleExprNE '.' '(' ')'
                 | BasicExpr
BasicExpr      ::=
                 | Id
                 | '(' Expr ')'
                 | LambdaExpr
                 | 'this' | 'null' | 'true' | 'false'
                 | DecimanNum | BinNum | OctalNum | HexNum
                 | Character | String
LambdaExpr ::= '(' 'fun' [ClosureParams] [Params] [Type] FunBody ')'
             | '(' 'fun' [ClosureParams] [Params] [Type] '=' Expr ')'
ClosureParams ::= '.' '{' [IdList] '}'

Stmt ::= ';'
       | Expr ';'
       | '{' {Stmt} '}'
       | IfStmt
       | ForStmt
       | WhileStmt
       | 'break' ';'
       | 'continue' ';'
```

```
         | 'return' [Expr] ';'
         | Declaration

IfStmt    ::= 'if' [Mods] '(' Expr ')' Stmt ['else' Stmt]
ForStmt   ::= 'for' [Mods] '(' Id [Type] '=' Expr ')' Stmt
WhileStmt ::= 'while' [Mods] '(' Expr [';' (Stmt | Expr)] ')' Stmt
```

Appendix B

Havlak loop recognition in Sparrow

```
import SL.Vector;
import SL.List;
import SL.Map;
import SL.Set;
import SL.Utils;

// BasicBlockEdge only maintains two pointers to BasicBlocks.
class BasicBlockEdge {
  fun ctor(cfg: @MaoCFG, fromName, toName: Int) {
    from ctor cfg.createNode(fromName);
    to   ctor cfg.createNode(toName);
    from->addOutEdge(to);
    to->addInEdge(from);
    cfg.addEdge(this);
  }
  var from, to: BasicBlock Ptr;
}

// BasicBlock only maintains a vector of in-edges and
// a vector of out-edges.
class BasicBlock {
  fun ctor(name: Int) {
    this.name ctor name;
  }
  fun numPred = inEdges.size;
  fun numSucc = outEdges.size;
  fun addOutEdge(to: BasicBlock Ptr)   = outEdges += to;
  fun addInEdge(from: BasicBlock Ptr)  = inEdges += from;
  fun inEdgesRange   = inEdges all;
  fun outEdgesRange = outEdges all;
  fun hash = name;
  private var inEdges, outEdges: Vector(BasicBlock Ptr);
  private var name: Int;
}

// MaoCFG maintains a list of nodes.
class MaoCFG
{
  fun dtor {
    basicBlockMap.values delete;
    edges.all delete;
  }
  fun createNode(name: Int): BasicBlock Ptr {
    var node: @Ptr(BasicBlock) = basicBlockMap(name);
    if ( !node ) {
      node = new(BasicBlock, name);
    }
    if ( basicBlockMap.size == 1 )
      startNode = node;
    return node;
  }
  fun addEdge(edge: BasicBlockEdge Ptr)    = edges += edge;
  fun getNumNodes                          = basicBlockMap.size;
  fun basicBlocksRange = basicBlockMap.values();
  var startNode: BasicBlock Ptr;
  private var basicBlockMap: Map(Int, BasicBlock Ptr);
  private var edges: List(BasicBlockEdge Ptr);
}

// SimpleLoop
// Basic representation of loops, a loop has an entry point,
// one or more exit edges, a set of basic blocks, and potentially
// an outer loop - a "parent" loop.
// Furthermore, it can have any set of properties, e.g.,
// it can be an irreducible loop, have control flow, be
// a candidate for transformations, and what not.
class SimpleLoop {
  fun addNode(bb: BasicBlock Ptr)          = basicBlocks += bb;
  fun addChildLoop(loop: SimpleLoop Ptr)   = children += loop;
  fun setParent(parent: SimpleLoop Ptr) {
    this.parent = parent;
    this.parent->addChildLoop(this);
  }
  fun setNestingLevel(level: Int) {
    nestingLevel = level;
    isRoot = (level == 0);
  }
  fun hash = counter;
  fun >>(os: @OutStream) {
    os << "loop-" << counter << ", nest: " << nestingLevel
      << ", depth: " << depthLevel << endl;
  }
  var children: Set(SimpleLoop Ptr);
  var isRoot: Bool;
  var nestingLevel: Int;
  var depthLevel: Int;
  var counter: Int;
  private var parent: SimpleLoop Ptr;
  private var basicBlocks: Set(BasicBlock Ptr);
}

