KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

Type Constructor Polymorphism for Scala: Theory and Practice

Promotoren :
Prof. Dr. ir. W. JOOSEN
Prof. Dr. ir. F. PIESSENS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Adriaan MOORS

May 2009

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

Type Constructor Polymorphism for Scala: Theory and Practice

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Adriaan MOORS

U.D.C. 681.3*D33

May 2009

D/2009/7515/48
ISBN 978-94-6018-065-1 (printed version)

Abstract

A static type system is an important tool in efficiently developing correct software. The recent introduction of "genericity" in object-oriented programming languages has greatly enhanced the expressiveness of their type systems. Genericity, which is also called "parametric polymorphism", is extremely useful as it allows the definition of polymorphic lists, which use a type parameter to abstract over the type of their elements. However, as genericity is first-order, we cannot again abstract over the type of these parameterised lists. We generalised Scala's support for parametric polymorphism to the higher-order case, as this additional power turns out to be useful in practice. We call the result "type constructor polymorphism", as Scala programmers may now safely abstract over type constructors, such as the type of polymorphic lists. We describe the theoretical underpinnings as well as the practical side of our extension of Scala's type system. Our generalisation, amplified by the synergy with Scala's existing features such as implicits, represents an important asset in the library designer's abstraction-building tool belt, while the user of these abstractions need not worry about their inner workings. The theoretical side of the story focusses on the lacunae in the existing Scala formalisms, and presents Scalina, our core calculus that solves these. Finally, we elaborate on our vision for future improvements of the type system, based on our work on Scalina and practical experience with type constructor polymorphism.

Acknowledgements

I remember very well the thrill of Wouter Joosen hiring me in the summer of 2004, and would like to thank him, as well as my other supervisor, Frank Piessens, for keeping that feeling alive — inspiring, supporting, and challenging me along the way. Thank you, Wouter. Thank you, Frank. It's been a great ride, and I'm sure it will continue to be.

I should mention that part of that support came from a research grant from the *Institute for the promotion of Innovation by Science and Technology* in Flanders (IWT-Flanders).

Besides my supervisors, I would like to extend my gratitude to the following colleagues for providing me with valuable feedback: Marko van Dooren, Dave Clarke, Bart Jacobs, Jan Smans, Erik Ernst, Andreas Rossberg, Eddy Truyen, and the anonymous reviewers that diligently and selflessly improved my papers over the years.

My internship with Martin Odersky at the École Polytechnique Fédérale de Lausanne was an essential phase of my research — a turning point even — when everything came together and concrete results finally appeared. Thank you, Martin, for the thrilling, rewarding challenges you provided me with, and the great co-operation ever since. Also, thank you for allowing me to escape Java's clutches to rediscover the joy of programming in Scala. Of course, my stay in Lausanne would not have been the great experience it was without the interesting and fun people whom I met there. So, thank you, Burak, Gilles, Ingo, Iulian, Lex, Philipp, and Sean!

Once the chain reaction was set off in Lausanne, time really began to fly. The first internship sparked the next one — at Microsoft Research, Cambridge — as Don Syme happened to be visiting Lausanne. Thank you, Don, for another glimpse of language development with a vision, grounded in the real world. Again, the internship entailed meeting a lot of great people. Carsten, Mike, Neel, Otmar, and Pat: cheers, guys!

I am proud and grateful to count these colleagues among my friends. I would also like to thank my other friends, who brightened my life outside the office: Peter, Thomas, Dave, Thijs&Stéphanie, Rein&Goedele, Joaquin&Veerle, Fiona&Koen, Thomas&Sara, Bobby&Kathy, Ilse&Bram, Geraldine&Gijs, Marko, Batist, Kim, Lieselot, as well as my sister, Anne-Katrien, and her fiancé, Stijn. Please accept my apologies if you were not mentioned explicitly, my only excuse is the mental exhaustion often experienced when nearing the end of writing a dissertation. Finally, I would like to thank my family, and especially my parents, Gerard and Marie-Michèle, without whom none of this would have been possible, in every possible sense. Bedankt voor jullie onvoorwaardelijke steun en liefde!

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Essentially, a programming language allows a programmer to implement a solution to a certain problem. Furthermore, the programmer must be able to accurately describe the problem, so that the language can verify whether the solution indeed solves the problem at hand. Of these three facets — implementation, specification, and verification — our research focusses on the latter two.

Specification and verification are tackled in various ways by different languages. For example, in a dynamically typed language, such as Ruby, the problem is generally specified using unit tests — small programs that execute a part of the implementation and check its result — and verification thus corresponds to running these tests. Similarly, run-time assertions may be used to specify the implementation's desired behaviour, and verification amounts to evaluating these conditions while the program is running. Both techniques detect bugs that arise while the program is running. Thus, in this approach to verification, if a buggy fragment of the program is not executed, the error is not discovered.

In a statically typed language, such as Scala, types are used as a lightweight way of specifying the problem, and verification is performed before the program is run. The expressible properties are typically not as rich as with unit testing or run-time assertions, but verification deals with all possible executions of the program. Soundness of the type system implies that it will detect all violations against the specifications that it can express. The trade-off is that verification is a conservative approximation, in the sense that it may rule out valid programs. However, type checking can be made more flexible to combine the best of both worlds.

A sophisticated type system allows the expert programmer to encapsulate domain knowledge so that the mainstream programmer, as the user of these abstractions, may soundly instantiate them. The type system communicates and enforces rich constraints on how these abstractions may be used: it acts on behalf of the expert programmer so that the number of users of an abstraction scales independently of the availability of the programmer that designed them.

This approach hinges on two equally important requirements: the type system must be powerful enough to allow the expert to express the required constraints, while being unobtrusive to the user: it should be the guiding hand, not the yoke that constrains.

As a simple example, consider *parametric polymorphism*, which most statically-typed object-oriented languages now support as "genericity". This feature allows the expert programmer to accurately express several important abstractions, such as a communication channel that transmits objects of a given type, where that type may be chosen by the user of the channel. The abstraction itself ensures that the channel will indeed only convey this type of object. Another common example is that of polymorphic lists.

Parametric polymorphism can be understood by considering the corresponding construct at the level of values: functions, or, equivalently, methods. A method's result is parameterised by the concrete value of its parameters, which need not be known concretely for the method to be written. However, in order to get an actual result, the method must be supplied with concrete arguments for its parameters. Similarly, a polymorphic class or method can be expressed in terms of its type parameters, which represent unknown types that must be made concrete by the clients of the class or method.

As a concrete example, the successor function is not sensitive to (does not "care" about) the exact value of the number that it increases — it can do so for any number (ignoring overflow). Similarly, the polymorphic class of lists is not sensitive to the exact type of its elements. For any given type, the list can ensure that it only holds element of that type. At the same time, users of the list would like this information to be preserved: they know that they are dealing with a list of integers, for example, and adding another integer to the list should result in a list of integers, while the list should prevent a string from being added.

Now, with parametric polymorphism this kind of consistency can be enforced without defining a new list for every specific type. A polymorphic list simply abstracts over the concrete type of its elements using a type parameter. As such, a polymorphic list can be thought of as a function that operates on types: it takes an arbitrary type T and yields the type of the list that holds elements of that given type T. To ensure consistency, a list of elements of type T only allows another element of type T to be added to the list. This way, when the user retrieves the first element of a list of elements of type T, for example, the list may guarantee that the element has type T, irrespective of the concrete type that this T stands for.

Without parametric polymorphism, the implementer of the list could only express that it holds any object (using subtype polymorphism), and, for every interaction, the user had to check (resorting to run-time casts) that the expected type of object was retrieved. Thus, genericity constitutes a win on both fronts: the expert acquired an important tool, and the user needs not worry about casting anymore. Even though, instead of casting, the user now has to know about type parameters, type inference can correctly determine most type parameters automatically, so that the net effect of introducing genericity into the language is indeed positive.

Finally, the mainstream approach to genericity has a significant shortcoming: it is limited to *first-order* parametric polymorphism. This means that a type parameter can only stand for a type that is not parameterised. Thus, a type that abstracts over a type – a "type *constructor*" – cannot be abstracted over. This may seem like an esoteric restriction, but it turns out that it rules out several relevant object-oriented abstractions.

We generalised Scala's support for parametric polymorphism to the higher-order case, and call the result "type constructor polymorphism", since we can now abstract over type constructors. This extension fits nicely with the Scala philosophy of fusing functional and object-oriented programming [OAC$^+$06, OSV08].

1.1 Contributions

This thesis describes our addition to the tools available to the expert Scala programmer, while shielding the user from the increased complexity. Furthermore, it is not an isolated patch. It is a step along the way to a more powerful as well as a more user-friendly type system. More concretely, our experience with type constructor polymorphism indicates that a more general kind of polymorphism will provide further opportunities for scrapping boilerplate and defining more powerful abstractions in general. This is an exciting avenue for future research.

Concretely, we discuss the following contributions:

- we designed an extension of Scala with type constructor polymorphism

- we implemented the design in the official Scala compiler
- we devised a novel design pattern that relies on our extension and showed how it removes redundant code in Scala's collections library
- we developed a new formalism for Scala that resolves an issue with the encoding of our extension in the existing formal model

Our design smoothly generalises Scala's existing support for parametric polymorphism to the higher-order case. It is fully backwards compatible, with minor syntactic changes and no impact on the performance of compiled programs. We implemented our design in the official Scala compiler, including support for higher-order subtyping and (definition-site) variance, type constructor inference, and kind inference and kind checking.

To validate our extension, we devised the Builder pattern, which encapsulates the creation of collections. Briefly, it uses type constructor polymorphism to abstract over the collection that it builds, so that this behaviour can be implemented once. Thus, methods that produce different types of collections in various ways (e.g., `map`, `filter`, `slice`,...), no longer need to be duplicated for every type of collection that they create.

A more thorough validation was recently performed by Martin Odersky, when refactoring Scala's collections library. The library could be cleaned up significantly, with minimal impact on existing users. Moreover, the refactoring used a novel style of programming, which revealed interesting opportunities to further refine and extend the type system. This experience has validated the utility of our extension and the corresponding programming methodology in a realistic and intricate setting. At the same time, the encountered limitations indicate that our proposed refinements — such as kind polymorphism — are warranted. Finally, other researchers have since used type constructor polymorphism in their own work [HORM08].

Besides the practical validation, we have developed a theoretical underpinning for our extension. We first experimented with encoding it in an existing Scala formalism, the νObj calculus [OCRZ03], but discovered a subtle, though fundamental, shortcoming of the formalism that precludes a faithful encoding of type parameters as their object-oriented counterpart, abstract type members. Thus, we developed a new object-oriented calculus that emphasises uniform support for abstraction at the value and type levels, and that fully subsumes parameterisation, the functional style of abstraction.

To briefly touch on our plans for future work, we observe the trend of introducing abstraction mechanisms at ever-increasing levels: first, genericity allows abstracting over types (but not type constructors) at the level of types, we generalised this to allow abstracting over type constructors, and note that abstracting over kinds would lift certain limitations of type constructor polymorphism, such as the verbosity of abstracting over bounds, or even the impossibility of abstracting over variance annotations.

Instead of keeping with these incremental improvements, it seems appropriate to investigate a mechanism that escapes this cadence, that incorporates this trend of an abstraction mechanism at level N introducing the need for an abstraction mechanism at level $N+1$. Thus, we see level polymorphism, which Sheard is investigating in Ωmega [She08], as an important direction of future research. In a sense, Scalina foreshadows this by striving to provide a uniform way of abstracting over values and types at the level of types and values alike. We will elaborate on this in chapters 2 and 5.

1.2 Other work

The core contribution that is discussed explicitly in this thesis was realised in the broader context of programming language research, which resulted in several other contributions to the field:

- the exploration of datatype-generic programming in Scala [MPJ06, Moo07], which was the original motivation for extending Scala with direct support for type constructor polymorphism;
- maintenance and further development of Scala's library for combinator parsing [MPO08b], and a novel (experimental) library of combinators to specify variable scoping as part of the grammar, which uses a lightweight approach to data-type generic programming to reduce the boilerplate that is normally required to correctly implement substitution and other aspects of variable binding – moreover, this library also provides a domain-specific language to express typing rules, factoring out the algorithmic details using a monad for logic programming;
- collaboration on an approach to polymorphically embed domain-specific languages in Scala [HORM08] – the parser combinators are a concrete example of this idea.

1.3 Structure of the thesis

The following chapter sets the scene for the rest of this thesis; it motivates why the type system is an important asset in producing correct software, it introduces Scala, and it discusses research related to type constructor polymorphism.

Chapter 3 presents type constructor polymorphism, its utility, soundness, and feasibility. More concretely, we discuss Scala's collection library as a realistic application of type constructor polymorphism, we introduce the underlying theory, and we show that its integration in an object-oriented language makes the type system more powerful, at a modest cost in complexity from the viewpoint of the users as well as the language implementer.

Chapter 4 introduces Scalina, a purely object-oriented calculus that introduces the missing link for an object-oriented calculus to faithfully encode functional-style abstraction. Thus, it provides the formal underpinning of type constructor polymorphism. Furthermore, we use it for our research on uniformly abstracting over types and values at the level of values and types.

Chapter 5 reflects on the presented contributions and sketches the domain in which the continuation of our research is situated.

Chapter 2

Background and State of the Art

In a broader sense, our thesis is that the type system is an essential tool in producing high-quality software. The following chapters discuss our concrete contributions, which are focussed on improving a certain aspect of Scala's type system. In this chapter, we explore the research area that is concerned with specifying and verifying functional correctness of software. We focus on correctness as a measure for software quality, ignoring the plethora of other metrics, which are often much harder to measure objectively. In Chapter 5 we will return to this discussion, and describe in more detail the "ideal" type system that we are working towards.

Besides actually implementing it, producing correct software can be split in two tasks: specification and verification. These tasks should be performed in the same modular fashion as the implementation. Clearly, we require concise and precise specifications that are easy to write and understand. Of course, verification must accurately detect discrepancies between the implementation and its specification. More interestingly, though, verification must be flexible in order to match the importance that is attributed to the correctness of a certain module at a certain point in the development process. Roughly speaking, if it is too costly to specify simple correctness criteria, the approach will not be used; if it is easy to specify simple criteria, but impossible to tackle the hard ones, the approach is of limited use.

The problem with current approaches to producing correct software is that they are either easy to use, but unable to verify complex specifications, or else they can express and verify precise specifications, but it is prohibitively costly to do so. Unfortunately, a module typically does not require thorough verification during the prototyping phase, but it will certainly require more quality assurance once the product matures. Currently, this inherent evolution requires an expensive switch to a different verification technique, since none of them is both flexible and powerful enough to deal with both situations efficiently.

In this chapter, we survey the three most widely used techniques to enforcing functional correctness, pointing out where they are lacking. We briefly describe our proposed solution – a scalable type system – but defer a more detailed discussion to Chapter 5, as this is a long-term research objective. Briefly, for a verification approach to be a worthy investment, it must be so powerful as to deal with the complex requirements of crucial modules in a mature product, while being so flexible and scalable as to facilitate applying it from the prototyping stage onwards, evolving smoothly alongside the implementation.

Chapter outline This chapter provides breadth so that the following chapters can focus on depth. We first sketch the broader research area of software specification, in which our work is situated. Since we leveraged Scala as a platform for validating our research, we provide a quick introduction. Finally, we discuss research that relates to the thesis as a whole. More specifically, we introduce the notion of higher-kinded types, trace their heritage in the functional programming community, and underscore their novelty in the context of an object-oriented programming language.

2.1 Efficiently producing correct software

By the divide-and-conquer adage, modular development, which relies on encapsulation, is essential in implementing any non-trivial piece of software. To properly encapsulate a module, its functionality must be described accurately without revealing its inner workings. Thus, to understand a module, a client need not look further than its specification, as it is independently guaranteed to be sound with respect to its implementation. The quality of software – its functional correctness – can be measured objectively by checking that the specification and the implementation correspond.

Several challenges arise in verifying that a module's implementation meets its specification. Individually, specification, implementation, and verification are complex tasks. Ensuring their mutual consistency while evolving them independently is an even greater challenge. The expectations of software necessarily change over time, as does its implementation. Besides the obvious changes in requirements, this evolution also arises from bugs in the specification and the implementation that must be fixed.

The three most commonly used approaches to checking functional correctness are: testing, type checking, and formal verification. A fourth is extracting programs from proofs, but it is not that widely used. In the remainder of this section we discuss these approaches in turn, highlighting their strengths and weaknesses. Section 5.2.1 discusses how these strengths can be integrated into the type system.

2.1.1 Testing

The most direct way of checking whether a program is correct is to simply execute it and check the result. In this approach, the functionality is specified by implementing a suite of tests that each execute a part of the program, and verification simply consists of executing these tests and checking their results.

Since tests are developed in the same language as the program itself, this approach is readily mastered by a programmer. Tests for different parts can be written individually, and failing tests do not block development. This facilitates modular software development and rapid prototyping. Thus, the approach scales well with the desired degree of correctness by varying which test failures are taken into account.

However, testing does not scale well with the complexity of the program and its specification due to the local nature of a test – each exercises only a small fragment of the program. Specifications are programs and verification corresponds to program execution. Thus, specification and verification are coupled relatively tightly. Moreover, there is no significant step up in abstraction in going from program to specification. More precisely, specifications are not quantified over program executions, nor can they abstract over program entities.

Many of the challenges in the testing approach can be traced back to this lack of abstraction. The notion of code coverage springs from the inability of quantifying over program runs. Whereas a type system can provide high-level guarantees that hold for *every execution* of a well-typed program – e.g., that it never calls a non-existent method – testing cannot directly express such a constraint.

Similarly, testing requires concrete input, as it cannot abstract over it. Thus, a specification is only verified for a specific set of values. Moreover, testing a certain module often requires input from another module, e.g., an object that is an instance of a class that is defined in another module. To respect modularity, dependencies on other modules must be limited to the interfaces declared by that module. Thus, the foreign class cannot be used directly, and a mock object must be created that implements the interface. Recent work on automated [GKS05], parameterised [TS06], and assume-guarantee [BGP06] testing aims to address these shortcomings.

2.1.2 Formal verification

A more abstract way of ensuring functional correctness is to introduce a formal logic as a layer of indirection between the program and its specification. Concrete programs are approximated automatically by more high-level formulae in a logic, and specifications express the desired functionality as logical formulae about the encoded program. Finally, verification amounts to showing that the formulae that encode the specification are derivable from the logical encoding of the implementation. Typical approaches to program verification such as JML [LBR06] and Spec$^{\#}$ [BLS05] involve an automated theorem prover to show the derivability of the formulae that constitute the specification. However, they also include tools to derive tests from specifications, or to check certain assertions dynamically.

The expressive power of the approach stems from the employed logic, such as Hoare logic [Hoa83], or its extension to separation logic [Rey02]. One aspect of this expressiveness is quantification over program entities. On one hand, this is a step up from testing, where the lack of this kind of quantification forces the creation of mock objects. Furthermore, static verification reasons about every possible execution of a program at once, thus quantifying over program execution. On the other hand, static verification of these high-level assertions is prone to false positives, whereas tests are based on concrete data, so that a failure is never spurious. In general, static verification allows richer specifications, but they are harder to write and verify. Tests are simpler, though more labour-intensive, to write.

This trade-off between complexity and expressive power is well known in software engineering. Modules and polymorphism are some of the most important tools to battle complexity and spread the investment cost, without sacrificing expressive power. Unfortunately, the logics that are currently used in these approaches do not support all of these language constructs. More precisely, the logics are not higher-order, so that a formula cannot abstract over another formula. It is possible to work around this by duplicating the formula as a pure method, which can then be abstracted over using the programming language's polymorphism. Nonetheless, full support for reuse is essential for formal verification to succeed.

From the perspective of the language designer, another kind of redundancy arises in this approach. Besides the program verification logic, the type system induces a logic of its own. Care must be taken to align them. More precisely, the way a program is approximated by logic formulae in the verification logic should incorporate the correctness criteria that are enforced by the type system. Instead of introducing an independent logic for program verification, we propose — as future work — to enrich the type system so that its logic can accommodate program verification.

2.1.3 Type systems

Few general-purpose program languages have a type system that rivals formal verification in specification power. Sophisticated type systems, such as those of Haskell or Scala, can express interesting invariants about data structures, but they do not subsume the state of the art in specification languages. Thus, we focus on how type systems should evolve in order to compete with the previous approaches, rather than discussing the status quo.

Nonetheless, functional correctness can be specified and verified using the type system, and languages with a rich dependent type system [AMM05], which have been available for quite some time, support this. Moreover, Nanevski et al have recently integrated Hoare-style specifications [NMB08] in such a type system. An even more powerful way to achieve this integration is to enrich the type system so that we can embed a domain-specific language for verification into it. Either way, verification is integrated into the type system so that polymorphism can be applied to specifications.

More generally, reuse must be fostered at all levels by ensuring that polymorphism and other language constructs apply to specifications, types, and values alike. Sheard is exploring *level polymorphism* in Ωmega, a Haskell-like language that supports type-level computation [She07].

To make such a powerful type system practical to use, it must easily be “muted” so that it does not get in the way while prototyping. Later, when correctness becomes more important, its strength is dialled up so that it can express precise specifications. In terms of verification, the type system thus subsumes testing in a dynamically typed language as well as full-blown formal verification. Recent work on optional [BG93], soft [CF91], gradual [AD03, ST07], and hybrid [Fla06] type checking is taking important steps towards this goal.