// LoopStructureGraph
// Maintain loop structure for a given CFG.
// Two values are maintained for this loop, depth, and nesting level.
// For example:
// loop        nesting level    depth
//----------------------------------------
// loop-0      2                0
//   loop-1    1                1
//   loop-3    1                1
//     loop-2  0                2
class LoopStructureGraph {
  fun ctor {
    loopCounter ctor 1;
    root ctor new(SimpleLoop);
    root->counter = 0;
    root->setNestingLevel(0);
    loops += root;
  }
  fun dtor = killAll;
  fun killAll {
    for ( loop = loops.all )
      loop.release;
    loops.clear;
  }
  fun createNewLoop: SimpleLoop Ptr {
    var loop: SimpleLoop Ptr = new(SimpleLoop);
    loop->counter = (loopCounter++);
    return loop;
  }
  fun addLoop(loop: SimpleLoop Ptr)   = loops += loop;
  fun calculateNestingLevel {
    // link up all 1st level loops to artificial root node.
    for ( loop = loops.all )
      if ( !loop->isRoot && !loop->parent )
        loop->setParent(root);

    // recursively traverse the tree and assign levels.
    calculateNestingLevelRec(root, 0);
  }
  private fun calculateNestingLevelRec(loop:SimpleLoop Ptr,depth:Int)
     {
    loop->depthLevel = depth;
    for ( c = loop->children.all ) {
      calculateNestingLevelRec(c, depth+1);
      loop->setNestingLevel(max(loop->nestingLevel, 1+c->nestingLevel
        ));
    }
  }
  fun numLoops = loops.size;
  fun >>(os: @OutStream)                = dumpRec(os, root, 0);
  private fun dumpRec(os: @OutStream, loop: SimpleLoop Ptr, indent:
    Int) {
    os << loop.get;
    for ( c = loop->children.all )
      dumpRec(os, c, indent+1);
  }
  private var root: SimpleLoop Ptr;
  private var loops: List(SimpleLoop Ptr);
  private var loopCounter: Int;
}

// Union/Find algorithm after Tarjan, R.E., 1983, Data Structures
// and Network Algorithms.
class UnionFindNode {
  // Initialize this node.
  fun init(bb: BasicBlock Ptr, dfsNumber: Int) {
    this.parent     ctor this;
    this.bb         ctor bb;
    this.dfsNumber  ctor dfsNumber;
  }

  // Union/Find Algorithm - The find routine.
  // Implemented with Path Compression (inner loops are only
  // visited and collapsed once, however, deep nests would still
  // result in significant traversals).
  fun findSet: UnionFindNode Ptr {
    var nodeList: List(UnionFindNode Ptr);

    var node: UnionFindNode Ptr = this;
    while ( node != node->parent ; node = node->parent ) {
      if ( node->parent != node->parent->parent )
        nodeList += node;
    }
    for ( n = nodeList.all )
      n->parent = node->parent;
    return node;
  }

  // Union/Find Algorithm - The union routine.
  // We rely on path compression.
  fun union(b: UnionFindNode Ptr) {
    parent = b;
  }
  var bb: BasicBlock Ptr;
  var loop: SimpleLoop Ptr;
  var dfsNumber: Int;
  private var parent: UnionFindNode Ptr;
}

//-------------------------------------------------------------------
```

```
// Loop Recognition
// based on:
//   Paul Havlak, Nesting of Reducible and Irreducible Loops,
//      Rice University.
//   We avoid doing tree balancing and instead use path compression
//   to avoid traversing parent pointers over and over.
//   Most of the variable names and identifiers are taken literally
//   from this paper (and the original Tarjan paper mentioned above).
//-------------------------------------------------------------------
class HavlakLoopFinder {
  // Local types used for Havlak algorithm, all carefully
  // selected to guarantee minimal complexity.
  using NodeVector = UnionFindNode Vector;
  using BasicBlockMap = Map(BasicBlock Ptr, Int);
  using IntList = Int List;
  using IntSet = Int Set;
  using NodeList = (UnionFindNode Ptr) List;
  using IntListVector = IntList Vector;
  using IntSetVector = IntSet Vector;
  using IntVector = Int Vector;
  using CharVector = Byte Vector;