Another kind of flexibility can be derived from applying the Curry-Howard isomorphism [How69] to automated theorem proving in logic, which corresponds to the type system inferring missing *code* based on its expected type. As an example of this idea, Djinn generates Haskell functions based on their type, and QuickCheck is an automatic testing tool that generates test data based on its type [CH00]. This is another illustration of how types can be put to good use, generating missing implementations during the prototyping phase, or by helping to automate testing.

Furthermore, the type system itself should be extensible, so as to express and encapsulate domain-specific specification "templates" that can be reused with many modules in the same problem domain, so that the cost of developing these type system modules can be amortised. Bracha describes the idea of plugging in different modules into the type checker [Bra04], and Chameleon [WSSR05, SS08] is an important step towards realising this goal. On one hand, Chameleon is a Haskell-like functional language at the level of values; on the other hand, type-level programs are expressed using constraint handling rules [Frü94]. More recently, Schrijvers et al have proposed an approach in which functional programming is used at both levels [SJCS08].

Finally, it is no longer reasonable to expect type checking to be decidable. A diverging type checker simply indicates a bug in the type-level program, and a type-level debugger [SSW03a, SSW03b] should be available to help fixing it.

The remaining challenge is to integrate these ideas into a *scalable type system*, which is discussed in more detail in Section 5.2.1.

2.2 Higher-kinded types

To quote Strachey, who coined the term "parametric polymorphism" [Str67]:

> Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure [. . .]

Technically, parametric polymorphism is simply the type-level equivalent of the core construct of the simply-typed lambda calculus [Chu40]: lambda abstraction. More commonly called a "function", a lambda abstraction is a value that abstracts over another value. Thus, a function is a "polymorphic value". At the level of types, a polymorphic type is a type that is parameterised. Thus, a polymorphic type, or a "type constructor", can be thought of as a function on the level of types: it takes a type as an argument and produces a type as its result.

In the early 1970's, Girard [Gir72] and Reynolds [Rey74] independently formalised these ideas. The serendipity of research on logic and programming languages resulting in the same system – called "system F" or "second-order lambda calculus" – is a striking real-life illustration of the Curry-Howard isomorphism [How69], which relates logic and programming languages. Briefly, it states that a type can be seen as a proposition, and a proof of that proposition corresponds to a term that is classified by that type. Thus, type checking and proof checking are closely related.

The strength of polymorphism can be categorised with respect to its *order*. At the level of values, first-order polymorphism does not allow a function to abstract over a function. Second-order polymorphism relaxes this a little, in the sense that it admits second-order functions, functions that abstract over a first-order function. Higher-order polymorphism allows abstracting over a function that abstracts over a function that abstracts over a function, and so on. At the level of types, the simply-typed lambda calculus is zero-order, as there is no way to abstract over types at all, whereas system F supports second-order polymorphism, and system F_ω supports type constructors that abstract over type constructors that abstract over type constructors, and so on.

Even though a higher-order type system supports abstracting over types of any order, zero-order types must not be confused with higher-order ones. Similarly, on the value level we can abstract over functions and simple values – "zero-order functions" – alike, but a function must still be distinguished from an ordinary value since the former can be applied to a suitable argument, whereas a value cannot. Functions are distinguished from regular values by their type, which is constructed from the function type constructor ($\Rightarrow$). Similarly, monomorphic types are classified by the *kind* $\star$, and type constructors have a kind that is constructed from the type-function kind constructor ($\rightarrow$). For example, `1 : Int`, and the identity function on integers (referred to as `id`) is written[1] `x: Int ⇒ x`. Thus, `id` has type `Int ⇒ Int`, and the application `id(int)` has type `Int`. At the type level, `List` has kind $\star \rightarrow \star$, and as `Int` is classified by $\star$, `List[Int]` again has kind $\star$.

The nesting level of the kind constructor $\rightarrow$ corresponds to the order of the type. First-order polymorphism admits a type of kind $\star \rightarrow \star$, but not of kind $(\star \rightarrow \star) \rightarrow \star$, which is second-order. Types of higher-order kinds are also called *higher-kinded types*.

It is important to point out that a higher-kinded type is not the same as a higher-*ranked* type. The latter type arises from a mixed style

[1] We use Scala syntax, which is explained below.

of polymorphism, where a value abstracts over a type. Consider the (type-)polymorphic version of the identity function in the notation of system F_{ω}^{sub}: $\Lambda\alpha : \star.\, \lambda x : \alpha.\, x$. This function has type $\forall\alpha : \star.\, \alpha \Rightarrow \alpha$, which in turn has kind $\star$. The type $\Lambda\alpha : \star.\, \alpha \Rightarrow \alpha$ looks similar, but it has kind $\star \to \star$, and therefore does not classify any value. The polymorphic identity function has a rank-1 type. A type's rank refers to the nesting depth of the $\forall$ type constructor [KT92].

These systems, along with those that allow types to depend on values, or that provide richer ways of forming types, have been organised in the λ-cube by Barendregt [Bar91].

2.2.1 Functional programming languages

Fragments of system F have served as the basis for many programming languages. The most notable example is Haskell [HJW$^+$92], which has supported higher-kinded types for over 15 years [HHJW07].

Although Haskell has higher-kinded types, it eschews subtyping. Most of the use-cases for subtyping are subsumed by type classes, which handle overloading systematically [WB89]. However, it is not (yet) possible to abstract over class contexts [Hug99, Jon94, Kid07, CJSS07]. In our setting, this corresponds to abstracting over a type that is used as a bound, as discussed in Section 3.6.3.

The interaction between higher-kinded types and subtyping is a well-studied subject [CW85, Car88b, CCH$^+$89, PS97, CG03]. As far as we know, none of these approaches combine bounded type constructors, subkinding, subtyping *and* variance, although all of these features are included in at least one of them.

2.2.2 Object-oriented programming languages

Languages with virtual types or virtual classes, such as gbeta [Ern99], can encode type constructor polymorphism through abstract type members. The idea is to model a type constructor such as `List` as a simple abstract type that has a type member describing the element type. Since Scala has virtual types, `List` could also be defined as a class with an abstract type member instead of as a type-parameterised class:

```
abstract class List { type Elem }
```

Then, a concrete instantiation of `List` could be modelled as a type refinement, as in `List{type Elem = String}`. The crucial point is that in this encoding `List` is a type, not a type constructor. So first-order polymorphism suffices to pass the `List` constructor as a type argument or an abstract type member refinement.

Compared to type constructor polymorphism, this encoding has a serious disadvantage, as it permits the definition of certain accidentally empty type abstractions that cannot be instantiated to concrete values later on. By contrast, type constructor polymorphism has a *kind soundness* property that guarantees that well-kinded type applications never result in nonsensical types. This is discussed in more detail in Chapter 4.

Type constructor polymorphism has recently started to trickle down to object-oriented languages. Cremet and Altherr's work on extending Featherweight Generic Java with higher-kinded types [CA08] partly inspired the design of our syntax. However, since they extend Java, they do not model type members and path-dependent types, definition-site variance, or intersection types. They do provide direct support for anonymous type constructors. Furthermore, although their work demonstrates that type constructor polymorphism can be integrated into Java, they only provide a prototype of a compiler and an interpreter. However, they have developed a mechanised soundness proof and a pencil-and-paper proof of decidability.

Finally, we briefly mention OCaml and C++. C++'s template mechanism is related, but, while templates are very flexible, this comes at a steep price: they can only be type-checked after they have been expanded. Recent work on "concepts" [GJS$^+$06] supports modular type checking by encapsulating the requirements of templates as concepts, so that the library implementation can be checked separately from its clients. A concept is similar to an abstract class in Scala, which declares abstract type and value members. Concepts thus encapsulate requirements, and a generalised notion of constraints is used to express a template's requirements. These constraints resemble Haskell type class contexts and Scala's implicit arguments. Finally, concept maps are provided to aid retro-active extension; they are closely related to type class instances, or implicits values in Scala. Scala's implicits and their relation to Haskell type classes are discussed in Section 3.6.

In OCaml (as in ML), type constructors are first-order [LDG$^+$07, 6.8.1]. Thus, although a type of, e.g., kind $\star \rightarrow \star \rightarrow \star$ is supported, types of kind $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$ cannot be expressed directly. However, ML dialects that support applicative functors, such as OCaml and Moscow ML, can encode type constructor polymorphism in much the same way as languages with virtual types.

2.3 Scala

This section briefly introduces Scala [OAC$^+$06, OSV08]. We assume familiarity with a Java-like language, and focus on what makes Scala different.

2.3.1 Outline of the syntax

A Scala program is essentially a tree of nested definitions. A definition starts with a keyword, followed by its name, a classifier, and the entity to which the given name is bound, if it is a concrete definition. If the root of the tree is the compilation unit, the next level consists of objects (introduced by the keyword **`object`**) and classes (**`class`**, **`trait`**), which in turn contain members. A member may again be a class or an object, a constant value member (**`val`**), a mutable value member (**`var`**), a method (**`def`**), or a type member (**`type`**). Note that a type annotation always follows the name (or, more generally, the expression) that it classifies.

On one hand, Scala's syntax is very regular, with the *keyword/name/-classifier/bound entity*-sequence being its lead motif. Another important aspect of this regularity is nesting, which is virtually unconstrained. On the other hand, syntactic sugar and type inference enable flexibility and succinctness. For example, `buffer += 10` is shorthand for the method call `buffer.+=(10)`, where += is a user-definable identifier. Moreover, the classifier can be elided from most definitions; the compiler can infer it from the right-hand side.

2.3.2 Classes, traits, and objects

In Scala, a class can inherit from another class and one or more traits. A trait is a class that can be composed with other traits using mixin composition. Mixin composition is a restricted form of multiple inheritance, which avoids ambiguities by linearising the graph that results from composing classes that are themselves composites. The main difference between an abstract class and a trait is that the latter can be composed using mixing inheritance. Another difference is that traits cannot define constructors[2]. There are a few more – internal – differences due to the constraints of the underlying platform.

Thus, the reader may safely think of a class as a degenerate kind of trait that is included in Scala for pragmatic reasons. For brevity, most examples use traits, even though an (abstract) class can be used instead if mixin composition is not required.

[2]This limitation may be lifted in future versions.

Classes may contain *type* members. An abstract type member is similar to a type parameter. The main difference between parameters and members is their scope and visibility. A type parameter is syntactically part of the type that it parameterises, whereas a type member – like value members – is encapsulated, and must be selected explicitly. Similarly, type members are inherited, while type parameters are local to their class.

Type parameters are made concrete using type application. Thus, given the definition **`class`** `List[T]`, `List` is a type constructor (or type function), and `List[Int]` is the application of this function to the argument `Int`. Abstract type members are made concrete using *abstract type member refinement*, a special form of mixin composition. Note that `List` is now an abstract class, since it has an abstract member `T`:

```
trait List {
  type T
}
```

This abstract member is made concrete as follows:

```
List{type T=Int}
```

Note that, with the extension that is discussed in Chapter 3, abstract type members may also be parameterised, as in **`type`** `Container[X]`.

The complementary strengths of type parameters and abstract type members are a key ingredient of Scala's recipe for scalable component abstractions [OZ05]. Furthermore, the ability to specify a self type for a class also plays an important role. Essentially, a self type expresses an assumption about the type of the instances of a class. A class with self type `T` can only be instantiated as part of a composition that is a subtype of `T`. By default, a class's self type is the class itself, so in absence of an explicit self type declaration, this condition is trivially met by simply instantiating the class.

As an example, suppose we wish to express that the type checker component (`Typers`) relies on the abstract syntax component (`Syntax`), and support for substitution (`Substitution`). Thus, we define a trait `Typers` and declare its self type to be the intersection type `Syntax` **`with`** `Substitution`, which is a subtype of both `Syntax` and `Substitution`. Note that a class's own type is implicitly included in its self type, so that **`this`** and its alias `self`, which we introduced as part of the self type declaration, actually have the type `Typers` **`with`** `Syntax` **`with`** `Substitution`. The notation is reminiscent of a function type, since a class can be approximated as a function that takes an instance and returns a record of the methods that can be called on that instance.

```
trait Typers { self : Syntax with Substitution ⇒
}
```

Finally, an **`object`** introduces a class with a singleton instance, which can be referred to using the object's name. Consider the following object definition.

```
object Foo
```

This definition can be approximated[3] as follows:

```
class Foo
val Foo: Foo = new Foo
```

2.3.3 Functions

Since Scala is a functional language, functions are first-class values. Thus, like an integer, a function can be written down directly: `x: Int ⇒ x + 1` is the successor function on integers. Furthermore, a function can be passed as an argument to a (higher-order) function or method. Functions and methods are treated similarly in Scala, the main difference is that a method is called on a target object.

The following definition introduces a function `len` that takes a `String` and yields an `Int` by calling `String`'s `length` method on its argument `s`:

```
  val len: String ⇒ Int = s ⇒ s.length
```

In the classifier of the definition, the type `String ⇒ Int`, the arrow ⇒ is a type constructor, whereas it introduces an anonymous function on the right-hand side (where a value is expected). This anonymous function takes an argument `s` of type `String` and returns `s.length`. Thus, the application `len("four")` yields `4`.

Note that the Scala compiler infers [OZZ01] the type of the argument `s`, based on the expected type of the value `len`. The direction of type inference can also be reversed:

```
  val len = (s: String) ⇒ s.length
```

The right-hand side's anonymous function can be abbreviated using syntactic sugar that implicitly introduces functional abstraction. This can be thought of as turning `String`'s `length` method into a function:

[3]A couple of subtleties are lost in translation, namely the initialisation behaviour and the self type.

```
val len: String ⇒ Int = _.length
```

Finally, since Scala is purely object-oriented at its core, a function is represented internally as an object with an `apply` method that is derived straightforwardly from the function. Thus, one more equivalent definition of `len`:

```
object len {
  def apply(s: String): Int = s.length
}
```

Methods may define one or more lists of value parameters, in addition to a list of type parameters. Thus, a method can be seen as a value that abstracts over values and types. For example, `def iterate[T](a: T) (next: T ⇒ T, done: T ⇒ Boolean): List[T]` introduces a method with one type parameter `T`, and two argument lists. Methods with multiple argument lists may be partially applied. For example, for some object `x` on which `iterate` is defined, `x.iterate(0)` corresponds to a higher-order function with type `(Int⇒Int, Int⇒Boolean) ⇒ List[Int]`. Note that the type parameter `T` was inferred to be `Int` from the type of the argument `a`.

Finally, we show how methods can be encoded using only abstract members and mixin composition. This is not an essential aspect of the introduction to Scala, but this pattern will return in Section 2.3.5, where it is used to abstract over types instead of values. A method definition can be encoded by an abstract class with abstract members for the method's arguments, and a concrete member with a fixed name to define the method's result:

```
trait Len {
  val s: String
  val apply: Int = s.length
}
```

Method invocation is now split in two phases: first, the abstract members, which represent the arguments, are made concrete by mixing in an anonymous class, and the resulting class is instantiated. Second, the method's result is computed by selecting the `apply` member.

```
(new Len { val s: String = "four"}).apply
```

2.3.4 Types

To a Java programmer, most Scala types will be familiar, except for intersection types, path-dependent types, type selection, and definition-site variance annotations. Finally, certain types are spelled differently: `Object` becomes `Any`, `Unit` is a regular type that corresponds to the `void` keyword in Java, and `Nothing` is the (uninhabited) subtype of all types, which cannot be expressed in Java.

An intersection type, such as `A` **`with`** `B` can be understood as the type that is a subtype of both `A` and `B`. A type is a subtype of an intersection type if it is a subtype of every constituent of the intersection. The members of an intersection type correspond to the members of the mixin composition of the constituent types, where the linearisation order determines overriding, except that concrete members trump abstract ones.

Scala introduces path-dependent types as a companion to type members. To ensure soundness, an abstract type member may only be selected on a singleton type, which has the shape `p.`**`type`**, where `p` is a path. A path is an expression that is statically known to always refer to the same value. This restriction is imposed syntactically; a path has the shape $\mathtt{x.l}_0\mathtt{.\ldots.l}_N$, where `x` is an immutable variable, and the $\mathtt{l}_i$ refer to immutable value members (introduced by the **`val`** keyword). Furthermore, path equality, and thus equality of singleton types, is decided based on syntactic criteria, so that type checking remains decidable.

A type selection such as `p.`**`type`**`#T` is abbreviated to `p.T` in Scala. If `T` is an abstract type member, the types `p.T` and `q.T` are equal iff the paths `p` and `q` are equal, which corresponds to the requirement that `q` has type `p.`**`type`**.

Another difference with Java is that Scala allows type parameters to specify their variance. Variance enriches the subtyping relation so that subtyping of two types that are constructed from the same type constructor can be derived from their type arguments. For example, **`class`** `List[+T]` introduces the type constructor `List`, whose type parameter is *covariant*. This means that `List[A]` is a subtype of `List[B]` iff `A` is a subtype of `B`. With a *contravariant* type parameter, this is inverted, so that **`class`** `OutputChannel[-T]` entails that `OutputChannel[A]` is a subtype of `OutputChannel[B]` iff `A` is a *supertype* of `B`. Without an explicit variance annotation, type arguments must be equal for the constructed types to be equal.

```
trait Iterable[A, Container[X]] {
  def map[B](f: A ⇒ B): Container[B]
}

trait List[A] extends Iterable[A, List]
```

Listing 2.1: Expressing `Iterable` using parameterisation

```
trait TypeFunction1 { type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container{type A = B}
}

trait List extends Iterable { type Container = List }
```

Listing 2.2: Encoding `Iterable`'s type parameters as members

2.3.5 Encoding higher-kinded types

Before we extended Scala with type constructor polymorphism, it could be encoded to a certain extent. As discussed above, Scala's abstract type members closely correspond to type parameters, and abstract type member refinement can be seen as the object-oriented counterpart of type application. Here, we show the essence of the encoding; it is discussed in more detail in the following chapters.

To make this concrete, Listing 2.1 uses type constructor polymorphism to express the well-known `Iterable` abstraction in Scala. The `Iterable` trait (an abstract class) takes two type parameters: the first one represents the type of the elements, and the second one abstracts over the type constructor of the container. To denote that it abstracts over a type constructor, the `Container` parameter declares a formal type parameter `X`.

Listing 2.2 demonstrates the encoding. Here, `Iterable` abstracts over the type of its elements and the container using abstract members. The `A` type member is inherited from `TypeFunction1`, and the `Container` type constructor parameter is represented as an abstract type member that is bounded to be a `TypeFunction1`. `map`'s result type is expressed by refining `Container`'s abstract type member `A` so that it equals `B`.

Besides the syntactic overhead, the major disadvantage with this encoding is that it may result in the accidental definition of non-sensical types. Type constructor polymorphism solves both problems, as discussed in the following chapter.

2.4 Conclusion

This chapter motivated our focus on the type system from a broader perspective. We briefly surveyed different approaches to producing correct software, and discussed why the type system approach is the most promising one. Essentially, the type system can express rich specifications in an abstract way. Since a lot of work has already gone into supporting reuse and modularity through the type system, giving specifications the same status as types lifts the benefits long enjoyed by software engineers to the realm of software verification. The direct integration of specifications into the language reduces redundancy and eases co-evolution of implementation and specification.

Furthermore, the type system can be made much more flexible than the rigid systems that are the norm in current statically typed languages, without losing the associated benefits. According to the importance of correctness in a certain module, its specification may range from irrelevant and non-existent, over being checked without being enforced, to extremely precise and mandatory. Finally, expressive types can also be leveraged to generate missing pieces of the implementation, so that the investment pays off even beyond encapsulation, enforcement, and the prevention of bugs.

The rest of this thesis describes our concrete contributions to improving an aspect of Scala's type system. The second half of this chapter introduced our extension, type constructor polymorphism, which generalises Scala's support for parametric polymorphism to the higher-order case. We discussed the long history of research on higher-order polymorphism, which has mostly been limited to functional programming languages. Finally, the subset of Scala that is used in the examples of the following chapters was introduced, including the encoding of our extension and the intuition behind its limitations. The following chapters elaborate on the theory and practice of type constructor polymorphism, they explain in detail why the encoding was lacking, and introduce a novel object-oriented calculus that does support a faithful encoding.

Chapter 3

Type Constructor Polymorphism for Scala

This chapter is based on the paper that we presented at OOPSLA 2008 [MPO08a]. The paper was written by the author of this thesis, except for parts of the introduction and of Section 3.6, which were contributed by Martin Odersky. The co-authors provided significant help in improving the structure and the presentation of the paper. We also acknowledge the anonymous reviewers for their helpful feedback. Part of the research – the author's implementation of type constructor polymorphism in the Scala compiler – was performed during a stay at prof. Odersky's lab at EPFL.

3.1 Introduction

First-order parametric polymorphism is now a standard feature of statically typed programming languages. Starting with System F [Gir72, Rey74] and functional programming languages, the constructs have found their way into object-oriented languages such as Java, C$^{\#}$, and many more. In these languages, first-order parametric polymorphism is usually called *generics*. Generics rest on sound theoretical foundations, which were established by Abadi and Cardelli [AC96, AC95], Igarashi et al. [IPW01], and many others; they are well-understood by now.

One standard application area of generics are collections. For instance, the type `List[A]` represents lists of a given element type `A`, which can be chosen freely. In fact, generics can be seen as a generalisation of the type of arrays, which has always been parametric in the type of its elements.

First-order parametric polymorphism has some limitations, however.

Although it allows to abstract over types, which yields *type constructors* such as `List`, these type constructors cannot be abstracted over. For instance, one cannot pass a type constructor as a type argument to another type constructor. Abstractions that require this are quite common, even in object-oriented programming, and this restriction thus leads to unnecessary duplication of code. We provide several examples of such abstractions.