  using BB_TOP            = Byte(0);  // uninitialized
  using BB_NONHEADER      = Byte(1);  // a regular BB
  using BB_REDUCIBLE      = Byte(2);  // reducible loop
  using BB_SELF           = Byte(3);  // single BB loop
  using BB_IRREDUCIBLE    = Byte(4);  // irreducible loop
  using BB_DEAD           = Byte(5);  // a dead BB
  using BB_LAST           = Byte(6);  // Sentinel

  using kUnvisited = -1;              // Marker for uninitialized nodes.
  using kMaxNonBackPreds = 32*1024;   // Safeguard against pathologic algorithm behavior.

  fun ctor(cfg: @MaoCFG, lsg: @LoopStructureGraph) {
    this.cfg := cfg;
    this.lsg := lsg;
  }

  // IsAncestor
  // As described in the paper, determine whether a node 'w' is a
  // "true" ancestor for node 'v'.
  // Dominance can be tested quickly using a pre-order trick
  // for depth-first spanning trees. This is why DFS is the first
  // thing we run below.
  fun isAncestor(w, v: SizeType, last: @IntVector) = (w <= v) && (v <= last(w));

  // DFS - Depth-First-Search
  // DESCRIPTION:
  // Simple depth first traversal along out edges with node numbering
  fun doDFS(currentNode: BasicBlock Ptr, nodes: @NodeVector, number: @BasicBlockMap, last: @IntVector, current: Int): Int {
    nodes(current).init(currentNode, current);
    number(currentNode) = current;
    var lastId = current;
    for ( target = currentNode->outEdgesRange ) {
      if ( number(target) == kUnvisited )
        lastId = doDFS(target, nodes, number, last, lastId+1);
    }
    last(number(currentNode)) = lastId;
    return lastId;
  }

  // FindLoops
  // Find loops and build loop forest using Havlak's algorithm, which
  // is derived from Tarjan. Variable names and step numbering has
  // been chosen to be identical to the nomenclature in Havlak's
  // paper (which is similar to the one used by Tarjan).
  fun findLoops {
    if ( !cfg.startNode )
      return;
    var size = cfg.getNumNodes;
    var nonBackPreds: IntSetVector = size;
    var backPreds: IntListVector = size;
    var header: IntVector = size;
    var type: CharVector = size;
    var last: IntVector = size;
    var nodes: NodeVector = size;
    var number: BasicBlockMap;

    // Step a:
    //   - initialize all nodes as unvisited.
    //   - depth-first traversal and numbering.
    //   - unreached BB's are marked as dead.
    for ( bb = cfg.basicBlocksRange )
      number.insert(bb, kUnvisited);

    doDFS(cfg.startNode, nodes, number, last, 0);

    // Step b:
    // - iterate over all nodes.
    //   A backedge comes from a descendant in the DFS tree, and
    //   non-backedges from non-descendants (following Tarjan).
    //   - check incoming edges 'v' and add them to either
    //     - the list of backedges (backPreds) or
    //     - the list of non-backedges (nonBackPreds)
    for ( w = 0..size ) {
      header(w) = 0;
      type(w) = BB_NONHEADER;
      var nodeW: BasicBlock Ptr = nodes(w).bb;
      if ( !nodeW ) {
        type(w) = BB_DEAD;
        continue;   // dead BB
      }
      if ( nodeW->numPred > 0 ) {
        for ( nodeV = nodeW->inEdgesRange ) {
          var v = number(nodeV);
          if ( v == kUnvisited )
            continue;    // dead node
          if ( isAncestor(w, v, last) )
            backPreds(w) += v;
          else
            nonBackPreds(w) += v;
        }
      }
    }
    // Start node is root of all other loops.
    header(0) = 0;

    // Step c:
    // The outer loop, unchanged from Tarjan. It does nothing except
    // for those nodes which are the destinations of backedges.
    // For a header node w, we chase backward from the sources of the
    // backedges adding nodes to the set P, representing the body of
    // the loop headed by w.
    // By running through the nodes in reverse of the DFST preorder,
    // we ensure that inner loop headers will be processed before the
    // headers for surrounding loops.
    for ( w = Int(size-1)...0 ../ (-1) ) {
      var nodePool: NodeList;        // this is 'P' in Havlak's paper
      var nodeW = nodes(w).bb;
      if ( !nodeW )
        continue;    // dead BB

      // Step d:
      for ( v = backPreds(w).all ) {
        if ( v != w )
          nodePool += nodes(v).findSet;
        else
          type(w) = BB_SELF;
      }

      // Copy node_pool to worklist.
      var workList: NodeList = nodePool.all;
      if ( nodePool.isEmpty )
        type(w) = BB_REDUCIBLE;

      // work the list...
      while ( !workList.isEmpty ) {
        var x = workList.front;
        workList.popFront;

        // Step e:
        // Step e represents the main difference from Tarjan's method
        // Chasing upwards from the sources of a node w's backedges.
        // If there is a node y' that is not a descendant of w, w is
        // marked the header of an irreducible loop, there is another
        // entry into this loop that avoids w.
        // The algorithm has degenerated. Break and
        // return in this case.
        var nonBackSize = nonBackPreds(x->dfsNumber).size;
        if ( nonBackSize > kMaxNonBackPreds ) {
          lsg.killAll;
          return;
        }
        for ( t = nonBackPreds(x->dfsNumber).all ) {
          var y: @UnionFindNode = nodes(t);
          var yDash = y.findSet;
          if ( !isAncestor(w, yDash->dfsNumber, last) ) {
            type(w) = BB_IRREDUCIBLE;
            nonBackPreds(w).insert(yDash->dfsNumber);
          } else {
            if ( yDash->dfsNumber != w ) {
              if ( find(nodePool.all, yDash).isEmpty ) {
                workList += yDash;
                nodePool += yDash;
              }
            }
          }
        }
      }
      // Collapse/Unionize nodes in a SCC to a single node
      // For every SCC found, create a loop descriptor and link it in
      if ( !nodePool.isEmpty || type(w) == BB_SELF ) {
        var loop = lsg.createNewLoop;

        // At this point, one can set attributes to the loop, such as
        // the bottom node:
        //    IntList::iterator iter  = back_preds[w].begin;
        //    loop bottom is: nodes[*backp_iter].node);
        // the number of backedges:
        //    back_preds[w].size
        // whether this loop is reducible:
        //    type[w] != BB_IRREDUCIBLE
        // TODO(rhundt): Define those interfaces in the Loop Forest.
        nodes(w).loop = loop;
        for ( node = nodePool.all ) {
          header(node->dfsNumber) = w;
          node->union(nodes(w));
          if ( node->loop.isSet )
            node->loop->setParent(loop);
          else
            loop->addNode(node->bb);
        }
        lsg.addLoop(loop);
      }
    }
  }
  private var cfg: @MaoCFG;
  private var lsg: @LoopStructureGraph;
}

// External entry point.
fun findHavlakLoops(cfg:@MaoCFG, lsg:@LoopStructureGraph):SizeType {
  var finder = HavlakLoopFinder(cfg, lsg);
  finder.findLoops;
  return lsg.numLoops;
}

//--- TESTING CODE --------------------
fun buildDiamond(cfg: @MaoCFG, start: Int): Int {
  new(BasicBlockEdge, cfg, start  , start+1);
  new(BasicBlockEdge, cfg, start  , start+2);
  new(BasicBlockEdge, cfg, start+1, start+3);
  new(BasicBlockEdge, cfg, start+2, start+3);
  return start+3;
}
fun buildConnect(cfg: @MaoCFG, start, end: Int) {
  new(BasicBlockEdge, cfg, start, end);
}
fun buildStraight(cfg: @MaoCFG, start, n: Int): Int {
  for ( i = 0..n )
    buildConnect(cfg, start + i, start + i + 1);
  return start + n;
}
fun buildBaseLoop(cfg: @MaoCFG, from: Int): Int {
  var header   = buildStraight(cfg, from, 1);
  var diamond1 = buildDiamond(cfg, header);
  var d11      = buildStraight(cfg, diamond1, 1);
  var diamond2 = buildDiamond(cfg, d11);
  var footer   = buildStraight(cfg, diamond2, 1);
  buildConnect(cfg, diamond2, d11);
  buildConnect(cfg, diamond1, header);
  buildConnect(cfg, footer, from);
  footer = buildStraight(cfg, footer, 1);
  return footer;
}