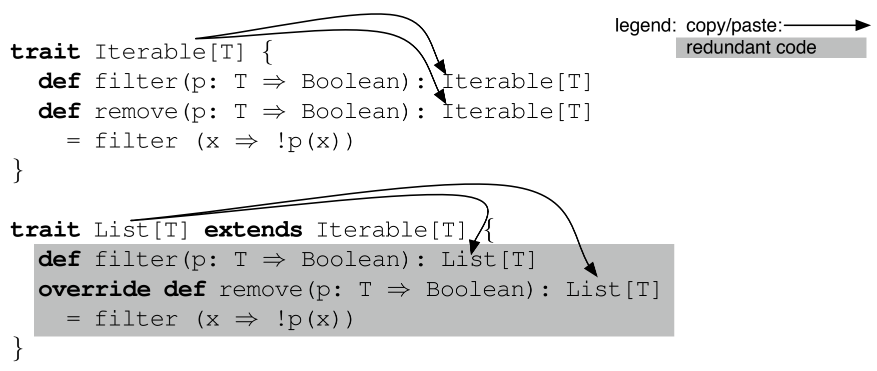
The generalisation of first-order polymorphism to a higher-order system was a natural step in lambda calculus [Gir72, Rey74, BMM90]. This theoretical advance has since been incorporated into functional programming languages. For instance, the Haskell programming language [HJW^{+}92] supports type constructor polymorphism, which is also integrated with its type class concept [Jon95]. This generalisation to types that abstract over types that abstract over types ("higher-kinded types") has many practical applications. For example, comprehensions [Wad92], parser combinators [HM96, LM01], as well as more recent work on embedded Domain Specific Languages (DSL's) [CKcS07, HORM08] critically rely on higher-kinded types.

The same needs – as well as more specific ones – arise in object-oriented programming. LINQ brought direct support for comprehensions to the .NET platform [BMS05, Mei07], Scala [OAC^{+}06] has had a similar feature from the start, and Java 5 introduced a lightweight variation [GJSB05, Sec. 14.14.2]. Parser combinators are also gaining momentum: Bracha uses them as the underlying technology for his Executable Grammars [Bra07], and Scala's distribution includes a library [MPO08b] that implements an embedded DSL for parsing, which allows users to express parsers directly in Scala, in a notation that closely resembles EBNF. Type constructor polymorphism is crucial in defining a common parser interface that is implemented by different back-ends.

Chapter outline We present type constructor polymorphism and motivate its introduction in a real-world object-oriented language by showing how it reduces boilerplate code, that it is safe, and that it can be implemented in a production-quality compiler.

Section 3.2 demonstrates that our extension reduces boilerplate that arises from the use of genericity. We establish intuitions with a simple example, and extend it to a realistic implementation of the comprehensions fragment of `Iterable`.

Section 3.3 present the type and kind system. We discuss the surface syntax in full Scala, and the underlying model of kinds that capture both lower and upper bounds, and variances of types. Based on the ideas es-



```
trait Iterable[T] {
  def filter(p: T ⇒ Boolean): Iterable[T]
  def remove(p: T ⇒ Boolean): Iterable[T]
    = filter (x ⇒ !p(x))
}

trait List[T] extends Iterable[T] {
  def filter(p: T ⇒ Boolean): List[T]
  override def remove(p: T ⇒ Boolean): List[T]
    = filter (x ⇒ !p(x))
}
```

Listing 3.1: Limitations of Genericity

tablished in the theoretical part, Section 3.4 refines `Iterable`, so that it accommodates collections that impose bounds on the type of their elements.

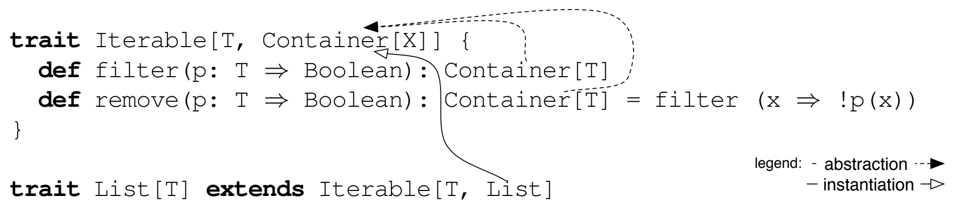
We have validated the practicality of our design by implementing our extension in the Scala compiler, and we report on our experience in Section 3.5. Throughout this chapter, we discuss various interactions of type constructor polymorphism with existing features in Scala. Section 3.6 focusses on the integration with Scala's implicits, which are used to encode Haskell's type classes. Our extension lifts this encoding to type *constructor* classes. Furthermore, due to subtyping, Scala supports abstracting over type class contexts, so that the concept of a bounded monad can be expressed cleanly, which is not possible in (mainstream extensions of) Haskell.

Finally, Martin Odersky's refactoring of Scala's collection library relied heavily on type constructor polymorphism. In Section 3.5.3, we briefly discuss how this experience validates our extension, as well as the opportunities for improvement that emerged.

3.2 Reducing code duplication with type constructor polymorphism

This section illustrates the benefits of generalising genericity to type constructor polymorphism using the well-known `Iterable` abstraction. The first example, which is due to Lex Spoon, illustrates the essence of the problem in the small. Section 3.2.1 extends it to more realistic proportions.

Listing 3.1 shows a Scala implementation of the trait `Iterable[T]`. It contains an abstract method `filter` and a convenience method `remove`. Subclasses should implement `filter` so that it creates a new collection



```
trait Iterable[T, Container[X]] {
  def filter(p: T ⇒ Boolean): Container[T]
  def remove(p: T ⇒ Boolean): Container[T] = filter (x ⇒ !p(x))
}

trait List[T] extends Iterable[T, List]
```

Listing 3.2: Removing Code Duplication

by retaining only the elements of the current collection that satisfy the predicate `p`. This predicate is modelled as a function that takes an element of the collection, which has type `T`, and returns a `Boolean`. As `remove` simply inverts the meaning of the predicate, it is implemented in terms of `filter`.

Naturally, when filtering a list, one expects to again receive a list. Thus, `List` overrides `filter` to refine its result type covariantly. For brevity, `List`'s subclasses, which implement this method, are omitted. For consistency, `remove` should have the same result type, but the only way to achieve this is by overriding it as well. The resulting code duplication is a clear indicator of a limitation of the type system: both methods in `List` are redundant, but the type system is not powerful enough to express them at the required level of abstraction in `Iterable`.

Our solution, depicted in Listing 3.2, is to abstract over the type constructor that represents the container of the result of `filter` and `remove`. The improved `Iterable` now takes two type parameters: the first one, `T`, stands for the type of its elements, and the second one, `Container`, represents the *type constructor* that determines part of the result type of the `filter` and `remove` methods. More specifically, `Container` is a type parameter that itself takes one type parameter. Although the name of this higher-order type parameter (`X`) is not needed here, more sophisticated examples will show the benefit of explicitly naming[1] higher-order type parameters.

Now, to denote that applying `filter` or `remove` to a `List[T]` returns a `List[T]`, `List` simply instantiates `Iterable`'s type parameter to the `List` type constructor.

In this simple example, one could also use a construct like Bruce's `MyType` [BSvG95]. However, this scheme breaks down in more complex

[1]In full Scala '_' may be used as a wild-card name for higher-order type parameters.

```
trait Builder[Container[X], T] {
  def +=(el: T): Unit
  def finalise(): Container[T]
}

trait Iterator[T] {
  def next(): T
  def hasNext: Boolean

  def foreach(op: T ⇒ Unit): Unit
    = while(hasNext) op(next())
}
```

Listing 3.3: `Builder` and `Iterator`

cases, as demonstrated in the next section.

3.2.1 Improving Iterable

In this section we design and implement the abstraction that underlies comprehensions [Wad92]. Type constructor polymorphism plays an essential role in expressing the design constraints, as well as in factoring out boilerplate code without losing type safety. More specifically, we discuss the signature and implementation of `Iterable`'s `map`, `filter`, and `flatMap` methods. The LINQ project brought these to the .NET platform as `Select`, `Where`, and `SelectMany` [Mei06].

Comprehensions provide a simple mechanism for dealing with collections by transforming their elements (`map`, `Select`), retrieving a subcollection (`filter`, `Where`), and collecting the elements from a collection of collections in a single collection (`flatMap`, `SelectMany`).

To achieve this, each of these methods interprets a user-supplied function in a different way in order to derive a new collection from the elements of an existing one: `map` transforms the elements as specified by that function, `filter` interprets the function as a predicate and retains only the elements that satisfy it, and `flatMap` uses the given function to produce a collection of elements for every element in the original collection, and then collects the elements in these collections in the resulting collection.

The only collection-specific operations that are required by a method such as `map`, are iterating over a collection, and producing a new one. Thus, if these operations can be abstracted over, these methods can be implemented in `Iterable` in terms of these abstractions. Listing 3.3 shows

```
trait Buildable[Container[X]] {
  def build[T]: Builder[Container, T]

  def buildWith[T](f: Builder[Container,T] ⇒ Unit)
       : Container[T] = {

    val buff = build[T]

    f(buff)

    buff.finalise()
  }
}
```

Listing 3.4: `Buildable`

the well-known, lightweight, `Iterator` abstraction that encapsulates iterating over a collection, as well as the `Builder` abstraction, which captures how to produce a collection, and thus may be thought of as the dual of `Iterator`.

`Builder` crucially relies on type constructor polymorphism, as it must abstract over the type constructor that represents the collection that it builds. The += method is used to supply the elements in the order in which they should appear in the collection. The collection itself is returned by `finalise`. For example, the `finalise` method of a `Builder[List, Int]` returns a `List[Int]`.

Listing 3.4 shows a minimal `Buildable` with an abstract `build` method, and a convenience method, `buildWith`, that captures the typical use-case for `build`.

By analogy to the proven design that keeps `Iterator` and `Iterable` separated, `Builder` and `Buildable` are modelled as separate abstractions as well. In a full implementation, `Buildable` would contain several more methods, such as `unfold` (the dual of `fold` [GJ98]), which should not clutter the lightweight `Builder` interface.

Note that `Iterable` (Listing 3.5) uses a type constructor member, `Container`, to abstract over the precise type of the container, whereas `Buildable` uses a parameter. Since clients of `Iterable` generally are not concerned with the exact type of the container (except for the regularity that is imposed by our design), it is neatly encapsulated as a type member. `Buildable`'s primary purpose is exactly to create and populate a specific kind of container. Thus, the type of an instance of the `Buildable` class

```
trait Iterable[T] {
  type Container[X] <: Iterable[X]
  def elements: Iterator[T]

  def mapTo[U, C[X]](f: T ⇒ U)
                    (b: Buildable[C]): C[U] = {
    val buff = b.build[U]
    val elems = elements

    while(elems.hasNext) buff += f(elems.next)

    buff.finalise()
  }
  def filterTo[C[X]](p: T ⇒ Boolean)
                    (b: Buildable[C]): C[T] = {
    val elems = elements

    b.buildWith[T]{ buff ⇒
      while(elems.hasNext){
        val el = elems.next
        if(p(el)) buff += el
      }
    }
  }
  def flatMapTo[U,C[X]](f: T⇒Iterable[U])
                       (b: Buildable[C]): C[U] = {
    val buff = b.build[U]
    val elems = elements

    while(elems.hasNext)
      f(elems.next).elements.foreach{ el ⇒ buff += el }

    buff.finalise()
  }

  def map[U](f: T ⇒ U)
            (b: Buildable[Container]): Container[U]
    = mapTo[U, Container](f)(b)
  def filter(p: T ⇒ Boolean)
            (b: Buildable[Container]): Container[T]
    = filterTo[Container](p)(b)
  def flatMap[U](f: T ⇒ Container[U])
                (b: Buildable[Container]): Container[U]
    = flatMapTo[U, Container](f)(b)
}
```

Listing 3.5: `Iterable`

```
class List[T] extends Iterable[T]{
  type Container[X] = List[X]

  def elements: Iterator[T]
    = new Iterator[T] {
        // standard implementation
    }
}
```

Listing 3.6: `List` subclasses `Iterable`

should specify the type of container that it builds. This information is still available with a type member, but it is less manifest.

The `map`/`filter`/`flatMap` methods are implemented in terms of the even more flexible trio `mapTo`/`filterTo`/`flatMapTo`. The generalisation consists of decoupling the original collection from the produced one – they need not be the same, as long as there is a way of building the target collection. Thus, these methods take an extra argument of type `Buildable[C]`. Section 3.6 shows how an orthogonal feature of Scala can be used to relieve callers from supplying this argument explicitly.

For simplicity, the `mapTo` method is implemented as straightforwardly as possible. The `filterTo` method shows how the `buildWith` convenience method can be used.

The result types of `map`, `flatMap`, and their generalisations illustrate why a `MyType`-based solution would not work: whereas the type of **`this`** would be `C[T]`, the result type of these methods is `C[U]`: it is the same type *constructor*, but it is applied to different type *arguments*!

`List` can now easily be implemented as a subclass of `Iterable`, as shown in Listing 3.6. The type constructor of the container is fixed to be `List` itself, and the standard `Iterator` trait is implemented. This implementation does not offer any new insights, so we have omitted it. Listing 3.7 shows the object that implements the `Buildable` interface for `List`.

To illustrate the versatility of the `Buildable` abstraction, Listing 3.8 implements the `Buildable` interface for `Option`. An `Option` corresponds to a list that contains either 0 or 1 elements, and is commonly used in Scala to avoid `null`'s.

```
object ListBuildable extends Buildable[List]{
  def build[T]: Builder[List, T]
    = new ListBuffer[T] with Builder[List, T] {
        // += is inherited from ListBuffer
        // (Scala standard library)
        def finalise(): List[T] = toList
      }
}
```

Listing 3.7: Building a `List`

```
object OptionBuildable extends Buildable[Option] {
  def build[T]: Builder[Option, T]
    = new Builder[Option, T] {
        var res: Option[T] = None()

        def +=(el: T)
          = if(res.isEmpty) res = Some(el)
            else throw new UnsupportedOperation-Exception(">1
    ␣elements")

        def finalise(): Option[T] = res
      }
}
```

Listing 3.8: Building an `Option`

```
val bdays: List[Option[Date]] = List(
  Some(new Date("1981/08/07")), None,
  Some(new Date("1990/04/10")))
def toYrs(bd: Date): Int = // omitted

val ages: List[Int]
 = bdays.flatMapTo[Int, List]{ optBd ⇒
     optBd.map{d ⇒ toYrs(d)}(OptionBuildable)
   }(ListBuildable)

val avgAge = ages.reduceLeft[Int](_ + _) / ages.length
```

Listing 3.9: Example: using `Iterable`

3.2.2 Example: using Iterable

This example demonstrates how to use `map` and `flatMap` to compute the average age of the users of, say, a social networking site. Since users do not have to enter their birthday, the input is a `List[Option[Date]]`. An `Option[Date]` either holds a date or nothing. Listing 3.9 shows how to proceed.

First, a small helper is introduced that computes the current age in years from a date of birth. To collect the known ages, an optional date is transformed into an optional age using `map`. Then, the results are collected into a list using `flatMapTo`. Note the use of the more general `flatMapTo`. With `flatMap`, the inner `map` would have had to convert its result from an `Option` to a `List`, as `flatMap(f)` returns its results in the same kind of container as produced by the function `f` (the inner `map`). Finally, the results are aggregated using `reduceLeft` (not shown here). The full code of the example is available online[2].

Note that the Scala compiler infers most proper types (we added some annotations to aid understanding), but it does not infer type constructor arguments. Thus, type argument lists that contain type constructors, must be supplied manually.

Finally, the only type constructor that arises in the example is the `List` type argument. In fact, in this example, it could have been inferred by the compiler, but this is not always possible in more complex scenarios. This demonstrates that the complexity of type constructor polymorphism, much like with genericity, is concentrated in the internals of the library. The upside is that library designers and implementers have more control over the interfaces of the library, while clients remain blissfully ignorant of the underlying complexity. (As noted earlier, Section 3.6 will show how the arguments of type `Buildable[C]` can be omitted.)

3.2.3 Members versus parameters

The relative merits of abstract members and parameters have been discussed in detail by many others [BOW98, TT99, Ern01]. The Scala philosophy is to embrace both: sometimes parameterisation is the right tool, and at other times, abstract members provide a better solution. Technically, it is possible to safely encode parameters as members, which – surprisingly – was not possible in earlier calculi. Chapter 4 discusses this encoding in detail.

[2] `http://www.cs.kuleuven.be/~adriaan/?q=genericshk`

```
TypeParamClause ::= `[' TypeParam {`,' TypeParam} `]'
TypeParam       ::=  id [TypeParamClause]
                          [`>:' Type] [`<:' Type]

AbstractTpMem   ::=  `type' TypeParam
```

Figure 3.1: Syntax for type declarations (type parameters and abstract type members)

Our examples have used both styles of abstraction. `Buildable`'s main purpose is to build a certain container. Thus, `Container` is a type parameter: a characteristic that is manifest to external clients of `Buildable`, as it is (syntactically) part of the type of its values. In `Iterable` a type member is used, as its external clients are generally only interested in the type of its elements. Syntactically, type members are less visible, as `Iterable[T]` is a valid proper type. To make the type member explicit, one may write `Iterable[T]{`**`type`**` Container[X]=List[X]}`. Alternatively, the `Container` type member can be selected on a singleton type that is a subtype of `Iterable[T]`.

3.3 Of types and kinds

Even though proper types and type constructors are placed on equal footing as far as parametric polymorphism is concerned, one must be careful not to mix them up. Clearly, a type parameter that stands for a proper type, must not be applied to type arguments, whereas a type constructor parameter cannot classify a value until it has been turned into a proper type by supplying the right type arguments.

In this section we give an informal overview of how programmers may introduce higher-kinded type parameters and abstract type members, and sketch the rules that govern their use. We describe the surface syntax that was introduced with the release of Scala 2.5, and the underlying conceptual model of kinds.

3.3.1 Surface syntax for types

Figure 3.1 shows a simplified fragment of the syntax of type parameters and abstract type members, which we collectively call "type declarations". The full syntax, which additionally includes variance annotations, is described

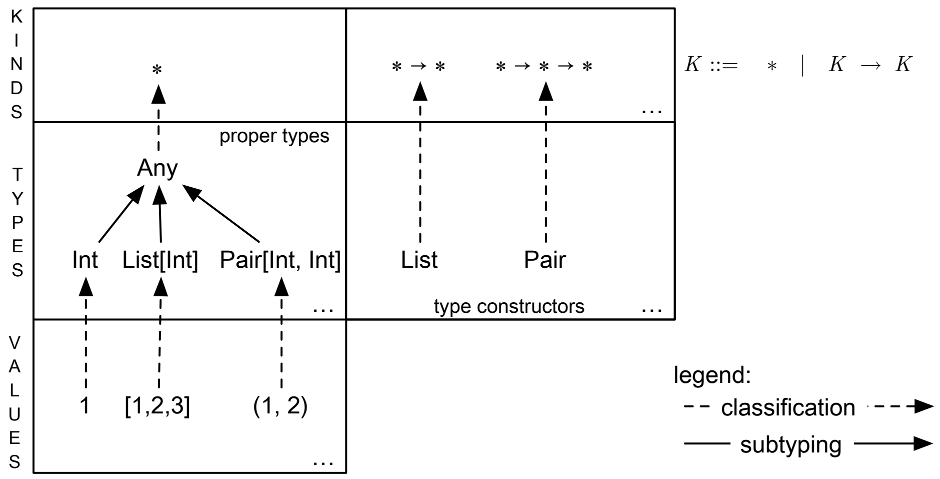


Figure 3.2: Diagram of levels

in the Scala language specification [Ode07]. Syntactically, our extension introduces an optional `TypeParamClause` as part of a type declaration. The scope of the higher-order type parameters that may thus be introduced, extends over the outer type declaration to which they belong.

For example, `Container[X]` is a valid `TypeParam`, which introduces a type constructor parameter that expects one type argument. To illustrate the scoping of higher-order type parameters, `Container[X] <: Iterable [X]` declares a type parameter that, when applied to a type argument `Y` – written as `Container[Y]` – must be a subtype of `Iterable[Y]`.

A more complicated example, `C[X <: Ordered[X]] <: Iterable[X]` introduces a type constructor parameter `C`, with an F-bounded higher-order type parameter `X`, which occurs in its own bound as well as in the bound of the type parameter that it parameterises. Thus, `C` abstracts over a type constructor so that, for any `Y` that is a subtype of `Ordered[Y]`, `C[Y]` is a subtype of `Iterable[Y]`

3.3.2 Kinds

Conceptually, kinds are used to distinguish a type parameter that stands for a proper type, such as `List[Int]`, from a type parameter that abstracts over a type constructor, such as `List`. An initial, simplistic kind system is illustrated in the diagram in Fig. 3.2, and it is refined in the remainder of this section. The figure shows the three levels of classification, where entities in lower levels are classified by entities in the layer immediately

```
Kind ::= '⋆(' Type ',' Type ')'
       | [id '@' ] Kind '->' Kind
```

Figure 3.3: Kinds (not in surface syntax)

above them.

Kinds populate the top layer. The kind $\star$ classifies types that classify values, and the $\rightarrow$ kind constructor is used to construct kinds that classify type constructors. Note that kinds are inferred by the compiler. They cannot appear in Scala's surface syntax.

Nonetheless, Fig. 3.3 introduces syntax for the kinds that classify the types that can be declared as described in the previous section. The first kind, `⋆(T, U)`, classifies proper types (such as type declarations without higher-order type parameters), and tracks their lower (`T`) and upper bounds (`U`). It should be clear that this kind is easily inferred, as type declarations either explicitly specify bounds or receive the minimal lower bound, `Nothing`, and the maximal upper bound, `Any`. Note that intersection types can be used to specify a disjunction of lower bounds, and a conjunction of upper bounds. Since we mostly use upper bounds, we abbreviate `⋆(Nothing, T)` to `⋆(T)`, and `⋆(Nothing, Any)` is written as $\star$.

We refine the kind of type constructors by turning it into a *dependent* function kind, as higher-order type parameters may appear in their own bounds, or in the bounds of their outer type parameter.

In the examples that was introduced above, `Container[X]` introduces a type constructor parameter of kind $\star \rightarrow \star$, and `Container[X]` <: `Iterable[X]` implies the kind `X @ ⋆ → ⋆(Iterable[X])` for `Container`. Finally, the declaration `C[X <: Ordered[X]] <: Iterable[X]` results in `C` receiving the kind `X @ ⋆(Ordered[X]) → ⋆(Iterable[X])`. Again, the syntax for higher-order type parameters provides all the necessary information to infer a (dependent) function kind for type constructor declarations.

Informally, type constructor polymorphism introduces an indirection through the kinding rules in the typing rule for type application, so that it uniformly applies to generic classes, type constructor parameters, and abstract type constructor members. These type constructors, whether concrete or abstract, are assigned function kinds by the kind system. Thus, if `T` has kind `X @ K → K'`, and `U` has kind `K`, in which `X` has been replaced by `U`, a type application `T[U]` has kind `K'`, with the same substitution applied. Multiple type arguments are supported through the obvious generalisation

```
class Iterable[Container[X], T]

trait NumericList[T <: Number]
      extends Iterable[NumericList, T]
```

Listing 3.10: `NumericList`: an illegal subclass of `Iterable`

(taking the necessary care to perform simultaneous substitutions).

3.3.3 Subkinding

Similar to the subtyping relation that is defined on types, subkinding relates kinds. Thus, we overload <: to operate on kinds as well as on types. As the bounds-tracking kind stems from Scala's bounds on type declarations, subkinding for this kind simply follows the rules that were already defined for type member conformance: $\star$`(T, U)` <: $\star$`(T', U')` if `T'` <: `T` and `U` <: `U'`. Intuitively, this amounts to interval inclusion. For the dependent function kind, we transpose subtyping of dependent function types [AC01] to the kind level.

3.3.4 Example: why kinds track bounds

Suppose `Iterable`[3] is subclassed as in Listing 3.10. This program is rejected by the compiler because the type application `Iterable[NumericList, T]` is ill-kinded. The kinding rules classify `NumericList` as a $\star$`(Number)` $\rightarrow \star$, which must be a subkind of the expected kind of `Iterable`'s first type parameter, $\star \rightarrow \star$. Now, $\star$`(Number)` <: $\star$, whereas subkinding for function kinds requires the argument kinds to vary contravariantly.

Intuitively, this type application must be ruled out, because passing `NumericList` as the first type argument to `Iterable` would "forget" that `NumericList` may only contain `Number`'s: `Iterable` is kind-checked under the assumption that its first type argument does not impose any bounds on its higher-order type parameter, and it could thus apply `NumericList` to, say, `String`. The next section elaborates on this.

Fortunately, `Iterable` can be defined so that it can accommodate bounded collections, as shown in Listing 3.11. To achieve this, `Iterable` abstracts over the bound on `Container`'s type parameter. `NumericList` instantiates this bound to `Number`. We refine this example in Section 3.4.

[3]For simplicity, we define `Iterable` using type parameters in this example.

```
class Iterable[Container[X <: Bound], T <: Bound, Bound]

trait NumericList[T <: Number]
        extends Iterable[NumericList, T, Number]
```

Listing 3.11: Safely subclassing `Iterable`

3.3.5 Kind soundness

Analogous to type soundness, which provides guarantees about value-level abstractions, kind soundness ensures that type-level abstractions do not go "wrong".

At the value level, passing, e.g., a `String` to a function that expects an `Integer` *goes wrong* when that function invokes an `Integer`-specific operation on that `String`. Type soundness ensures that application is type-preserving, in the sense that a well-typed application evaluates to a well-typed result.

As a type-level example, consider what happens when a type function that expects a type of kind $\star \rightarrow \star$, is applied to a type of kind $\star$`(Number)` $\rightarrow \star$. This application *goes wrong*, even though the type function itself is well-kinded, if it does something with that type constructor that would be admissible with a type of kind $\star \rightarrow \star$, but not with a type of kind $\star$`(Number)` $\rightarrow \star$, such as applying it to `String`. If the first, erroneous, type application were considered well-kinded, type application would not be kind-preserving, as it would turn a well-kinded type into a nonsensical, ill-kinded, one (such as `NumericList[String]`).

3.3.6 Conclusion

In summary, the presented type system is based on Polarized F^{ω}_{sub} [Ste98], Cardelli's power type, and subkinding. Our bounds-tracking kind $\star$`(L, U)`. corresponds to Cardelli's power type [Car88a]. Subkinding is based on interval inclusion and the transposition of subtyping of dependent function types [AC01] to the level of kinds. The integration of these constructs into an object-oriented type system is the main contribution described in this section. The following chapter presents a more formal account of this type system.

```
trait Builder[Container[X <: B[X]], T <: B[T], B[Y]]
trait Buildable[Container[X <: B[X]], B[Y]] {
  def build[T <: B[T]]: Builder[Container, T, B]
}
trait Iterable[T <: Bound[T], Bound[X]] {
  type Container[X <: Bound[X]] <: Iterable[X, Bound]

  def map[U <: Bound[U]](f: T ⇒ U)
    (b: Buildable[Container, Bound]): Container[U] = ...
}
```

Listing 3.12: Essential changes to extend Iterable with support for (F-)bounds

3.4 Bounded Iterable

As motivated in Section 3.3.4, in order for `Iterable` to model collections that impose an (F-)bound on the type of their elements, it must accommodate this bound from the start.

To allow subclasses of `Iterable` to declare an (F-)bound on the type of their elements, `Iterable` must abstract over this bound. Listing 3.12 generalises the interface of the original `Iterable` from Listing 3.5. The implementation is not affected by this change.

Listing 3.13 illustrates various kinds of subclasses, including `List`, which does not impose a bound on the type of its elements, and thus uses `Any` as its bound (`Any` and `Nothing` are kind-overloaded). Note that `NumericList` can also be derived, by encoding the anonymous type function `X → Number` as `Const1[Number]#Apply`.

Again, the client of the collections API is not exposed to the relative complexity of Listing 3.12. However, without it, a significant fraction of the collection classes could not be unified under the same `Iterable` abstraction. Thus, the clients of the library benefit, as a unified interface for collections, whether they constrain the type of their elements or not, means that they need to learn fewer concepts.

Alternatively, it would be interesting to introduce kind-level abstraction to solve this problem. Tentatively, `Iterable` and `List` could then be expressed as:

```
trait Iterable[T : ElemK, ElemK : Kind]
class List[T] extends Iterable[T, ⋆]
```

This approach is more expressive than simply abstracting over the upper

```
class List[T] extends Iterable[T, Any] {
  type Container[X] = List[X]
}

trait OrderedCollection[T <: Ordered[T]]
        extends Iterable[T, Ordered] {
  type Container[X <: Ordered[X]] <: OrderedCollection[X]
}

trait Const1[T]{type Apply[X]=T}

trait Number
class NumericList[T <: Number]
        extends Iterable[T, Const1[Number]#Apply] {
  type Container[X <: Number] = NumericList[X]
}
```

Listing 3.13: (Bounded) subclasses of Iterable

```
trait Iterable[T >: L <: U, L <: U, U] {
  type Container[X >: L <: U] <: Iterable[X, L, U]
}

class String extends Iterable[Char, Char, Char] {
  type Container[X >: Char <: Char] = String
}
```

Listing 3.14: String as a subclass of Iterable

bound on the element type, as kinds can express lower and upper bounds, and variance. This would become even more appealing in a language that allows the user to define new kinds [She07].

As a more concrete example, consider the ubiquitous `String`: conceptually, it is a collection of characters, so that it is desirable to model it as an `Iterable`. However, this would required extending our bounded `Iterable` even further, so that it also abstracts over lower bounds, as shown in Listing 3.14. Note that we have simplified the bounds `L` and `U` to proper types. More importantly, `String` does not take any type parameters itself, as `Char` is the only type that satisfies the bounds on `Iterable`'s element type. With kind polymorphism, we could have reused the above kind-polymorphic definition of `Iterable`:

```
class String extends Iterable[Char, ⋆(Char, Char)]
```

Another realistic application for kind polymorphism arose during the refactoring of the collections library, where variance must be taken into account (see Section 3.5.3). For now, two variants of `IterableImpl` – the implementation of the `Iterable` interface – are necessary: one for covariant collections, and another for non-variant collections, even though their sole difference is the variance of two related type parameters:

```
trait IterableImplCov[+CC[+B] <: IterableImplCov[CC, B]
                                     with Iterable[B], +A]

trait IterableImplNov[+CC[B]  <: IterableImplNov[CC, B]
                                     with Iterable[B], A]
```

With a kind that captures the variance of a type constructor, one `IterableImpl` could support both variations.

3.5 Full Scala

In this section we discuss our experience with extending the full Scala compiler with type constructor polymorphism. As discussed below, the impact[4] of our extension is mostly restricted to the type checker. Finally, we list the limitations of our implementation, and discuss the interaction with variance. The implementation supports variance annotations on higher-order type parameters, but this has not been integrated in the formalisation yet.

3.5.1 Implementation

Extending the Scala compiler with support for type constructor polymorphism came down to introducing another level of indirection in the well-formedness checks for types.

Once abstract types could be parameterised (a simple extension to the parser and the abstract syntax trees), the check that type parameters must always be proper types had to be relaxed. Instead, a more sophisticated mechanism tracks the kinds that are inferred for these abstract types. Type application then checks two things: the type that is used as a type constructor must indeed have a function kind, and the kinds of the supplied arguments must conform to the expected kinds. Additionally, one must ensure that type constructors do not occur as the type of a value.

[4]The initial patch to the compiler can be viewed at `http://lampsvn.epfl.ch/trac/scala/changeset/10642`

Since Scala uses type erasure in the back-end, the extent of the changes is limited to the type checker. Clearly, our extension thus does not have any impact on the run-time characteristics of a program. Ironically, as type erasure is at the root of other limitations in Scala, it was an important benefit in implementing type constructor polymorphism.

Similar extensions in languages that target the .NET platform face a tougher challenge, as the virtual machine has a richer notion of types and thus enforces stricter invariants. Unfortunately, the model of types does not include higher-kinded types. Thus, to ensure full interoperability with genericity in other languages on this platform, compilers for languages with type constructor polymorphism must resort to partial erasure, as well as code specialisation in order to construct the necessary representations of types that result from abstract type constructors being applied to arguments.

Type constructor inference

Scala uses local type inference [PT00, OZZ01] to ease the burden of type annotation. In many places, the compiler can reconstruct omitted type information from the types of expressions and the bounds on abstract types. Essentially, unknown types are replaced by type variables that record the constraints that must be met in order for type checking to succeed. The solution of these constraints determines the missing type.

To extend this algorithm to deal with type constructors, type variables must now take parameters. More concretely, when inferring a concrete type constructor for a missing type constructor argument, the corresponding type parameter is replaced by a type variable. Since this type parameter has type parameters, the type variable must as well. Thus, the checks that record the constraints associated with the type variable must be generalised to deal with parameterised type variables.

At the time of writing, the author's implementation of type constructor inference had not yet been incorporated into the latest stable release (version 2.7.3), but it is expected to be available in version 2.8 of the official Scala compiler.

Limitations

Syntactically, there are a few limitations that we would like to lift in upcoming versions. As it stands, we do not directly support partial type application and currying, or anonymous type functions. However, these features can be encoded, as illustrated in Section 3.4.

3.5.2 Variance

Another facet of the interaction between subtyping and type constructors is seen in Scala's support for definition-site variance annotations [EKRY06]. Variance annotations provide the information required to decide subtyping of types that result from applying the same type constructor to different types.

As the classical example, consider the definition of the class of immutable lists, **`class`** `List[+T]`. The `+` before `List`'s type parameter denotes that `List[T]` is a subtype of `List[U]` if `T` is a subtype of `U`. We say that `+` introduces a covariant type parameter, `-` denotes contravariance (the subtyping relation between the type arguments is the inverse of the resulting relation between the constructed types), and the lack of an annotation means that these type arguments must be identical.

Variance annotations pose the same kind of challenge to the model of kinds as did bounded type parameters: kinds must encompass them as they represent information that should not be glossed over when passing around type constructors. The same strategy as for including bounds into $\star$ can be applied here, except that variance is a property of type *constructors*, so it should be tracked in $\rightarrow$, by distinguishing $\xrightarrow{+}$ and $\xrightarrow{-}$ [Ste98].

Without going in too much detail, we illustrate the need for variance annotations on higher-order type parameters and how they influence kind conformance.

Listing 3.15 defines a perfectly valid `Seq` abstraction, albeit with a contrived `lift` method. Because `Seq` declares `C`'s type parameter `X` to be covariant, it may use its covariant type parameter `A` as an argument for `C`, so that `C[A]` <: `C[B]` when `A` <: `B`.

`Seq` declares the type of its **`this`** variable to be `C[A]` (`self: C[A]` ⇒ declares `self` as an alias for **`this`**, and gives it an explicit type). Thus, the `lift` method may return **`this`**, as its type can be subsumed to `C[B]`.

Suppose that a type constructor that is invariant in its first type parameter could be passed as the argument for a type constructor parameter that assumes its first type parameter to be covariant. This would foil the type system's first-order variance checks: `Seq`'s definition would be invalid if `C` were invariant in its first type parameter.

The remainder of Listing 3.15 sets up a concrete example that would result in a run-time error if the type application `Seq[A, Cell]` were not ruled out statically.

More generally, a type constructor parameter that does not declare any variance for its parameters does not impose any restrictions on the variance

```
trait Seq[+A, C[+X]] { self: C[A] ⇒
  def lift[B >: A]: C[B] = this
}

class Cell[A] extends Seq[A, Cell] { // only static error
  private var cell: A = _
  def set(x: A) = cell = x
  def get: A = cell
}

class Top
class Ext extends Top {
  def bar() = println("bar")
}

val exts: Cell[Ext] = new Cell[Ext]
val tops: Cell[Top] = exts.lift[Top]
tops.set(new Top)
exts.get.bar()  // method not found error,
                // if the above static error is ignored
```

Listing 3.15: Example of unsoundness if higher-order variance annotations are not enforced.

of the parameters of its type argument. However, when either covariance or contravariance is assumed, the corresponding parameters of the type argument must have the same variance.

3.5.3 Refactoring the collections library

Recently, Odersky refactored the Scala collections library leveraging type constructor polymorphism. More concretely, the core of the library is now structured as a single hierarchy of classes that implement the core functionality for abstract collections. These higher-level classes are parametric in the type constructor of the collection that they represent. The abstract classes are specialised into two symmetric hierarchies of concrete mutable and immutable collections, instantiating the type constructor parameter to the concrete type constructor of the collection.

Thus, the functionality that is unrelated to a collection's mutability has been pulled up to the abstract hierarchy, whose root models an abstract collection (`IterableTemplate`), and includes more specific collections, such as ordered collections (`OrderedIterableTemplate`), indexable collections (`SequenceTemplate`), O(1)-indexable collections (`VectorTemplate`), sets (`SetTemplate`), and maps (`MapTemplate`). As a result, the classes in the mutable and immutable hierarchies – lists, lazy lists, vectors, sets and maps based on hashing, . . . – simply inherit the corresponding collection template, instantiate the type constructor to their concrete type constructor, and implement their truly specific functionality.

This has greatly reduced code duplication, so that, filtering a `Vector`, for example, now yields a `Vector`, even though `filter` was implemented at the top of the hierarchy in `IterableTemplate`. The `Builder` pattern plays an important role in pulling up this redundant code, as explained earlier.

Unfortunately, the change in lines of code due to this refactoring cannot easily be correlated with our extension's potential for reducing code duplication, as the refactoring involved much more than simply reshaping the existing library to leverage type constructor polymorphism. Old functionality was removed, new methods were added and existing ones were implemented more elegantly or more efficiently, without necessarily relying on type constructor polymorphism.

Besides confirming the utility of type constructor polymorphism, the refactoring also challenged its expressiveness. More specifically, the core implementation classes, such as `IterableTemplate`, are required in a covariant (for immutable subclasses) and a nonvariant (for mutable subclasses) variety, even though the nonvariant version can be derived from the covariant one by simply dropping the appropriate '+' variance annotations. Currently, this redundancy is avoided by only implementing the nonvariant version, which is casted to the covariant version using an annotation to override the compiler's kind checks.

Again, this motivates kind polymorphism as a more powerful generalisation of type constructor polymorphism. As discussed in Section 3.4, kinds can be used to abstract more succinctly over the higher-order bounds of an abstract type constructor. Bounds behave similarly to variance annotations with respect to conformance, and both are captured by kinds, so that kind polymorphism could solve these limitations of type constructor polymorphism, which were both encountered in realistic applications.

3.6 Leveraging Scala's implicits

In this section we discuss how the introduction of type constructor polymorphism has made Scala's support for implicit arguments more powerful. Implicits have been implemented in Scala since version 1.4. They are the minimal extension to an object-oriented language so that Haskell's type classes [WB89] can be encoded [Ode06].

We first show how to improve the example from Section 3.2 using implicits, so that clients of `Iterable` no longer need to supply the correct instance of `Buildable[C]`. Since there generally is only one instance of `Buildable[C]` for a particular type constructor `C`, it becomes quite tedious to supply it as an argument whenever calling one of `Iterable`'s methods that requires it.

Fortunately, Scala's implicits can be used to shift this burden to the compiler. It suffices to add the **`implicit`** keyword to the parameter list that contains the `b: Buildable[C]` parameter, and to the `XXXIsBuildable` objects. With this change, which is sketched in Listing 3.16, callers (such as in the example of Listing 3.9) typically do not need to supply this argument.

In the rest of this section we explain this feature in order to illustrate the interaction with type constructor polymorphism. With the introduction of type constructor polymorphism, our encoding of type classes is extended to constructor classes, such as `Monad`, as discussed in Section 3.6.3. Moreover, our encoding exceeds the original because we integrate type constructor polymorphism with subtyping, so that we can abstract over bounds. This would correspond to abstracting over type class contexts, which is not supported in Haskell [Hug99, Jon94, Kid07, CJSS07]. Section 3.6.3 discusses this in more detail.

```
trait Iterable[T] {
  def map[U](f: T ⇒ U)
            (implicit b: Buildable[Container]): Container[U]
    = mapTo[U, Container](f)
    // no need to pass b explicitly
    // similar for other methods
}

implicit object ListBuildable extends Buildable[List]
   {...}
implicit object OptionBuildable extends Buildable[Option]
   {...}

// client code (see previous example):
val ages: List[Int] = bdays.flatMapTo[Int, List]{
                        maybeDate ⇒ maybeDate.map{toYrs(_)}
                       }
```

Listing 3.16: Snippet: leveraging implicits in `Iterable`

3.6.1 Introduction to implicits

The principal idea behind implicit parameters is that arguments for them can be left out from a method call. If the arguments corresponding to an implicit parameter section are missing, they are inferred by the compiler.

Listing 3.17 introduces implicits by way of a simple example. It defines an abstract class of monoids and two concrete implementations, `StringMonoid` and `IntMonoid`. The two implementations are marked with an **`implicit`** modifier.

Listing 3.18 implements a `sum` method, which works over arbitrary monoids. `sum`'s second parameter is marked **`implicit`**. Note that `sum`'s recursive call does not need to pass along the `m` implicit argument.

The actual arguments that are eligible to be passed to an implicit parameter include all identifiers that are marked **`implicit`**, and that can be accessed at the point of the method call without a prefix. For instance, the scope of the `Monoids` object can be opened up using an import statement, such as **`import`** `Monoids._` This makes the two implicit definitions of `stringMonoid` and `intMonoid` eligible to be passed as implicit arguments, so that one can write:

```
sum(List("a", "bc", "def"))
sum(List(1, 2, 3))
```

```
abstract class Monoid[T] {
  def add(x: T, y: T): T
  def unit: T
}

object Monoids {
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid
                  extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

Listing 3.17: Using implicits to model monoids

```
def sum[T](xs: List[T])(implicit m: Monoid[T]): T
  = if(xs.isEmpty) m.unit else m.add(xs.head, sum(xs.tail))
```

Listing 3.18: Summing lists over arbitrary monoids

```
class  Ord a  where
  (<=) :: a → a → Bool

instance Ord Date where
  (<=)     = ...

max     :: Ord a ⇒ a → a → a
max x y = if x <= y then y else x
```

Listing 3.19: Using type classes to overload <= in Haskell

These applications of `sum` are equivalent to the following two applications, where the formerly implicit argument is now given explicitly.