fun sprMain {
  cout << "Welcome to LoopTesterApp, Sparrow edition" << endl;
  cout << "Constructing cfg..." << endl;
  var cfg: MaoCFG;
  cout << "Constructing lsg..." << endl;
  var lsg: LoopStructureGraph;
  cout << "Constructing Simple CFG..." << endl;
  cfg.createNode(0);  // top
  buildBaseLoop(cfg, 0);
  cfg.createNode(1);  // bottom
  new(BasicBlockEdge, cfg, 0, 2);
  cout << "15000 dummy loops" << endl;
  for ( dummyLoop = 0..15000 ) {
    findHavlakLoops(cfg, LoopStructureGraph());
  }
  cout << "Constructing CFG..." << endl;
  var n = 2;
  for ( parlooptrees = 0..10 ) {
    cfg.createNode(n + 1);
    buildConnect(cfg, 2, n + 1);
```

```
      ++n;
      for ( i = 0..100 ) {
        var top = n;
        n = buildStraight(cfg, n, 1);
        for ( j = 0..25 )
          n = buildBaseLoop(cfg, n);
        var bottom = buildStraight(cfg, n, 1);
        buildConnect(cfg, n, top);
        n = bottom;
      }
      buildConnect(cfg, n, 1);
    }
    cout << "Performing Loop Recognition\n1 Iteration" << endl;
    var numLoops = findHavlakLoops(cfg, lsg);
    cout << "Another 50 iterations..." << endl;
    var numIterations = programArgs(1) asInt;
    var sum: SizeType = 0;
    for ( i = 0..numIterations ) {
      cout << '.' << flush;
      sum += findHavlakLoops(cfg, LoopStructureGraph());
    }
    cout << endl << "Found " << numLoops << " loops (including
      artificial root node)(" << sum << ')' << endl;
    cout << lsg;
  }
```

APPENDIX C

N-Queens at compile-time in Sparrow

In this appending we list the full solution of the NQueens problem discussed in chapter 10 along with all the test code that we've used.

```
import SL.Array;
import SL.Vector;
import SL.String;
import Math;
import OS;

//using runAtCt = false;
//using printRt = 0;
using numQueens = 4;
using autoTestMode = true;

using PlacementType = Array(Int);
using SolutionsType = Vector(PlacementType);

fun[rtct] testQueens(placements: @PlacementType, k, y: Int): Bool {
    for ( i = 0..k ) {
        if ( y == placements(i) || k-i == Math.abs(y-placements(i)) )
            return false;
    }
    return true;
}
fun[rtct] backtracking(curSol: @PlacementType, k, n: Int,
                          res: @SolutionsType) {
    for ( y = 0..n ) {
        if ( testQueens(curSol, k, y) ) {
            curSol(k) = y;
```

```
                if ( k == n-1 )
                    res.pushBack(curSol);
                else
                    backtracking(curSol, k+1, n, res);
            }
        }
}
fun[rtct] nQueens(n: Int): SolutionsType {
    var res: SolutionsType;
    var curSol: PlacementType = n;
    backtracking(curSol, 0, n, res);
    return res;
}