```
sum(List("a", "bc", "def"))(stringMonoid)
sum(List(1, 2, 3))(intMonoid)
```

If there are several eligible arguments that match an implicit parameter's type, a most specific one will be chosen using the standard rules of Scala's static overloading resolution. If there is no unique most specific eligible implicit definition, the call is ambiguous and will result in a static error.

3.6.2 Encoding Haskell's type classes with implicits

Haskell's type classes have grown from a simple mechanism that deals with overloading [WB89], to an important tool in dealing with the challenges of modern software engineering. Its success has prompted others to explore similar features in Java [WLT07].

An example in Haskell

Listing 3.19 defines a simplified version of the well-known `Ord` type class. This definition says that if a type `a` is in the `Ord` *type class*, the function `<=` with type `a → a → Bool` is available.

The *instance declaration* **`instance`** `Ord Date` gives a concrete implementation of the `<=` operation on `Date`'s and thus adds `Date` as an *instance* to the `Ord` type class. To constrain an abstract type to instances of a type class, *contexts* are employed. For example, `max`'s signature constrains `a` to be an instance of `Ord` using the context `Ord a`, which is separated from the function's type by a ⇒.

Conceptually, a context that constrains a type `a`, is translated into an extra parameter that supplies the implementations of the type class's meth-

```
trait Ord[T] {
  def <= (other: T): Boolean
}

import java.util.Date

implicit def dateAsOrd(self: Date)
  = new Ord[Date] {
    def <= (other: Date) = self.equals(other)
                        || self.before(other)
  }

def max[T <% Ord[T]](x: T, y: T): T  = if(x <= y) y else x
```

Listing 3.20: Encoding type classes using Scala's implicits

ods, packaged in a so-called "method dictionary". An instance declaration specifies the contents of the method dictionary for this particular type.

Encoding the example in Scala

It is natural to turn a type class into a class, as shown in Listing 3.20. Thus, an instance of that class corresponds to a method dictionary, as it supplies the actual implementations of the methods declared in the class.

The instance declaration **`instance`** `Ord Date` is translated into an implicit method that converts a `Date` into an `Ord[Date]`. The method dictionary for the instance at `Date` corresponds to an object of type `Ord[Date]`.

Because of Scala's object-oriented nature, the creation of method dictionaries is driven by member selection. Whereas the Haskell compiler selects the right method dictionary fully automatically, this process is triggered by calling missing methods on objects of a type that is an instance (in the Haskell sense) of a type class that does provide this method. When a type class method, such as <=, is selected on a type `T` that does not define that method, the compiler searches an implicit value that converts a value of type `T` into a value that does support this method. In this case, the implicit method `dateAsOrd` is selected when `T` equals `Date`.

Note that Scala's scoping rules for implicits differ from Haskell's. Briefly, the search for an implicit is performed locally in the scope of the method call that triggered it, whereas this is a global process in Haskell.

Contexts are another trigger for selecting method dictionaries. The context `Ord a` of the `max` method becomes a view bound `T <% Ord[T]`, which

```
def max[T](x: T, y: T)
          (implicit conv: T ⇒ Ord[T]): T
  = if(x <= y) y else x
```

Listing 3.21: Desugaring view bounds

```
def max[T](x: T, y: T)(c: T ⇒ Ord[T]): T
  = if(c(x).<=(y)) y else x
```

Listing 3.22: Making implicits explicit

is syntactic sugar for an implicit parameter that converts the bounded type to its view bound. Thus, when the `max` method is called, the compiler must find the appropriate implicit conversion. Listing 3.21 removes this syntactic sugar, and Listing 3.22 goes even further and makes the implicits explicit. Clients would then have to supply the implicit conversion explicitly: `max(dateA, dateB)(dateAsOrd)`.

Conditional implicits

By defining implicit methods that themselves take implicit parameters, Haskell's conditional instance declarations can be encoded:

```
instance Ord a ⇒ Ord (List a) where
  (<=)     = ...
```

This is encoded in Scala as:

```
implicit def listAsOrd[T](self: List[T])(implicit v: T ⇒ Ord
   [T]) =
  new Ord[List[T]] {
    def <= (other: List[T]) = // compare elements in self and
     other
  }
```

Thus, two lists with elements of type `T` can be compared as long as their elements are comparable.

Type classes and implicits both provide ad-hoc polymorphism. Like parametric polymorphism, this allows methods or classes to be applicable to arbitrary types. However, parametric polymorphism implies that a method or a class is truly indifferent to the actual argument of its type parameter, whereas ad-hoc polymorphism maintains this illusion by selecting different methods or classes for different actual type arguments.

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b

data (Ord a) ⇒ Set a = ...

instance Monad Set where
 -- (>>=) :: Set a → (a → Set b) → Set b
```

Listing 3.23: `Set` cannot be made into a `Monad` in Haskell

This ad-hoc nature of type classes and implicits can be seen as a retroactive extension mechanism. In OOP, virtual classes [OH92, Ern99] have been proposed as an alternative that is better suited for retroactive extension. However, ad-hoc polymorphism also allows types to drive the selection of functionality as demonstrated by the selection of (implicit) instances of `Buildable[C]` in our `Iterable` example. `Buildable` clearly could not be truly polymorphic in its parameter, as that would imply that there could be one `Buildable` that knew how to supply a strategy for building any type of container. Java's static overloading mechanism is another example of ad-hoc polymorphism.

3.6.3 Exceeding type classes

Haskell's `Monad` abstraction [Wad95] does not apply to type constructors with a constrained type parameter, such as `Set`, as explained below. Resolving this issue in Haskell is an active research topic [CJSS07, CKJM05, Hug99]. As discussed in Section 3.4, it is quite possible to extend `Iterable` to deal with bounded element types. From the perspective of the type system, there is little difference between Scala's `Iterable` and Haskell's `Monad`, besides *spelling*.

Listing 3.23 illustrates that the `Monad` abstraction does not accommodate constraints on the type parameter of the `m` type constructor that it abstracts over. Since `Set` is a type constructor that constrains its type parameter, it is not a valid argument for `Monad`'s `m` type parameter: `m a` is allowed for any type `a`, whereas `Set a` is only allowed if `a` is an instance of the `Ord` type class. Thus, passing `Set` as `m` might violate this constraint.

For reference, Listing 3.24 shows a direct encoding of the `Monad` type class. Note that the `>>=` method corresponds to `flatMap` in `Iterable`, whose type member `Container` has been replaced by `Monad`'s type parameter `M`. To solve the problem in Scala, we generalise `Monad` to `BoundedMonad` in Listing 3.25 to deal with bounded type constructors. Finally, the encod-

```
trait Monad[A, M[X]] {
  def >>= [B](f: A ⇒ M[B]): M[B]
}
```

Listing 3.24: `Monad` in Scala

```
trait BoundedMonad[A <: Bound[A], M[X <: Bound[X]], Bound[X]] {
  def >>= [B <: Bound[B]](f: A ⇒ M[B]): M[B]
}

trait Set[T <: Ord[T]]

implicit def SetIsBoundedMonad[T <: Ord[T]](
  s: Set[T]): BoundedMonad[T, Set, Ord] = ...
```

Listing 3.25: `Set` as a `BoundedMonad` in Scala

ing from Section 3.6.2 is used to turn a `Set` into a `BoundedMonad`.

3.6.4 Aside: implicit arguments versus subtype bounds

As type checking drives the selection of implicit values, the static and dynamic realms meet in Scala's support for implicit arguments. Even though subtype bounds are a strictly static notion, they can sensibly be compared to implicits. We explore their differences and similarities, and illustrate how implicits can be used to approximate, and – in a way – *surpass* subtype bounds.

Conceptually, regardless of whether they abstract over types or values, the parameters of a method constrain how this method may be called. In Pierce's exposition of system F [Pie02, Fig. 23-1, p. 343], for example, this is made explicit in the operational semantics, which includes similar evaluation rules for value application (E-App1, E-App2) and type application (E-TApp). In practice, type application and value application are normally executed during distinct phases. While type checking, type applications are performed, and value applications happen at run time.

However, Scala's implicit arguments blur this distinction. More specifically, Listing 3.26 defines the method `foo` using a subtype bound on its type parameter, as well as using an implicit argument that requires a witness to the subtype constraint to be supplied at run time. In this context, that witness is statically known to be the implicit value `subtype_witness`, as the implicit method gives rise to a (type-)polymorphic function with type $\forall T <: U, U.T \Rightarrow U$. Thus, any implicit argument of type `A ⇒ B` can

```
class Bar { def baz = 42 }

def foo[T <: Bar](x: T) = x.baz

// equivalent definition (modulo run-time behaviour, ...):
def foo[T](x: T)(implicit c: T ⇒ Bar) = x.baz

implicit def subtype_witness[T <: U, U](x: T): U = x
```

Listing 3.26: Comparing bounds and implicits

```
trait Iterable[T] {
// ...
  def flatten[U](implicit b: Buildable[Container], c: T ⇒
    Iterable[U]): Container[U]
    = flatMapTo[U, Container](c)(b)
  }
}

val x: Iterable[Iterable[Int]] = List(List(1), List(2, 3))
val y: Iterable[Int] = x.flatten // = List(1, 2, 3)
```

Listing 3.27: `Iterable` with `flatten`

be satisfied by `subtype_witness[A, B]` if `A <: B`.

From the caller's perspective, there is no syntactic difference between the two versions of `foo`. Both can be invoked as, e.g., `foo(new Bar)`. Furthermore, note that no casts are involved. Finally, the compiler could determine that inlining `subtype_witness` in `foo` does not change the method's run-time behaviour in any way, so that `foo`'s implicit argument list can be eliminated altogether when `subtype_witness` is selected as the implicit value.

More interestingly, we use this scheme in Fig. 3.27 to emulate generalised constraints [EKRY06] in Scala. A well-known example is the `flatten` operation, which cannot properly be defined as a member of `Iterable[T]` without generalised constraints. Listing 3.5 shows how to add it to the implementation of `Iterable` that was discussed in Section 3.2.1. The implicit argument `c` allows the `flatten` method to constrain `Iterable`'s type parameter `T` to be a subtype of `Iterable[U]`. Normally, a method can only bound its own type parameters.

Note that the `b: Buildable[Container]` argument can also be interpreted as a generalised constraint, namely on `Iterable`'s type constructor member `Container`. However, here, the witness to the constraint has computational content: it allows us to build a collection of type `Container[T]`, for any `T`. Thus, it would not have made sense to use a subtype bound on `Container` in the first place, as a bound on an abstract type only serves to provide more information about values of that type. Since `Container` is a type constructor, it does not classify any values, and even though the type `Container[T]` does have values, which the bound thus constrains, we do not have any such value – we are trying to create one!

3.7 Conclusion

Genericity is a proven technique to reduce code duplication in object-oriented libraries while making them easier to use by clients. The prime example is a collections library, where clients no longer need to cast the elements they retrieve from a generic collection.

Unfortunately, though genericity is extremely useful, the first-order variant is self-defeating in the sense that abstracting over proper types gives rise to type constructors, which cannot be abstracted over. Thus, by using genericity to reduce code duplication, other kinds of boilerplate arise. Type constructor polymorphism allows to further eliminate these redundancies, as it generalises genericity to type constructors.

As with genericity, most use cases for type constructor polymorphism arise in library design and implementation, where it provides more control over the interfaces that are exposed to clients, while reducing code duplication. Moreover, clients are not exposed to the complexity that is inherent to these advanced abstraction mechanisms. In fact, clients *benefit* from the more precise interfaces that can be expressed with type constructor polymorphism, just like genericity reduced the number of casts that clients of a collections library had to write.

We implemented type constructor polymorphism in Scala 2.5. The essence of our solution carries over easily to Java, see Altherr et al. for a proposal [CA08].

Finally, we have only reported on one of several applications that we have experimented with. Embedded domain specific languages (DSL's) [CKcS07, HORM08] are another promising application area of type constructor polymorphism. We are currently applying these ideas to our parser combinator library, a DSL for writing EBNF grammars in Scala [MPO08b]. In part in collaboration with the author of this thesis, Hofer, Ostermann et al. are investigating similar applications [HORM08], which critically rely on type constructor polymorphism.

Chapter 4

Scalina: the Essence of Abstraction in Scala

This chapter is based on the paper that we presented at FOOL 2008 [MPO08c]. The paper was written by the author of this thesis. We acknowledge insightful comments, especially from the co-authors and Erik Ernst, as well as from the anonymous reviewers.

4.1 Introduction

Scalina is a purely object-oriented calculus that provides the formal underpinning for our extension of Scala with type constructor polymorphism, which was discussed in the previous chapter. Scalina introduces a number of novelties with respect to earlier object-oriented calculi [IPW01, OCRZ03, CGLO06]. The most notable improvement over the νObj calculus is that kind checking ensures type applications never "go wrong". We dub this property *kind soundness.*

Traditionally, most object-oriented languages and the underlying formalisms use a mix of FP-style and OO-style abstraction. The former style is based on lambda abstraction and function application, and OO-style abstractions are built using abstract members and composition (via subclassing or mixin composition).

Java, for example, uses functional abstraction for methods and classes, which may be parametric in types and values. Of course, Java also supports OO-style abstraction: a class with an abstract method abstracts from the implementation of that method. A subclass is expected to provide the concrete implementation.

```
trait Iterable[A, Container[X]] {
  def map[B](f: A ⇒ B): Container[B]
}

trait List[A] extends Iterable[A, List]
```

Listing 4.1: Expressing `Iterable` using parameterisation

Like νObj, Scalina is a purely object-oriented calculus: there are no constructs for parameterisation. Yet, as we will demonstrate, Scalina is able to express the same abstractions as, for example, system F_{ω}^{sub} [Car88a, PS97, CG03], with the same safety guarantees.

Chapter outline The rest of this section elaborates on the problem statement and gives some initial insight into our solution. Then, we get our feet wet with Scalina's syntax and intuitions in Section 4.2, before delving deeper in the levels of terms (Section 4.3) and types (Section 4.4). The latter two sections discuss computation and classification at the respective levels. We briefly motivate Scalina's design and position it in the design space in Section 4.5. In Section 4.6 we make the relation between Scalina and system F_{ω}^{sub} more precise. We sketch the meta-theory in Section 4.7. Finally, we briefly discuss related work (Section 4.8) before concluding in Section 4.9.

4.1.1 Kind soundness

Scala supports two styles of abstraction: the functional style uses parameterisation, whereas abstract members represent the object-oriented way. It is natural to ask whether one style can be used exclusively. We focus on how the object-oriented style can encode the functional one, since Scala is object-oriented at its core.

Scala's abstract type members closely correspond to type parameters, and abstract type member refinement can be seen as the object-oriented counterpart of type application. Abstract type member refinement is a restricted form of mixin composition that can be used to override abstract type members with concrete ones. However, it turns out that this encoding does not preserve the safety properties that are ensured by parameterisation.

To make this concrete, Listing 4.1 uses parameterisation to express the well-known `Iterable` abstraction in Scala. The `Iterable` trait (an

```
trait TypeFunction1 { type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container{type A = B}
}

trait List extends Iterable { type Container = List }
```

Listing 4.2: Encoding `Iterable`'s type parameters as members

```
trait NumericList[A <: Number]
    extends Iterable[A, NumericList]
```

Listing 4.3: `NumericList`: an illegal subclass of `Iterable`

abstract class) takes two type parameters: the first one represents the type of the elements, and the second one abstracts over the type constructor of the container. To denote that it abstracts over a type constructor, the `Container` parameter declares a formal type parameter `X`.

Listing 4.2 demonstrates the object-oriented style. Here, `Iterable` abstracts over the type of its elements and the container using abstract members. The `A` type member is inherited from `TypeFunction1`, and the `Container` type constructor parameter is represented as an abstract type member that is bounded to be a `TypeFunction1`. `map`'s result type is expressed by refining `Container`'s abstract type member `A` so that it equals `B`.

So far, the encoding remained faithful to the original. However, a discrepancy emerges when we encode an erroneous program. The type application `Iterable[A, NumericList]` in Listing 4.3 is not allowed by the compiler, whereas we will see its encoding is accepted without warning. If it were not ruled out, `map`'s result type could apply any type `B` to `NumericList`, while it accepts only subtypes of `Number`. By ruling out `Iterable[A, NumericList]`, the compiler prevents this error from ever happening.

Unfortunately, the encoding does not preserve this property, which we call "kind soundness". This is illustrated by Listing 4.4, which is considered a valid Scala program. The compiler silently accepts this program, even though we could never complete its implementation (at some point we will have to instantiate a `NumericList` for an arbitrary type of elements, and the compiler will catch our mistake). To relate this to type soundness, the

```
trait NumericList extends Iterable {
  type A <: Number
  type Container = NumericList // Incorrect, but no error
    reported!
}
```

Listing 4.4: The encoding of `NumericList` eludes the type checker

value-level equivalent of this oversight would be to allow passing a function of type, e.g., `Number ⇒ Any` to a function that expects a `Any ⇒ Any`.

Note that this indulgence does not imply *type* unsoundness, as these erroneous types cannot be instantiated. Nonetheless, we regard it as a shortcoming of the compiler that these vacuous intersection types are allowed to slip by unnoticed. Even though they are prevented from being instantiated, they could be unmasked earlier.

To motivate this desire for early detection of these inconsistencies, consider the analogy with abstract classes. Suppose classes would be allowed to be abstract implicitly, so that accidental abstract classes would not be discovered until a client attempts to instantiate them. However, this situation is considered undesirable by most languages, so that an abstract class must be marked as such explicitly. This eliminates the possibility that the programmer simply forgot to implement a method.

Not detecting erroneous type applications, which manifest themselves as intersection types that unexpectedly do not have any instances, has the same effect as allowing any class to be abstract implicitly: the error is detected eventually, but it could have been signalled earlier. Even though other uses of intersection types might sensibly result in empty types, we do not consider this to be one of them.

This kind unsoundness has its roots in the νObj calculus [OCRZ03], which allows abstract type members to be refined *covariantly*, thus `NumericList <: TypeFunction1`, so that the encoding of the erroneous type application results in a valid program.

We recover early error detection in Scalina by differentiating covariant and contravariant members, instead of assuming they all behave covariantly. This distinction corresponds to the fact that some members abstract over input, whereas others represent the output of the abstraction. Input members should behave contravariantly, like the types of function arguments, whereas covariance is required for output members, which correspond to a function's result type. With this distinction, a purely object-oriented calculus can encode functional-style abstraction with the same safety guar-

```
trait TypeFunction1 { deferred type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container◁{type A = B}
}

trait List extends Iterable { type Container = List }

trait NumericList extends Iterable {
  deferred type A <: Number // error: covariant change not
    allowed
  type Container = NumericList
}
```

Listing 4.5: Using un-members to recover kind soundness

antees.

If we look at the problem from the point of view of the *clients* of an abstraction, we distinguish external and internal clients. External clients supply information to an abstraction without knowing exactly which subtype of the abstraction they are dealing with. Therefore, the constraints on these missing pieces of information must only be *weakened* in subtypes. Internal clients, which are tightly related by subtyping, should be able to strengthen the result of the abstraction.

Thus, Scalina complements Scala's covariant type members with contravariant ones, which we shall call "un-members". Listing 4.5 shows a pseudo-Scala rendition of the encoding, where un-members are indicated using the **`deferred`** keyword. They are made concrete by external clients using the `... ◁{ ... }` construct.

Since the `A` type member is an input to the abstraction, it must behave contravariantly, so that `NumericList` is not allowed to strengthen the bound on the `A` un-member that it inherited from `Iterable`.

4.1.2 Methodology

To summarise the above example, a programmer should use un-members to model the input to an abstraction. This corresponds to the arguments of a method or the type parameters of a generic class. Normal members are used to define the result of the abstraction.

Note that un-members and abstract members impose an ordering discipline. A type un-member that classifies a value un-member must be refined before the value can be supplied. This corresponds to a polymorphic value in functional programming. Furthermore, types may contain abstract members, but objects must not. Therefore, an object cannot be created until the abstract members have been made concrete.

The example in Section 4.2.3 will illustrate these points in more detail.

4.1.3 Contributions

Functional abstraction clearly distinguishes a function's arguments from its result. In the object-oriented setting, a similar distinction must be made for abstract members. We introduce un-members, which safely model the input to an abstraction, and re-use traditional members to represent the result of the abstraction. Thus, an object with un-members may be thought of as a curried function that takes its keyword arguments in any order. The members of such an object represent its results. Moreover, unlike in the functional style, the result and the arguments are treated uniformly, so that the type of the result may refer to the arguments. In our setting, this is merely a curiosity, but in closely related work this becomes an important benefit [DR08].

We study purely object-oriented abstraction in a dependently typed, three-level calculus that uses the same concepts for abstraction and computation on terms and types. As in the νObj calculus, function application is decomposed into refinement and member selection. Because the level of types is modelled after the level of terms, a type-level function is modelled as a type with type un-members.

The distinction between un-members, which behave contravariantly, and normal, covariant, members, is instrumental in proving soundness on the level of types and kinds. Due to the symmetric design of our calculus, the soundness proofs proceed by similar arguments at both levels.

4.2 Scalina: syntax and intuitions

Scalina is a three-level object-oriented calculus: we distinguish terms (objects), types, and kinds. Terms are for computation, types are used for classification as well as computation, and the role of kinds is strictly limited to classification. Computation is performed using two mechanisms: member selection and member refinement. Classification is more intricate, ranging from merely structural descriptions of the classified entities over nominal classification, the intersection of classifiers, singletons, and strictly empty classifiers.

4.2.1 Syntax

Figures 4.1 and 4.2 outline Scalina's syntax.

The term level consists of member selection, member refinement, and instantiation. Analogously, a type may be a type selection, a refinement or a structural type. A structural type binds the self variable x in the members it includes; if the type of the self variable is not specified, it is assumed to be the structural type itself. We use the meta-variable R to refer to a structural type. Additionally, a type may be an intersection type, a singleton type (that depends on a path), the top or the bottom of the subtype lattice, or an un-type. Finally, we introduce $\Box T$, which stands for the result of refining all of T's un-members with unknown terms and types. We will discuss this construct in more detail in Section 4.2.2.

Figure 4.2 defines the shape of kinds, paths, values, and the typing context Γ. A path is a chain of member selections that starts with a variable or an instantiation expression **`new`** `T`, which represents an object. We mainly restrict the shape of paths to simplify the proofs in the meta-theory.

4.2.2 Core concepts

Before describing the rules that define computation and classification in Scalina, we build up intuitions about the core concepts that underlie these mechanisms.

Members and un-members

Members are the liaisons between the different levels: a type describes the value members that may be selected on the terms it classifies, as well as the type members that may be selected on the type itself. The description of a member consists of the label of the member, the classifier of the entity it stands for and – if the member is concrete – the actual entity it is bound to (its right-hand side, or RHS). For value members, the classifier is a type and the RHS is a term, and type members specify the kind that classifies the type they are bound to.

Scalina's *un-members* are a more radical departure from Scala. Un-members are used to encode parameterisation: they are placeholders for members that must be provided by the client of the abstraction, much like the arguments of a function. Un-members are turned into normal members

$t,\ u$	$::=$		term
	$\mid$	x	variable
	$\mid$	$t\,.\,l$	term selection
	$\mid$	$t \lhd \{cm\}$	term refinement
	$\mid$	$\textbf{new}\ T$	new
	$\mid$	(t)	
	$\mid$	$[\,x \mapsto t_2\,]\,t_1$	(meta) substitution
	$\mid$	$\_$	(meta) wildcard
	$\mid$	$C\,[\,t\,]$	(meta) context
TT	$::=$		type (excluding un-type)
	$\mid$	$T\,\#\,L$	type selection
	$\mid$	$T \lhd \{cm\}$	type refinement
	$\mid$	$\{x\ :\ T\ \Rightarrow\ \overline{m_i}^{\,i}\}$	structural type
	$\mid$	$\{x\ \Rightarrow\ \overline{m_i}^{\,i}\}$	structural type
	$\mid$	$T_1\ \&\ T_2$	mixin composition
	$\mid$	$p\,.\,\textbf{type}$	singleton type
	$\mid$	$\textbf{Any}$	
	$\mid$	$\textbf{Nothing}$	
	$\mid$	$\Box\, T$	necessarily T
	$\mid$	(T)	
	$\mid$	$[\,x \mapsto t\,]\,T$	(meta) substitution
	$\mid$	$[\,x \mapsto T\,]\,S$	(meta) substitution replace x.type by T in S
	$\mid$	$\_$	(meta) wildcard
$T,\ S$	$::=$		type
	$\mid$	TT	
	$\mid$	$\textbf{Un}\,[\,TT\,]$	
m	$::=$		member
	$\mid$	$\textbf{val}\,l\ :\ T$	abstract value member
	$\mid$	$\textbf{val}\,l\ :\ T = t$	concrete value member
	$\mid$	$\textbf{type}\,L\ :\ K$	abstract type member
	$\mid$	$\textbf{type}\,L\ :\ K\ =\ T$	concrete type member
	$\mid$	$\textbf{val}\,l\ :\ T\ \_$	(meta) value member (concrete/abstract)
	$\mid$	$\textbf{type}\,L\ :\ K\ \_$	(meta) type member (concrete/abstract)
	$\mid$	$[\,x \mapsto t\,]\,m$	(meta) substitution
	$\mid$	$\_$	(meta) wildcard
cm	$::=$		member refinement
	$\mid$	$\textbf{val}\,l = t$	
	$\mid$	$\textbf{type}\,L = T$	
	$\mid$	$\_$	(meta) wildcard

Figure 4.1: Scalina Syntax (terms and types)

using member refinement, which corresponds to passing arguments to a function. An entity with multiple un-members is the equivalent of a curried function: refining one of the un-members results in an entity with one less un-member to be refined. Once all un-members have been refined, the member representing the function's result may be selected to complete the application. This constitutes the essence of computation – on terms as well as types – in Scalina.

Members and un-members can be seen as the two halves of the contract specified by a classifier: members are *available* to the client, whereas it must *supply* the un-members. Note that abstract members have different semantics from un-members: an abstract member is made concrete using composition within a subtyping hierarchy, while an un-member is to be supplied by an external client. A type with abstract members cannot be instantiated. An abstract type can however be constrained (using the kind `Concrete(R)`) so that it does not contain any abstract members.

Terms

The canonical form of a term is an object. For syntactic economy, and since Scalina does not model effects yet, an object is represented by the instantiation of a type without abstract members. Conceptually, an entity is just a vessel for denoting to which entity each of its members – as described by the entity's classifier – is bound. Thus, an object contains mappings (from a label to a term) for all of the members specified in its type. Operationally, un-members can be thought of as members that are simply absent from this mapping.

Types

> If, on the term level, parameterising over functions is useful, doing the same on the level of types sounds like an obvious thing to do.
>
> Erik Meijer

To generalise Meijer's motivation for higher-kinded types [Mei07], rephrasing in our terminology: "If, on the term level, abstracting over terms that themselves abstract over terms is useful, doing the same on the level of types sounds like an obvious thing to do." Scalina manifestly supports this view by using the same abstraction mechanism on both levels: entities that abstract over other entities (using un-members) are themselves first-class entities.

KK	$::=$		kind (excluding un-kind)
	$\|$	$\mathbf{In}\,(T_1\,,\ T_2)$	
	$\|$	$\mathbf{Struct}\,(R)$	
	$\|$	$\mathbf{Nominal}\,(R)$	
	$\|$	$\mathbf{Concrete}\,(R)$	
	$\|$	(K)	
	$\|$	$[\,x\ \mapsto\ t\,]\,K$	(meta) substitution
	$\|$	$\_$	(meta) wildcard
K	$::=$		
	$\|$	KK	
	$\|$	$\mathbf{Un}\,[\,KK\,]$	
p	$::=$		path
	$\|$	x	variable
	$\|$	$p\,.\,l$	selection
	$\|$	$\mathbf{new}\ T$	new
	$\|$	(p)	
	$\|$	$[\,x\ \mapsto\ p_2\,]\,p_1$	(meta) substitution
v	$::=$		
	$\|$	$\mathbf{new}\ T$	new
Γ	$::=$		
	$\|$	$\emptyset$	
	$\|$	$\Gamma\,,\,x\ :\ T$	

Figure 4.2: Scalina Syntax (kinds, etc.)

Types play a dual role: besides computation, their main purpose is classifying terms. As explained in the introduction, types differ from terms in that they may contain abstract members for abstraction towards subtyping clients.

Types classify terms by specifying the labels and the types of the members that may be selected on these terms. A structural type classifies all terms that have the prescribed members. Note that we use kinds to distinguish nominal types from structural ones. An intersection type is inhabited by the terms that inhabit both its constituent types. A singleton type classifies exactly one object and an un-type does not classify any terms at all. An un-type is used as the classifier of a value un-member.

Type-level computation uses the same concepts as computation at the term level. However, because types may contain abstract members, we must be more careful. For soundness, type member selection is only allowed on types that (eventually) consist solely of concrete members, although the exact RHS need not be known. Type selection on a singleton type is always safe, even if the selected type member's right-hand side is not known statically. As long as it is not an un-member, the object that the singleton type depends on, could not have been created unless that member was concrete.

In Scala, these abstract type members may *only* be selected on singleton types. Scalina generalises this to the notion of *concrete types*, so that abstract type members may be selected on any type that necessarily contains only concrete type members, which naturally includes singleton types.

Similarly, it is always safe to assume that the type of the self variable does not contain any un-members: the self-variable can only be accessed as a consequence of an external member selection, which in turn is not allowed on objects with un-members. To exploit this invariant, we introduce the type $\Box T$, which stands for the result of refining all of T's un-members. We shall illustrate this with an example in Section 4.2.3

The canonical form of a type is computed by performing all allowed member selections. This corresponds to the β-normal form in functional calculi.

Kinds

Kinds are only used for classifying types: they denote which members may be selected on the types they classify. An interval kind takes over the role of the bounds of a Scala-style abstract type member: `In(S, T)` is inhabited by types that are subtypes of `T` and supertypes of `S`.

`Struct(R)` is inhabited by types that have at least the members specified in `R`. These members must be well-formed under the assumption that the self variable has the declared self type. `Nominal(R)` is similar to `Struct(R)`, except that it serves as a marker for concrete type bindings that represent classes: normalisation should not replace a type selection of this kind with its right-hand side.

Finally, T has kind `Concrete(R)` if it has at least the members specified in `R`, and none of these are abstract. Furthermore, $\Box T$ must be a subtype of the self type declared in `R`, so that such a type may be instantiated (if it is not a singleton type) or be used as the target of type member selection.

4.2.3 Example: polymorphic lists

Listing 4.6 implements polymorphic lists with `map` to illustrate Scalina's support for parametric polymorphism and higher-order functions.

First, we introduce a little syntactic sugar.

- The kind $\star$ should be expanded to **`Struct`**`({x ⇒ })`,
- the type `p.L` is shorthand for `p.`**`type`**`#L`,
- the following type members are easily expanded:
  - **`type`** `L = R` becomes **`type`** `L :` **`Struct`**`(R) = R`,
  - **`type`** `L ≺ T` means **`type`** `L :` **`Nominal`**`(R) = T`, where `R` is the expansion of `T` to its least structural supertype (by the $\prec\!\!\prec$ relation defined in Fig. 4.7).

Since type members must always be nested in other types, our program is a term that instantiates the structural type that represents our "universe" (hence the `u` as the self variable). The type `u.`**`type`**`#Fun1`, or using syntactic sugar, `u.Fun1`, corresponds to a top-level class in Scala.

The first abstraction is a polymorphic unary function. `Fun1` is a nominal type that expands to a structural type with self variable `self`, whose type is assumed to be the nominal type itself, with all its un-members refined. This special self type is crucial: without it, the body of the function could not access its arguments, as these would be considered un-members. In this example, $\Box u.Fun1$ expands to the structural type `{x ⇒` **`type`** `T1: ⋆ ;` **`type`** `T2: ⋆;` **`val`** `v: x.T1;` **`val`** `apply: x.T2}`

`Fun1` takes two type arguments: the type of its value argument (`T1`) and the type of its result (`T2`). It also requires one value argument (`v`). These arguments are un-members, which must be provided by the caller of

```
new { u ⇒
  type Fun1 ≺ {self : □ u.Fun1 ⇒
    type T1   : Un[⋆]        ; type T2   : Un[⋆]
    val v     : Un[self.T1] ; val apply : self.T2
  }
  type List ≺ {self : □ u.List ⇒
    type Element : Un[⋆]
    type map = { selfMap : □ self.map ⇒
      type Tgt : Un[⋆]
      val fun: Un[u.Fun1◁{type T1=self.Element}
                        ◁{type T2=selfMap.Tgt}]

      val apply: u.List◁{type Element=selfMap.Tgt}
    }

    val map: self.map
  }
  type Nil ≺ u.List & {self : □ u.Nil ⇒
    val map : self.map =
      new self.map & { s : □ self.map ⇒
        val apply: u.List◁{type Element = s.Tgt}
          = new (u.Nil ◁{type Element = s.Tgt})
      }
  }
  type Cons ≺ u.List & {self : □ u.Cons ⇒
    val hd: self.Element
    val tl: u.List◁{type Element=self.Element}

    val map : self.map =
      new self.map & { s : □ self.map ⇒
        val apply: u.List◁{type Element=s.Tgt}
          = new u.Cons◁{type Element=s.Tgt} & {sc ⇒
              val hd: s.Tgt
                = (fun◁{val v=self.hd}).apply
              val tl: u.List◁{type Element=s.Tgt}
                = (self.tl.map
                     ◁{type Tgt=s.Tgt}
                     ◁{val fun=s.fun}).apply
          }
    }
  }
}
```

Listing 4.6: Polymorphic List in Scalina

```
abstract class List[Element] {
  def map[Tgt](fun: Element ⇒ Tgt): List[Tgt]
}

class Nil[Element] extends List[Element] {
  def map[Tgt](fun: Element ⇒ Tgt) = new Nil[Tgt]
}

abstract class Cons[Element] extends
                 List[Element] { self ⇒
  val hd: Element
  val tl: List[Element]

  def map[Tgt](fun: Element ⇒ Tgt) = new Cons[Tgt]{
    val hd: Tgt = fun(self.hd)
    val tl: List[Tgt] = self.tl.map[Tgt](fun)
  }
}
```

Listing 4.7: Parametric List in Scala

the function. The abstract `apply` member models the function's body. It must be made concrete before an actual function value can be created.

`List` abstracts over the type of its elements (`Element`) and declares one abstract method, `map`. We define a structural type, `map`, and an abstract value member with the same name. This way, it becomes more convenient to make this member concrete, subclasses of `List` may simply use an instance of the composition of `map` with another type that makes the apply method concrete.

The implementation of the `map` "method" in `Nil` simply returns a new instance of `Nil` with the appropriate element-type. In `Cons`, the result is another cons cell that applies the supplied function to the head of the list and that recurses on the tail.

Note that `hd` and `tl` model constructor arguments: since they are required for an object of this type to be created, we use abstract members and not un-members.

Listing 4.7 shows a Scala rendition of the example that stays as close as possible to the Scalina version, using an idiomatic mix of functional and object-oriented abstractions.

Together, Listings 4.6 and 4.7 exemplify the essence of encoding Scala into Scalina. Section 4.6 specifies this encoding more generally for system F_ω^{sub}, but it easily carries over to Scala. The only subtlety arises from the discrepancy between the position-based nature of parameters, whereas members only have a name. Fortunately, this is solved by codifying a parameter with position i as a member with name arg$i$, and by using the equivalent of anonymous type functions to resolve ordering conflicts, such as in the following example.

```
type Foo[A, B] = Bar[B, A]
```

This Scala type member is encoded in Scalina as follows.

```
type Foo = {x ⇒ type arg1: Un[⋆]; type arg2: Un[⋆];
                type apply=Bar◁{type arg1=x.arg2;
                                type arg2=x.arg1}#apply}
```

4.3 Terms

4.3.1 Computation

Before we turn to the evaluation rules, we briefly consider how members are looked up at run time. For now, type members are statically bound and the role of types during evaluation is strictly limited to mapping the labels of the members of an object to terms. However, we anticipate support for virtual classes, which requires run-time lookup of types. Currently, our approach to lookup is equivalent to statically expanding types to mappings of labels to terms, with the corresponding trivial run-time lookup function.

To look up a member at run time, we use the unfold relation ($\prec$) defined in Fig. 4.3, which relies on the following helper relations: $T \ni ll \mapsto e \setminus\setminus x$ denotes that T expands to a structural type that contains a member with label ll and right-hand side e in which the self variable x is bound. ll stands for either a term- or a type-label, and e is a term or a type. Similarly, $T \ni^{\text{un}} ll$ is derivable if T has an un-member with the specified label: it expects this un-member to be refined.

Furthermore, we factor out what it means to refine a single member: **refineIf**(m', cm) can be seen as a function that returns the refinement of the un-member m' with the cm's RHS if their respective labels are the same, otherwise it simply returns m'. Similarly, $m = \textbf{refines}(m', cm)$ holds if m' and cm have the same label and m is the result of refining m' with cm. Finally, intersecting structural types corresponds to taking the union ($\uplus$) of the corresponding sets of members, with concrete members in the right type overriding corresponding members in the left one.

The actual lookup proceeds by expanding a type to the corresponding structural type, after which looking up the required label is easy. The

$\boxed{T \prec R}$ T expands to R at run time

$$\frac{T \ni L \mapsto S \setminus\!\setminus x \qquad [\,x \mapsto T\,]\,S \prec R}{T \,\#\, L \prec R}\ \text{LU\_SEL}$$

$$\frac{T \prec \{x : S \Rightarrow m_1 \mathinner{..} m_n\} \qquad \forall i \in 1..n.\ m'_i = \mathbf{refineIf}(m_i, cm)}{T \lhd\{cm\} \prec \{x : S \Rightarrow m'_1 \mathinner{..} m'_n\}}\ \text{LU\_RFN}$$

$$\frac{}{R \prec R}\ \text{LU\_REFL}$$

$$\frac{T \prec \{x \Rightarrow m_1 \mathinner{..} m_n\}}{T \prec \{x : \{x \Rightarrow m_1 \mathinner{..} m_n\} \Rightarrow m_1 \mathinner{..} m_n\}}\ \text{LU\_SELFX}$$

$$\frac{T_1 \prec \{x : S_1 \Rightarrow \overline{m_i}^{\,i}\} \qquad T_2 \prec \{x : S_2 \Rightarrow \overline{m'_j}^{\,j}\} \qquad \overline{m''_k}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m'_j}^{\,j}}{T_1 \,\&\, T_2 \prec \{x : S_2 \Rightarrow \overline{m''_k}^{\,k}\}}\ \text{LU\_MIX}$$

$$\frac{T \prec R}{(\mathbf{new}\ T)\,.\,\mathbf{type} \prec R}\ \text{LU\_SING}$$

$$\frac{p\,.\,\mathbf{type} \ni l \mapsto p' \setminus\!\setminus x \qquad ([\,x \mapsto p\,]\,p')\,.\,\mathbf{type} \prec R}{p\,.\,l\,.\,\mathbf{type} \prec R}\ \text{LU\_SINGX}$$

Figure 4.3: Type Expansion for Run-time Lookup

$\boxed{t \Rightarrow t'}$ t evaluates to t' in a single step

$$\frac{T \ni l \mapsto p \setminus\setminus x}{\mathbf{new}\, T\,.\,l \Rightarrow [\,x \mapsto \mathbf{new}\, T\,]\,p} \quad \text{E\_SEL}$$

$$\frac{\begin{array}{l} l \equiv \mathbf{label}\, cm \\ T \ni^{\mathrm{un}} l \end{array}}{(\mathbf{new}\; T) \lhd\{cm\} \Rightarrow \mathbf{new}\,(T \lhd\{cm\})} \quad \text{E\_RFN}$$

$$\frac{\begin{array}{l} T \ni l \mapsto t \setminus\setminus x \\ [\,x \mapsto \mathbf{new}\, T\,]\, t \Rightarrow t' \\ U \equiv (\mathbf{new}\; T)\,.\,l\,.\,\mathbf{type} \end{array}}{\mathbf{new}\; T \Rightarrow \mathbf{new}\,(T \,\&\, \{\_ \Rightarrow \mathbf{val}\, l : U = t'\})} \quad \text{E\_CTXMEM}$$

$$\frac{t \Rightarrow u}{t' \lhd\{\mathbf{val}\, l = t\} \Rightarrow t' \lhd\{\mathbf{val}\, l = u\}} \quad \text{E\_CTXRFNRHS}$$

$$\frac{t \Rightarrow u}{t \lhd\{cm\} \Rightarrow u \lhd\{cm\}} \quad \text{E\_CTXRFNTGT}$$

$$\frac{t \Rightarrow u}{t\,.\,l \Rightarrow u\,.\,l} \quad \text{E\_CTXSELTGT}$$

Figure 4.4: Term Evaluation

only tricky rule in the definition of the expansion relation $\prec$ is LU_SINGX. During evaluation, all types are of the shape `(new T).l1. ... .ln.type`. To reduce a selection $p.l$ to the base case, which is handled by LU_SING, we must lookup l in $p.type$ and inductively expand the resulting singleton type. To avoid extra complexity in the meta-theory, we factor in the evaluation rule for value selection instead of using evaluation directly.

The small-step evaluation relation that defines Scalina's operational semantics [WF94], is shown in Fig. 4.4. It consists of two evaluation rules and four congruence rules. The first evaluation rule, E_SEL, rewrites a member selection on an object to the RHS of that member after replacing the self variable by the object that was the target of the selection. The hypothesis that the label must be present in the object is represented as a lookup on the type. The side-condition that the member's RHS must be a path is crucial for proving type preservation: a path may only be replaced by a path. For now, all terms are paths in Scalina. However, in anticipation of adding effects to the calculus, we already distinguish paths and arbitrary terms.

E_RFN deals with refinement: it checks that the refined member was indeed an un-member (it was missing from the object), and then adds it to the object by refining the type that is used to track its members. The side-condition that l was an un-member is not necessary for proving type soundness, as the typing rules ensure that a well-typed term always meets it. We include it so that we can prove that un-members are never refined more than once by seeing that a program gets stuck if it violates that rule. However, by progress, well-typed terms never get stuck.

The only non-trivial congruence rule, E_CTXMEM, performs evaluation under member bindings, which can be thought of as running the constructor. This congruence rule is necessary to fulfil the side-condition of the rule for member selection. The shape of the type U is a technicality required by the proof of type preservation. It can be seen as an artefact of our using full-blown types for simply tracking the members of an object. The remaining congruence rules are standard.

4.3.2 Classification

Figure 4.5 defines the shape of well-typed terms. When checking a value member selection, we treat the case where the target of the selection is a path (T_SELPATH) differently from when it is not (T_SEL). Suppose we treated both cases equally. Consider e.g., `t: {x ⇒ val a: x.b.type = x.b; val b: Any}`, so that

$\boxed{\Gamma \vdash t : T}$ t has type T

$$\frac{\Gamma \vdash p : \{x \Rightarrow \mathbf{val}\, l : T\} \quad T \text{ not an un-type}}{\Gamma \vdash p\,.\,l : [\,x \mapsto p\,]\,T} \quad \text{T\_SELPATH}$$

$$\frac{\Gamma \vdash t : \{x \Rightarrow \mathbf{val}\, l : T\} \quad x \notin \mathbf{FV}(T) \quad T \text{ not an un-type}}{\Gamma \vdash t\,.\,l : T} \quad \text{T\_SEL}$$

$$\frac{\Gamma \vdash t : T \quad T \equiv \{x : S \Rightarrow m_1 \,..\, m_n\} \quad \exists i \in 1..n.\ m' = \mathbf{refines}(m_i, cm) \quad \Gamma, x : S \vdash m'\, \mathbf{WF} \quad x \notin \mathbf{FV}(cm)}{\Gamma \vdash t \lhd \{cm\} : T \lhd \{cm\}} \quad \text{T\_RFN}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec R \quad \Gamma \vdash R : \mathbf{Concrete}\,(R) \quad T \text{ not a singleton type} \quad T \text{ not of shape } \Box T'}{\Gamma \vdash \mathbf{new}\, T : T} \quad \text{T\_NEW}$$

$$\frac{\Gamma \vdash p : R}{\Gamma \vdash p : p\,.\,\mathbf{type}} \quad \text{T\_SING}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: S}{\Gamma \vdash t : S} \quad \text{T\_SUBSUME}$$

Figure 4.5: Term Classification

`t.a : t.b.`**`type`**. Now, for the singleton type `t.b.`**`type`** to be well-formed, `t` must be a path. Therefore, the selection is not allowed if this is not the case. If the declared type of the member does not rely on the self variable, the target need not be a path.