fun[autoCt] writeSolutions(solutions: @SolutionsType) {
    cout << "We have " << solutions.size() << " solutions\n";
    for ( sol = solutions.all() )
        writeSolution(sol);
}
fun[rtct] writeSolution(x: @PlacementType) {
    for ( v = x.all() ) {
        for ( i = 0..v )
            cout << "| ";
        cout << "|Q";
        for ( i = (v+1)..(x.size()) )
            cout << "| ";
        cout << "|\n";
    }
    cout << "\n";
}
fun[autoCt] prettyPrint(solutions: @SolutionsType) {
    // line 0           N = number of lines per board square
    // lines 1..N       the characters for a "white" board square
    // lines N+1..2N    the characters for a "black" board square
    // lines 2N+1..3N   the characters for a queen on white square
    // lines 3N+1..4N   the characters for a queen on black square
    var lines = readFileLines("queen-art.txt");
    if ( lines.isEmpty() ) OS.exit(-1);
    var n = asInt(lines(0).asStringRef());
    if ( lines.size() < 1+4*n ) OS.exit(-1);
    cout << "We have " << solutions.size() << " solutions\n";
    for ( sol = solutions.all() ) {
        for ( line = 0..sol.size() ) {
            var v = sol(line);
            for ( l = 0..n ) {
                for ( col = 0..sol.size() ) {
                    var isQueen = col==v;
                    var isWhite = (line+col)%2==0;
                    var idx = ife(isWhite, 1+l, 1+l+n) + ife(isQueen,
                       2*n, 0);
                    cout << lines(idx);
                }
                cout << endl;
            }
        }
        cout << "\n\n";
    }
}
fun[rtct] readFileLines(filename: StringRef): Vector(String) {
    var lines: Vector(String);
    var f = OS.File.open(filename, "r");
    if ( !f.isOpen() ) {
        cout << "Cannot open file: " << filename << "\n";
        return lines;
    }
    while ( !f.isEof() ) {
        var s = f.readLine();
        if ( !s.isEmpty() ) {
            s.resize(s.size()-1);   // Cut the end of line
            lines.pushBack(s);
        }
    }
    return lines;
}

// Different conversions from CT to RT

fun[rt] toRt(sols: SolutionsType ct): SolutionsType {
    var solsRt = sols;
    return solsRt;
}
fun[rt] toRt1(sols: SolutionsType ct): SolutionsType {
    var res: SolutionsType;
    res.reserve(ctEval(sols.size()));
    for[ct] ( sol = sols.all() ) {
        var[ct] size = sol size;
        var arr = PlacementType(ctEval(size));
        for[ct] ( i = 0..size )
            arr(i) = sol(i);

        res.pushBack(arr);
    }
    return res;
}
fun[rt] toRt2(sols: SolutionsType ct): SolutionsType {
    var res: SolutionsType;
    res.reserve(ctEval(sols.size()));
    for[ct] ( sol = sols.all() ) {
        var[ct] size = sol size;
        var arr = PlacementType(ctEval(size));
        for[ct] ( i = 0..size )
            arr(i) = ctEval(sol(ctEval(i)));
        res.pushBack(arr);
    }
    return res;
}
fun[rt] toRt3(sols: SolutionsType ct): SolutionsType {
    var res: SolutionsType;
    res.reserve(ctEval(sols.size()));
    for[ct] ( sol = sols.all() ) {
        var[ct] size = sol size;
        var arr = PlacementType(ctEval(size));
        for[ct] ( i = 0..size ) {
            arr(i) = sol(i);
            cout << res.size() << endl;
            cout << arr.size() << endl;
        }
        res.pushBack(arr);
    }
    return res;
}
fun[ct] toStringBuffer(sols: SolutionsType): String {
    var numSols = sols.size();
    var n = ife(numSols == 0, 1, sols(0).size());

    var res: String = 1 + numSols*n;
    res(0) = Char(n);
    for ( i = 0..numSols*n ) {
        var solNum = i / n;
        var qNum = i % n;
        res(1+i) = Char(sols(solNum)(qNum));
    }
    return res;
}
fun[rt] fromStringBuffer(sols: String): SolutionsType {
    var n: Int = sols(0);
    var numSols = (sols.size()-1) / n;
    var res: SolutionsType = repeat(PlacementType(n), numSols);
    for ( i = 0..numSols*n ) {
        var solNum = i / n;
        var qNum = i % n;
        res(solNum)(qNum) = Int( sols(1+i) );
    }
    return res;
}
fun[rt] toRt4(sols: SolutionsType ct): SolutionsType {
    return fromStringBuffer(toStringBuffer(sols));
}