Note that the rules for member selection rely on subsumption to discard all other members in the type of the target (as well as the selected member's RHS). This is not just a matter of cosmetics: this formulation ensures that the type of the target does not contain any other un-members, as they cannot be forgotten by subsumption. In terms of function types, the underlying intuition is that subsumption cannot change the number of arguments that a function takes. We will discuss this in more detail in the section on subtyping.

T_RFN classifies member refinement – in a sense, the dual of member selection. Essentially, this corresponds to checking the type of the argument while typing function application. This check is performed by requiring that there is a member with the same label as cm, and that the result of refining this member is well-formed.

We cannot use subsumption in this rule as the target that is being refined, may have several un-members. The type of refining a term is a refinement of the type of the term that is refined. Note that this type refinement could not be replaced by an intersection type. For such a type $T\&S$ to be well-kinded, S's members must conform to T's, but here, T contains an un-member whereas S does not, and subtyping can never relate un-members to regular members.

According to T_NEW, **new** T is well-typed with type T if T statically expands to the structural type R (by $\prec\!\!\prec$, defined in Fig. 4.7), where R is of kind **Concrete**(R). The remaining side conditions rule out degenerate cases. It is necessary to expand T to R and then check R has kind **Concrete**(R) because just checking that T : **Concrete**(R), implies that a *subset* R of T is safe to be instantiated, but not necessarily T itself.

Finally, a path has the corresponding singleton type if it is well-typed using the other rules (we assume finite derivations). Subsumption gives a well-typed term a less precise type.

4.4 Types and kinds

4.4.1 Computation

Type normalisation, as shown in Fig. 4.6, is the "operational semantics" of the type level. To compute the normal form of a type, all allowed type

$\boxed{\Gamma \vdash T \rightsquigarrow T'}$ T normalises to T'

$$\frac{\begin{array}{l}\Gamma \vdash T \ni \mathbf{type}\, L : K = S \setminus\!\setminus x \\ K \text{ not nominal} \\ \Gamma \vdash [x \mapsto T]\, S \rightsquigarrow S'\end{array}}{\Gamma \vdash T \,\#\, L \rightsquigarrow S'} \quad \text{N\_SEL}$$

$$\frac{\begin{array}{l}\Gamma \vdash T \rightsquigarrow \{x : S \Rightarrow m_1 \mathinner{..} m_n\} \\ \forall i \in 1..n.\ m_i' = \mathbf{refineIf}(m_i, cm)\end{array}}{\Gamma \vdash T \triangleleft\{cm\} \rightsquigarrow \{x : S \Rightarrow m_1' \mathinner{..} m_n'\}} \quad \text{N\_RFN}$$

$$\frac{\Gamma \vdash T \rightsquigarrow \{x \Rightarrow m_1 \mathinner{..} m_n\}}{\Gamma \vdash T \rightsquigarrow \{x : \{x \Rightarrow m_1 \mathinner{..} m_n\} \Rightarrow m_1 \mathinner{..} m_n\}} \quad \text{N\_SELFX}$$

$$\frac{\begin{array}{l}\Gamma \vdash T_1 \rightsquigarrow \{x : S_1 \Rightarrow \overline{m_i}^{\,i}\} \\ \Gamma \vdash T_2 \rightsquigarrow \{x : S_2 \Rightarrow \overline{m_j'}^{\,j}\} \\ \overline{m_k''}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m_j'}^{\,j}\end{array}}{\Gamma \vdash T_1 \,\&\, T_2 \rightsquigarrow \{x : S_2 \Rightarrow \overline{m_k''}^{\,k}\}} \quad \text{N\_MIX}$$

$$\frac{\Gamma \vdash p : q\,.\,\mathbf{type}}{\Gamma \vdash p\,.\,\mathbf{type} \rightsquigarrow q\,.\,\mathbf{type}} \quad \text{N\_SNG}$$

Figure 4.6: Type Normalisation

member selections are performed, refinements and compositions of structural types are normalised to the corresponding structural type, and paths are safely rewritten if they are statically known to refer to the same object.

The selection of a type member with declared kind **Nominal**(R) is in normal form: these bindings must not be crossed. Hence the side-condition in N_SEL. If this condition were omitted, normalisation would no longer be kind-preserving, as a type of kind **Nominal**(R) would be replaced by a type of kind **Struct**(R), which is not a subkind of **Nominal**(R). By analogy to the term level, normalisation checks only the minimal side conditions, a separate theorem proves that it is kind-preserving.

Type expansion includes type normalisation, but is more aggressive: it replaces a nominal type binding with its (structural) right-hand side and widens singleton types. This is needed when calculating all the members in a type. Since type expansion must yield the least structural supertype of a type, we cannot use typing in the rules X_SING*, as this may invoke subsumption.

X_SINGVAR expands a singleton type that depends on a variable, that must therefore be in Γ. X_SINGNEW handles the other bases case, similar to run-time expansion of types. Finally, X_SINGSEL peels one layer of member selection from the path by approximating the outermost selection by its declared type.

X_NCSRY expands $\Box T$ to the expansion of T, after essentially stripping all the `Un[...]`'s from the declared types and kinds of its members. It achieves this by "pretending" to refine every un-member with an unknown right-hand side, so that un-members essentially become abstract members.

4.4.2 Subtyping

Subtyping, as defined in Fig. 4.8, is mostly standard; the main novelties result from the interaction with un-members. Un-types introduce contravariance (by ST_UN), thus deviating from the norm of covariance. Since member subtyping is covariant, an un-member with declared type **Un**[T] may only be overridden by an un-member with a declared type that is a subtype of **Un**[T], thus it has the shape **Un**[T'] with $T <: T'$. This means the overriding member weakens the restriction on the term that must be supplied by the client.

If a type S expands to a type T, then surely it is a subtype of that type T. Expanding S can be thought of as computing a least structural supertype of S, following type selections, crossing nominal type bindings and widening singleton types. Similarly, type equality ($\cong$) – the least re-

$\boxed{\Gamma \vdash T \prec\!\!\prec R}$ T expands to the structural type R

$$\frac{\Gamma \vdash T \ni \mathbf{type}\, L : K = S \;\backslash\backslash\; x \qquad \Gamma \vdash [\,x \mapsto T\,]\, S \prec\!\!\prec R}{\Gamma \vdash T \,\#\, L \prec\!\!\prec R} \;\textsc{X\_sel}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x : S \Rightarrow m_1 \mathinner{..} m_n\} \qquad \forall i \in 1..n.\; m'_i = \mathbf{refineIf}(m_i, cm)}{\Gamma \vdash T \lhd \{cm\} \prec\!\!\prec \{x : S \Rightarrow m'_1 \mathinner{..} m'_n\}} \;\textsc{X\_rfn}$$

$$\frac{}{\Gamma \vdash R \prec\!\!\prec R} \;\textsc{X\_refl}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x \Rightarrow m_1 \mathinner{..} m_n\}}{\Gamma \vdash T \prec\!\!\prec \{x : \{x \Rightarrow m_1 \mathinner{..} m_n\} \Rightarrow m_1 \mathinner{..} m_n\}} \;\textsc{X\_selfX}$$

$$\frac{\Gamma \vdash T_1 \prec\!\!\prec \{x : S_1 \Rightarrow \overline{m_i}^{\,i}\} \qquad \Gamma \vdash T_2 \prec\!\!\prec \{x : S_2 \Rightarrow \overline{m'_j}^{\,j}\} \qquad \overline{m''_k}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m'_j}^{\,j}}{\Gamma \vdash T_1 \,\&\, T_2 \prec\!\!\prec \{x : S_2 \Rightarrow \overline{m''_k}^{\,k}\}} \;\textsc{X\_mix}$$

$$\frac{x : T \in \Gamma \qquad \Gamma \vdash T \prec\!\!\prec R}{\Gamma \vdash x \,.\, \mathbf{type} \prec\!\!\prec R} \;\textsc{X\_singVar}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec R}{\Gamma \vdash (\mathbf{new}\; T) \,.\, \mathbf{type} \prec\!\!\prec R} \;\textsc{X\_singNew}$$

$$\frac{\Gamma \vdash p \,.\, \mathbf{type} \ni \mathbf{val}\, l : T \;\_ \;\backslash\backslash\; x \qquad \Gamma \vdash [\,x \mapsto p\,]\, T \prec\!\!\prec R}{\Gamma \vdash p \,.\, l \,.\, \mathbf{type} \prec\!\!\prec R} \;\textsc{X\_singSel}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x : S \Rightarrow m_1 \mathinner{..} m_h\} \qquad \forall i \in 1..n.\; m'_i = \mathbf{refineIf}(m_i, \_)}{\Gamma \vdash \Box\, T \prec\!\!\prec \{x : S \Rightarrow m'_1 \mathinner{..} m'_n\}} \;\textsc{X\_ncsry}$$

$\boxed{\Gamma \vdash T \ni m \;\backslash\backslash\; x}$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x : \_ \Rightarrow m_1 \mathinner{..} m_n\} \qquad \exists i \in 1..n.\; m_i \equiv m}{\Gamma \vdash T \ni m \;\backslash\backslash\; x} \;\textsc{X\_lu}$$

Figure 4.7: Type Expansion

$\boxed{\Gamma \vdash T <: T'}$ T is a subtype of T'

$$\frac{\begin{array}{l}\Gamma \vdash S <: T \\ \Gamma \vdash T : K \\ \Gamma \vdash T <: T'\end{array}}{\Gamma \vdash S <: T'}\ \text{ST\_TRANS}$$

$$\frac{\begin{array}{l}\Gamma \vdash S : K \\ \Gamma \vdash S \mathrel{-\!\!\prec\!\!\prec} R\end{array}}{\Gamma \vdash S <: R}\ \text{ST\_EXP}$$

$$\frac{\begin{array}{l}\Gamma \vdash S : K \\ \Gamma \vdash T : K \\ \Gamma \vdash S \cong T\end{array}}{\Gamma \vdash S <: T}\ \text{ST\_EQ}$$

$$\frac{\begin{array}{l}\Gamma \vdash T \ni \mathbf{type}\, L : K \ \backslash\backslash\ x \\ \Gamma \vdash K <: \mathbf{In}\,(\_ , S)\end{array}}{\Gamma \vdash T \,\#\, L <: S}\ \text{ST\_ABS\_UPPER}$$

$$\frac{\begin{array}{l}\Gamma \vdash T \ni \mathbf{type}\, L : K \ \backslash\backslash\ x \\ \Gamma \vdash K <: \mathbf{In}\,(S , \_)\end{array}}{\Gamma \vdash S <: T \,\#\, L}\ \text{ST\_ABS\_LOWER}$$

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash T_1 \lhd\{\mathbf{type}\, L = U\} <: T_2 \lhd\{\mathbf{type}\, L = U\}}\ \text{ST\_INVAR}$$

$$\frac{\begin{array}{l}\Gamma \vdash S <: S_2 \\ \forall j \in 1..k.\ \exists i \in 1..n.\ (m_i \overset{\text{label}}{\equiv} m'_j \wedge \Gamma \vdash m_i <: m'_j) \\ \forall i \in 1..n.\ (m_i\, \mathbf{deferred} \Rightarrow \exists j \in 1..k.\ m_i \overset{\text{label}}{\equiv} m'_j)\end{array}}{\Gamma \vdash \{x : S \Rightarrow m_1 .. m_n\} <: \{x : S_2 \Rightarrow m'_1 .. m'_k\}}\ \text{ST\_R}$$

$$\frac{\begin{array}{l}\Gamma \vdash T_1 \mathrel{-\!\!\prec\!\!\prec} \{x : \_ \Rightarrow m_1 .. m_n\} \\ \Gamma \vdash T_2 \mathrel{-\!\!\prec\!\!\prec} \{x : \_ \Rightarrow m'_1 .. m'_k\} \\ \forall i \in 1..k.\ (m'_i\, \mathbf{deferred} \Rightarrow \exists j \in 1..n.\ m'_i \overset{\text{label}}{\equiv} m_j)\end{array}}{\Gamma \vdash T_1 \,\&\, T_2 <: T_1}\ \text{ST\_IelimR}$$

$$\frac{\begin{array}{l}\Gamma \vdash T_1 \mathrel{-\!\!\prec\!\!\prec} \{x : \_ \Rightarrow m_1 .. m_n\} \\ \Gamma \vdash T_2 \mathrel{-\!\!\prec\!\!\prec} \{x : \_ \Rightarrow m'_1 .. m'_k\} \\ \forall i \in 1..n.\ (m_i\, \mathbf{deferred} \Rightarrow \exists j \in 1..k.\ m_i \overset{\text{label}}{\equiv} m'_j)\end{array}}{\Gamma \vdash T_1 \,\&\, T_2 <: T_2}\ \text{ST\_IelimL}$$

Figure 4.8: Subtyping (1/2)

$\boxed{\Gamma \vdash T <: T'}$ T is a subtype of T'

$$\frac{\Gamma \vdash T <: T_1 \qquad \Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \,\&\, T_2} \quad \text{ST\_IINTRO}$$

$$\frac{\Gamma \vdash T : K \qquad T \text{ not an un-type}}{\Gamma \vdash T <: \mathbf{Any}} \quad \text{ST\_ANY}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \mathbf{Nothing} <: T} \quad \text{ST\_NOTHING}$$

$$\frac{\Gamma \vdash T <: S}{\Gamma \vdash \mathbf{Un}[\,S\,] <: \mathbf{Un}[\,T\,]} \quad \text{ST\_UN}$$

Figure 4.8: Subtyping (2/2)

flexive, symmetric, and transitive relation that includes normalisation – is included (by ST_EQ).

The rules ST_ABS_UPPER and ST_ABS_LOWER incorporate the declared kinds of abstract type members into the subtyping relation.

For simplicity, the current version of Scalina does not model variance for type constructors, which explains why ST_INVAR considers type un-members to be invariant.

Besides the usual width- and depth-subtyping, subtyping of structural types must take extra care to never forget any un-members during subsumption. Intuitively, subsumption allows a client to weaken its expectations of a type, but it should not relax the client's own obligations.

Subtyping of members is defined in Fig. 4.9. Value members always behave covariantly; a type member becomes invariant as soon as it is made concrete. This is related to the fact that Scalina does not admit late-binding for type members.

To relate this to subtyping of function types in system F_ω^{sub}, a type of the shape $S \to T$ can only be a subtype of a type with the same shape, i.e., a function of the same arity. In our system, the number of un-members denotes the "arity" and types can only be subtypes if they have the same un-members. **`Any`** constitutes the only safe exception to this rule. It is safe for a structural type with un-members to be a subtype of **`Any`**, as no members can be selected on a term that is only known to have type **`Any`**.

Similarly, if subtyping forgets either constituent of an intersection type,

$\boxed{\Gamma \vdash m <: m'}$ m is a submember of m'

$$\frac{\Gamma \vdash T <: T'}{\Gamma \vdash \mathbf{val}\, l : T\ \_ <: \mathbf{val}\, l : T'\ \_}\ \text{SM\_VAL}$$

$$\frac{\Gamma \vdash K <: K'}{\Gamma \vdash \mathbf{type}\, L : K\ \_ <: \mathbf{type}\, L : K'}\ \text{SM\_TYPEA}$$

$$\frac{}{\Gamma \vdash \mathbf{type}\, L : K = \_ <: \mathbf{type}\, L : K = \_}\ \text{SM\_TYPEC}$$

Figure 4.9: Subtyping for members

any un-members in the forgotten type must still be present in the remaining one. For example, suppose we have a term of type {x: S ⇒ **val** a: **Un**[T]; **val** b: T=x.a} & {x : S ⇒ **val** b : T}, with S = {x ⇒ **val** a : T; **val** b: T}. If we were allowed to subsume the term's type to {x : S ⇒ **val** b : T}, we could access b before a had been refined.

For brevity, we use m **deferred** to check that m's classifier is of the shape **Un**[T] or **Un**[K].

4.4.3 Classification

For the constructs that are shared by terms and types, classification is largely analogous, as shown in Fig. 4.10. The main difference is that we have to be careful to only select types that will eventually become concrete. For objects, this is always the case, but types with abstract type members are still types. Whereas a term with type T is known to contain concrete versions of all members (not including un-members) in T, a type with kind **Struct**(R) may contain abstract members. Therefore, we introduce the kind **Concrete**(R) that classifies only types with only concrete members.

The kind of a structural type reflects the type members that may be selected on that type. To be well-kinded according to K_R, the members of a structural type must be well-formed under the assumption that the self variable has the declared self type. The well-formedness judgement for members is defined in Fig. 4.11.

The intersection of two structural types is classified by the kind that tracks the union of their members. Note that the self type of the overriding type (the right-most constituent) must be a subtype[1] of the type containing

[1]This is a slight simplification of νObj, where S_1 need not be a subtype of S_2. νObj's composition operator requires the self type for the composition to be specified explicitly.

$\boxed{\Gamma \vdash T : K}$ T has kind K

$$\frac{\begin{array}{l} R \equiv \{x : S \Rightarrow m_1 \mathinner{..} m_n\} \\ \forall i \in 1..n.\ \Gamma\,, x : S \vdash m_i\ \mathbf{WF} \\ m_1 \mathinner{..} m_n\ \mathbf{noDuplicates} \end{array}}{\Gamma \vdash R : \mathbf{Struct}\,(R)}\ \textsc{K\_R}$$

$$\frac{\Gamma \vdash \{x : \{x \Rightarrow m_1 \mathinner{..} m_n\} \Rightarrow m_1 \mathinner{..} m_n\} : K}{\Gamma \vdash \{x \Rightarrow m_1 \mathinner{..} m_n\} : K}\ \textsc{K\_RX}$$

$$\frac{\begin{array}{l} \Gamma \vdash T_1 : \mathbf{Struct}\,(\{x : S_1 \Rightarrow \overline{m_i}^{\,i}\}) \\ \Gamma \vdash T_2 : \mathbf{Struct}\,(\{x : S_2 \Rightarrow \overline{m'_j}^{\,j}\}) \\ \forall i \in 1..n.\ \forall j \in 1..k.\ (m_i \overset{\text{label}}{\equiv} m'_j \Rightarrow \Gamma \vdash m'_j <: m_i) \\ \overline{m''_k}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m'_j}^{\,j} \\ \Gamma \vdash S_2 <: S_1 \end{array}}{\Gamma \vdash T_1 \,\&\, T_2 : \mathbf{Struct}\,(\{x : S_2 \Rightarrow \overline{m''_k}^{\,k}\})}\ \textsc{K\_mix}$$

$$\frac{\begin{array}{l} \Gamma \vdash T : \mathbf{Struct}\,(\{x : S \Rightarrow m_1 \mathinner{..} m_n\}) \\ \Gamma \vdash \Box\, T <: S \\ \forall i \in 1..n.\ m_i\ \mathbf{nonAbstract} \end{array}}{\Gamma \vdash T : \mathbf{Concrete}\,(\{x : S \Rightarrow m_1 \mathinner{..} m_n\})}\ \textsc{K\_concrete}$$

$$\frac{\begin{array}{l} \Gamma \vdash p : T \\ \Gamma \vdash T : \mathbf{Struct}\,(\{x : S \Rightarrow m_1 \mathinner{..} m_n\}) \end{array}}{\Gamma \vdash p\,.\,\mathbf{type} : \mathbf{Concrete}\,(\{x : S \Rightarrow m_1 \mathinner{..} m_n\})}\ \textsc{K\_sing}$$

$$\frac{\Gamma \vdash p\,.\,\mathbf{type} : \mathbf{Concrete}\,(\{x \Rightarrow \mathbf{type}\, L : K\})}{\Gamma \vdash p\,.\,\mathbf{type}\,\#\,L : [\,x \mapsto p\,]\,K}\ \textsc{K\_selpath}$$

$$\frac{\begin{array}{l} \Gamma \vdash T : \mathbf{Concrete}\,(\{x \Rightarrow \mathbf{type}\, L : K\}) \\ x \notin \mathbf{FV}(\,K) \end{array}}{\Gamma \vdash T\,\#\,L : K}\ \textsc{K\_sel}$$

$$\frac{\begin{array}{l} \Gamma \vdash T : \mathbf{Struct}\,(\{x : S \Rightarrow m_1 \mathinner{..} m_n\}) \\ \exists i \in 1..n.\ m' = \mathbf{refines}(m_i, cm) \\ \Gamma\,, x : S \vdash m'\ \mathbf{WF} \\ m''_1 \mathinner{..} m''_n = m_1 \mathinner{..} m_n \uplus m' \\ x \notin \mathbf{FV}(\,cm) \end{array}}{\Gamma \vdash T \lhd \{cm\} : \mathbf{Struct}\,(\{x : S \Rightarrow m''_1 \mathinner{..} m''_n\})}\ \textsc{K\_rfn}$$

Figure 4.10: Classifying Types (1/2)

$\boxed{\Gamma \vdash T : K}$ T has kind K

$$\frac{\begin{array}{c}\Gamma \vdash T : K_1 \\ \Gamma \vdash K_1 <: K_2\end{array}}{\Gamma \vdash T : K_2} \quad \textsc{K\_subsume}$$

$$\frac{}{\Gamma \vdash \textbf{Any} : \textbf{Struct}\,(\{x \Rightarrow \})} \quad \textsc{K\_any}$$

$$\frac{\Gamma \vdash R : \textbf{Struct}\,(R)}{\Gamma \vdash \textbf{Nothing} : \textbf{Struct}\,(R)} \quad \textsc{K\_nothing}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \textbf{Un}\,[\,T\,] : \textbf{Struct}\,(\{x \Rightarrow \})} \quad \textsc{K\_un}$$

Figure 4.10: Classifying Types (2/2)

$\boxed{\Gamma \vdash m\,\textbf{WF}}$ m is well-formed

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \textbf{val}\,l : T = t\,\textbf{WF}} \quad \textsc{M\_valC}$$

$$\frac{\Gamma \vdash T : \textbf{Struct}\,(R)}{\Gamma \vdash \textbf{type}\,L : \textbf{Nominal}\,(R) = T\,\textbf{WF}} \quad \textsc{M\_typeNom}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \textbf{type}\,L : K = T\,\textbf{WF}} \quad \textsc{M\_typeC}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \textbf{val}\,l : T\,\textbf{WF}} \quad \textsc{M\_valA}$$

$$\frac{\Gamma \vdash K\,\textbf{WF}}{\Gamma \vdash \textbf{type}\,L : K\,\textbf{WF}} \quad \textsc{M\_typeA}$$

Figure 4.11: Well-formedness of members

the overridden members. Each overriding member must be a submember of the corresponding member in T_1.