fun sprMain {
    if[ct] ( autoTestMode ) {
        var[ct] solutionsCt = nQueens(numQueens);
        var solutions = solutionsCt;
        writeSolutions(solutionsCt);
        writeSolutions(solutions);
    }
    else if[ct] ( isValid(runAtCt) ) {
        var[ct] solutionsCt = nQueens(numQueens);
        if[ct] ( isValid(printRt) )
        {
            if[ct] ( printRt == 0 )
                prettyPrint(toRt(solutionsCt));
            if[ct] ( printRt == 1 )
                prettyPrint(toRt1(solutionsCt));
            if[ct] ( printRt == 2 )
                prettyPrint(toRt2(solutionsCt));
            if[ct] ( printRt == 3 )
                prettyPrint(toRt3(solutionsCt));
            if[ct] ( printRt == 4 )
                prettyPrint(toRt4(solutionsCt));
        }
        else
            prettyPrint(solutionsCt);
    }
    else {
        var n = numQueens;
        prettyPrint(nQueens(n));
    }
}

/*<<<Running()
We have 2 solutions
| |Q| | |
| | | |Q|
|Q| | | |
| | |Q| |

| | |Q| |
|Q| | | |
| | | |Q|
| |Q| | |

>>>*/
```

APPENDIX D

N-Queens in C++, with constexpr

Here we present the the code in C++ that would solve at compile-time the N-Queens problem. This code handles just the computation of the solution, without any printing. Please see chapter 10 for more details.

```cpp
#include <iostream>

using namespace std;

#ifndef NUM_QUEENS
const int NUM_QUEENS = 4;
#endif

// 64-bit to represent all the queen positions.
// we use 4-bit per row to indicate the position of the queen
// => max 16 rows, max 32 columns
typedef unsigned long long Placements;
struct Result {
    int numSol;
    Placements aSolution;
    constexpr Result() : numSol(0), aSolution(0)
    { }
    constexpr Result(Placements sol) : numSol(1) , aSolution(sol)
    { }
    constexpr Result(int numSolutions, Placements sol)
        : numSol(numSolutions) , aSolution(sol)
    { }
    constexpr bool isValid() const {
        return numSol>0;
    }
    constexpr Result mergeWith(Result other) {
       return Result(numSol+other.numSol,
            numSol>0 ? aSolution : other.aSolution);
    }
};
constexpr int myabs(int x) {
    return x < 0 ? -x : x;
}
constexpr int getQueenPos(Placements plc, int row) {
    return (plc >> (row*4)) & 0x0f;
}
constexpr Placements setQueenPos(Placements plc, int row, int col) {
    return (plc & (~(Placements(0x0f) << (row*4))))
        | (Placements(col & 0x0f) << (row*4));
}
constexpr bool testQueen(Placements plc,
    int row, int col, int testRow = 0) {
    return testRow == row
        || (
            getQueenPos(plc, testRow) != col
            && myabs(row-testRow) !=
                myabs(col-getQueenPos(plc, testRow))
            && testQueen(plc, row, col, testRow+1)
        );
}

constexpr Result nQueens(int n, Placements curSol = 0,
                         int row = 0, int col = 0) {
    return row == n
        ? Result(curSol)
        : col == n
            ? Result()
            : (
                testQueen(curSol, row, col)
                    ? nQueens(n,
                        setQueenPos(curSol, row+1, 0),
                        row+1, 0)
                    : Result()
            ).mergeWith(
                nQueens(n,
                    setQueenPos(curSol, row, col+1),
                    row, col+1)
            );
}

int main(int, char**) {
    constexpr int n = NUM_QUEENS;

    constexpr Result res = nQueens(n);
    static_assert(res.numSol >= 2, "We must have some solutions");
    cout << "Num solutions: " << res.numSol << endl;
    cout << "A solution: ";
    for ( int i=0; i<n; ++i )
        cout << getQueenPos(res.aSolution, i) << " ";
    cout << endl;

    return 0;
}
```

APPENDIX E

RELEVANT RESEARCH PAPERS

We present here two relevant research papers for our thesis:

- Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “General Efficiency Analysis for the Sparrow Programming Language”. In: *20th International Conference on Control Systems and Computer Science (CSCS)*. 2015. Best paper award.
- Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. “Moving Computations from Run-time to Compile-time: Hyper-metaprogramming in Practice”. In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. 2014.