There are two ways for deriving that a set of members of a type are concrete. The easy way is if that type is a singleton type. Otherwise, for a type T to be classified as having a certain set of concrete members $m_1..m_n$, it must have a structural kind with declared self-type S and $\Box T <: S$. Naturally, this structural kind must denote the $m_1..m_n$ as concrete. However, due to subsumption, this set of members may be a *subset* of the actual members of T. Nonetheless, any type member in R may safely be selected: it will eventually become concrete.

Given the notion of types with concrete members – which was not necessary at the lower level since terms may not contain abstract members – type member selection is classified analogously to value member selection. Type refinement is almost literally the same as at the term level, as is subsumption.

Finally, the top and the bottom of the subtype lattice must be classified, as well as un-types. **`Any`**, and certainly **`Nothing`**, are not essential to the type system. However, **`Any`** is needed to be able to select a member on the universe (the top-level object): that member must have a type that does not depend on the universe's self variable, but all user-defined types are (indirectly) selected on the universe's self type. Since **`Any`** exists outside of the user-defined universe, it can serve this purpose. An alternative would be to introduce another variable binding construct, such as `let`. **`Nothing`** is used as the default lower bound of the interval kind. It may be given the same kind as any well-kinded structural type.

4.4.4 Subkinding

Subkinding (Fig. 4.12) introduces contravariance for un-kinds (SK_UN), so that type un-members conform contravariantly. Other than that, the relation defines a simple lattice, with the interval kind at the top.

A nominal type can be subsumed to a structural one (SK_NOM) – but not vice versa! A concrete type is also a structural one (SK_CONC). A structural type that includes all the members in R, is thus in the interval `(`**`Nothing`**`, R)` (SK_STRUCT).

Interval inclusion gives rise to subkinding (SK_CTX_IN). The kinds that classify structural types and concrete types have similar subsumption properties based on subtyping (SK_CTX_CONC, SK_CTX_STRUCT).

This new self type must be a subtype of the S_i.

$\boxed{\Gamma \vdash K <: K'}$ K is a subkind of K'

$$\frac{\Gamma \vdash K_2 <: K_1}{\Gamma \vdash \mathbf{Un}\,[\,K_1\,] <: \mathbf{Un}\,[\,K_2\,]} \quad \text{SK\_UN}$$

$$\frac{}{\Gamma \vdash \mathbf{Nominal}\,(R) <: \mathbf{Struct}\,(R)} \quad \text{SK\_NOM}$$

$$\frac{}{\Gamma \vdash \mathbf{Concrete}\,(R) <: \mathbf{Struct}\,(R)} \quad \text{SK\_CONC}$$

$$\frac{}{\Gamma \vdash \mathbf{Struct}\,(R) <: \mathbf{In}\,(\mathbf{Nothing}\,,\ R)} \quad \text{SK\_STRUCT}$$

$$\frac{\Gamma \vdash R_1 <: R_2}{\Gamma \vdash \mathbf{Concrete}\,(R_1) <: \mathbf{Concrete}\,(R_2)} \quad \text{SK\_CTX\_CONC}$$

$$\frac{\Gamma \vdash R_1 <: R_2}{\Gamma \vdash \mathbf{Struct}\,(R_1) <: \mathbf{Struct}\,(R_2)} \quad \text{SK\_CTX\_STRUCT}$$

$$\frac{\begin{array}{l}\Gamma \vdash T_2 <: S_2 \\ \Gamma \vdash S_1 <: T_1\end{array}}{\Gamma \vdash \mathbf{In}\,(T_1\,,\ T_2) <: \mathbf{In}\,(S_1\,,\ S_2)} \quad \text{SK\_CTX\_IN}$$

Figure 4.12: Subkinding

4.5 Design space

After introducing Scalina in detail, we look at the bigger picture by briefly positioning its abstraction mechanisms in the design space. Scalina's main goal is to provide the essential features to model an object-oriented language – such as objects with named members, mutual recursion through the self variable, and mixin composition – while also allowing functional concepts to be encoded with the same safety guarantees as in functional calculi.

For an expedient exploration of the design space of abstraction mechanisms, we shall restrict ourselves to investigating the instantiations of the following question: "Is a *term/type* that abstracts from a *term/type* using *a parameter/an abstract member* a first-class *term/type*?" We will answer these questions for Java, Scala, system F_{ω}^{sub}, and Scalina.

Note that we use 'term' to denote anything that resides at the 'base' level, such as an object in OO, or a function in FP. We do not imply any connection to syntactic terms. A 'type' is something that classifies terms, and thus resides at the next level. We use 'entity' to mean either a term or a type, when it only matters that the denoted entity can perform computation. Finally, a 'classifier' classifies an entity: a type classifies a term, and a kind classifies a type.

Table 4.1 gives an overview of the analysis discussed below. The row of an entry determines what is abstracted from, and the column denotes the level of the abstraction. When the constructs in a part of the table are not all first-class constructs, (+) and (-) are used to make the distinction. We consider a construct "first-class" if it can be abstracted over. '/' means 'not supported'. The superscripts in parentheses are intended to aid the reader in correlating the schematic representation in table 4.1 and the following discussion.

In Java, a term may only abstract from a term^{1} or a type^{2} using parameterisation (functional abstraction): as already mentioned, a method abstracts from the concrete values of its arguments, but a method is not a first-class term in Java. Similarly, a polymorphic method may have type parameters, but again, such a method is not a first-class entity. Terms cannot have abstract members$^{3,\ 4}$.

At the type level, still in Java, a constructor argument^{5} can be considered as parameterising a type in a value (a constructor, like a method, is not a first-class term). Since Java 5.0, a class (a type) may take type parameters^{6} , but a parameterised type is not a first-class type unless it is fully applied. Finally, a class with an abstract method^{7} is a first-class type that abstracts from a term. When deciding whether a type is first-class,

Construct	…in term (1st class?)	…in type (1st class?)
Java		
Parameter (FP)		
Term	method (-) [1]	constructor (-) [5]
Type	method (-) [2]	generic class (-) [6]
Abs. mem. (OO)		
Term	/ [3]	class w/abs. method (+) [7]
Type	/ [4]	/
Scala		
Parameter (FP)		
Term	method (-) anon. function (+) [8]	constructor (-)
Type	method (-)	generic class (+) [9]
Abs. mem. (OO)		
Term	/ [15]	abs. val/def (+)
Type	/ [16]	abs. type member (+) [10]
System F_ω^{sub}		
Term	$\lambda x : T.t$ [11]	/[14]
Type	$\lambda X <: T.t$ [12]	$\lambda X :: K.T$ [13]
Scalina		
Term (ext.)	obj. w/value un-member [17]	type w/value un-member [18]
Term (int.)	/ [19]	type w/value abs. mem. [21]
Type (ext.)	obj. w/type un-member [23]	type w/type un-member [24]
Type (int.)	/ [20]	type w/type abs. mem. [22]

Table 4.1: Abstraction mechanisms: overview

(The superscripts link the entries in the table to the relevant part of the discussion.)

we do not take into account whether it may be instantiated.

Scala introduces several improvements over Java. Firstly, λ-abstraction[8] is directly supported, thus a term abstracting from a term is a value. Secondly, we recently implemented direct support for type constructor polymorphism in Scala 2.5, so that a parameterised type[9] is considered a first-class (higher-kinded) type [MPO07]. Finally, a class may have abstract *type* members[10] .

System F_ω^{sub} is a purely functional calculus. Naturally, we only consider its support for abstraction using parameterisation. A term that abstracts from a term is written as $\lambda x : T.t$[11] . A term may also be parametric in a type: $\lambda X : K.t$[12] . A type abstracts from a type as $\lambda X : K.T$[13] . To abstract from terms at the level of types[14] , we must turn to dependently typed versions of the calculus.

The overview in table 4.1 contains a striking void in the quadrant of terms with abstract members[3,4,15, 16] .

Nevertheless, Self, one of the earliest OO languages, represents a method as an object with "argument slots" [US87]. In other words, a method is a first-class term (i.e., an object) that uses abstract members to abstract from other terms.

Finally, Scalina's object-oriented abstraction mechanisms are split out with respect to the clients they cater to. An object can abstract over an object that is to be supplied by an external client using a value un-member[17] . A type may abstract over a value in the same way [18] . Since objects are not allowed to have abstract members, they cannot abstract over terms[19] or types[20] for internal clients. Types, on the other hand, may contain abstract value[21] or type[22] members. Finally, an object[23] or a type[24] type can abstract over a type using a type un-member.

Thus, our brief survey has shown that Scalina supports all variations of abstraction mechanisms that are used in practice, without admitting too many features that do not appear in a full language. We designed Scalina so that it includes the main concepts of object-oriented languages, such as objects with named members and mutual recursion through a self variable, mixin composition, subtyping, and lightweight dependent types. Furthermore, although Scalina does not contain any mechanisms for parameterisation, it can safely and straightforwardly encode functional-style abstraction as well. Others have studied the advantages of OO-style abstraction over the functional style, and vice versa [BOW98, TT99, Ern01, Ern06].

4.6 Encoding system F_ω^{sub}

Table 4.2 shows how terms, types and kinds from system F_ω^{sub} [Pie02, Ch. 31] can be encoded in Scalina. Using Pierce's terminology, an abstraction is modelled as an object with an un-member `a` that represents the argument, and a member `apply` that encodes the body of the abstraction. Note that we have to infer the type `T'`. Application is decomposed into refining the `a` un-member with the encoding of the actual argument, and selecting the `apply` member.

The encoding of a polymorphic value re-uses the pattern we used for term abstraction, except that the argument is now a type un-member instead of a term-level one. We use an interval kind to model type bounds: '$<: T$' becomes '`: In(Nothing,` $[\![$`T`$]\!]$`)`'. Type application does not present new challenges.

At the level of types, function types and universal types become the obvious structural types, which we established when encoding (polymorphic) function values. Similarly, we simply hoist our term-level abstraction and application to the type level to replace operator abstraction and application. The kind-level is easily derived from the type that encodes operator abstraction.

The evaluation of the encoding of a value application proceeds by E_RFN pushing the refinement of the object to the object's type, so that E_SEL can look up the `apply` member in the type that now has a concrete value for it. Evaluating a type application also uses E_RFN to push the concrete type information into the type of the value, which tracks the value's members. However, this binding is never used during later evaluation steps, as the only types that interact with evaluation, are those that can be used to instantiate a new object. These types must statically expand to a structural type, which is not possible for type un-members.

Finally, we note that the contravariant rule for un-member conformance means that Scalina can encode full system F_{ω}^{sub}, and that the undecidability of the latter should thus carry over to Scalina.

4.7 Meta-theory

The traditional term-level safety proofs show that it suffices to type check a program once in order to guarantee certain properties for every possible evaluation trace. In Scalina, we ensure that member lookup never fails to find the required label with the corresponding right-hand side, and that an un-member is at most refined once on the same object.

The type-level guarantees are similar, though more subtle. Since type selection is only well-kinded if the target of the selection is known to become a concrete type during type checking, we ensure that selection can always proceed on types of kind **Concrete**(R). Note that we consider certain other type selections, such as selecting a nominal type, to be in canonical form, so that this kind of selection is not expected to proceed.

The complexity of the calculus warrants mechanising the meta-theoretic proofs. Although we have worked out the essential theorems on paper, and mechanised the meta-theory of a small subset of the calculus, more work is needed to adequately demonstrate full correctness. However, we intend to first simplify Scalina and experiment with level polymorphism before we invest in a fully mechanised meta-theory.

$[\![t]\!]^{t'}$	$\equiv$	replace the free variable in the encoding of t with t'
$[\![\lambda x : T.t]\!]$	$\equiv$	**new** {self ⇒ **val** a: **Un**[⟦T⟧]; **val** apply: T′ = ⟦t⟧$^{\texttt{self.a}}$}
$[\![t\,t']\!]$	$\equiv$	(⟦t⟧ ◁{**val** a = ⟦t′⟧}).apply
$[\![\lambda X <: T.t]\!]$	$\equiv$	**new** {self ⇒ **type** a: **Un**[**In**(**Nothing**, ⟦T⟧)]; **val** apply: T′ = ⟦t⟧$^{\texttt{self.a}}$}
$[\![t\,[T]]\!]$	$\equiv$	(⟦t⟧ ◁{**type** a = ⟦T⟧}).apply
$[\![\mathrm{Top}]\!]$	$\equiv$	**Any**
$[\![T \to T']\!]$	$\equiv$	{**val** a: **Un**[⟦T⟧]; **val** apply: ⟦T′⟧}
$[\![\forall X <: T.T']\!]$	$\equiv$	{**type** a: **Un**[**In**(**Nothing**, ⟦T⟧)]; **val** apply: ⟦T′⟧$^{\texttt{self.a}}$}
$[\![\lambda X :: K.T]\!]$	$\equiv$	{**type** a: **Un**[⟦K⟧]; **type** apply : K′ = ⟦T⟧$^{self.a}$}
$[\![T\,T']\!]$	$\equiv$	(⟦T⟧ ◁{**type** a = ⟦T′⟧})#apply
$[\![*]\!]$	$\equiv$	**Struct**({x ⇒ })
$[\![K \Rightarrow K']\!]$	$\equiv$	**Struct**({self ⇒ **type** a: ⟦K⟧; **type** apply: ⟦K′⟧})

Table 4.2: Informal encoding of system F_{ω}^{sub}-syntax in Scalina

4.8 Related work

4.8.1 Modelling OO

Given the wealth of research on extensions of the λ calculus, it is only natural that studies of the essence of object-oriented languages build on these ideas. Even though encoding objects requires a lot of extra machinery, such as records, subtyping, and recursive types, this complexity is probably inherent. However, modelling OO using a combination of FP *and* OO seems to fail Occam's razor. Nonetheless, a lot of object calculi fall in this category [IPW01, FKF98, CGLO06].

The other side of the spectrum – using a purely object-oriented calculus without FP concepts – can be traced back to Abadi and Cardelli's seminal work [AC95, AC96]. However, in their first-order system, "an object type is invariant in its component types". Thus, object types cannot encode function types in the presence of subtyping, as the latter require a mix of contravariance and covariance. To solve this, they introduce universal and existential quantification in their second-order system. Universal quantification, like un-members, behaves contravariantly. Similarly, existential quantification introduces covariance, which we allow for normal members. A similarity of interest is Cardelli's notion of power type [Car88a], which corresponds to Scalina's `In(S, T)` kind.

In other respects, Abadi and Cardelli's first-order system is more powerful than Scalina: our refinement operator does not allow recursion through the self variable. However, this limitation simplifies the calculus without ruling out refinement's primary use, which is similar to function application: supplying a value to a function does not rely on the values supplied earlier.

To further the similarity with application, refinements do not require type or kind annotations for the supplied entities. Nonetheless, it is possible to have type un-members that classify value un-members: the type un-members must then be refined before the value un-members, because the supplied types determine the acceptable values for the value un-members. Note that we use first-class types, which do have a self variable, for overriding.

Scalina was directly inspired by the νObj calculus [OCRZ03]. The main difference is that Scalina introduces un-members and refinement at the term and type level. νObj uses class templates for term-level abstraction, and only provides covariant abstract type members for type-level abstraction. The latter implies that well-formed type-level applications may surprise the type function with unexpected arguments. It is important to note that this does not have any impact on the run-time behaviour of such programs. It does however fail to provide the type-level equivalent of the guarantees that term-level abstraction builders have come to rely on.

4.8.2 Kind soundness

The νObj calculus [OCRZ03] does not possess the kind soundness property that was discussed in Section 3.3.5. Type parameters are encoded as abstract type members, which behave *covariantly.* There is no way to enforce contravariance for these type members. Thus, type-level functions cannot be encoded faithfully, as their arguments must adhere to a contravariant regime.

Related work seems to deviate from νObj's design, although making a precise comparison is complicated by the differences in features supported by the various approaches. In the notation of Cardelli [Car88b], the types from the example in Listing 4.3 are classified as follows:

```
NumericList : ALL[A::POWER[Number]] TYPE
Container : ALL[X::TYPE] TYPE
```

Cardelli does not define subkinding for these kinds, but does define subtyping for polymorphic functions ("All [X::K] B <: All [X::K$'$] B$'$ if K$'$<::K (where <:: denotes a subkind relation[, …]), and B<:B$'$ under the assumption that X::K$'$"). It seems reasonable to lift this rule (which deals with functions that take a type to yield a *value*) to the level of kinds, which results in our rule that deals with functions that take a type to yield a *type.*

Similarly, in the notation of Compagnoni and Goguen [CG03]:

```
NumericList : Pi A <: Number : ⋆. ⋆
Container : Pi X <: T⋆ : ⋆. ⋆
```

Although the authors require these bounds to be equal for the kinds to be comparable (their treatment does not include subkinding), we generalise based on the same observation as the previous paragraph, but using a slight different source of inspiration. Namely, Full System $F_{<:}$'s [CMMS94] rule that deals with bounded quantification at the value level (Sub Forall) also requires *contravariance* for the bounds of the quantifier.

We recover early error detection in Scalina by differentiating covariant and contravariant members, instead of assuming they all behave covariantly. This distinction corresponds to the fact that some members abstract over input, whereas others represent the output of the abstraction. Input members should behave contravariantly, like the types of a method's parameters, whereas covariance is required for output members, which correspond to a method's result type. With this distinction, a purely object-oriented calculus can encode functional-style abstraction with the same safety guarantees.

Moreover, in some ways, the object-oriented approach is more powerful than the functional one. In their work on the MixML calculus [DR08], Dreyer and Rossberg present a novel module system for ML, inspired by Scala, and they point out that the uniform treatment of a module's imports and exports is an important benefit of the OO style of abstraction over ML's functors. A functor's exports (the result, a member) may depend on its imports (its arguments, un-members in Scalina), but not the other way around. As in MixML, this arbitrary restriction is not present in Scalina, and an un-member's type may depend on a member, which corresponds to the import specifications of a functor referring to type components in its exports.

4.9 Conclusion

While developing the theoretical aspect of extending Scala with type constructor polymorphism, we discovered that existing Scala calculi, such as the νObj calculus, could not fully encode our extension. Thus, we set out to design Scalina, which improves the νObj calculus in order to provide better support for safe type-level abstraction. More specifically, Scalina is *kind sound*, which means that type-level computation (or abstraction) does not go wrong. To achieve this, we distinguish "input" and "output" members. Given the covariant nature of Scala's abstract type members, they should only be used for output.

Thus, Scalina introduces un-members to deal with input. Even though un-members are a fairly simple construct, they make the calculus significantly more expressive, as illustrated in the previous section by the relation to the MixML calculus. Additionally, we suspect them to be useful in modelling safe object initialisation, and plan to investigate this in future work.

Beyond this original motivation, Scalina has evolved into a research vehicle for experimenting with language constructs in support of a scalable type system. From the outset, our design has emphasised uniformity, as witnessed primarily by the similarity of the abstraction mechanisms at the level of terms and types. While working on the calculus as well as with applications of type constructor polymorphism, our striving for uniformity in abstraction mechanisms across levels has been confirmed, as discussed in the previous chapters.

The following chapter motivates our plans for further evolving and simplifying Scalina from a broader perspective. Briefly, more technically, these include introducing level polymorphism, which is a promising core construct of a scalable type system. Essentially, the idea is to collapse the different levels (of terms, types, kinds, ...) into a single level of level-indexed entities, so that, in the extreme, the same abstraction mechanism caters to a diverse audience of application programmers, expert library developers, as well as designers of type system modules. Moreover, with the idea of expressing specifications as types, this abstraction mechanism also plays a role in modularising software specification.

Because Scalina's theory is still changing, we have not yet invested in fully mechanising its meta-theory. Nonetheless, we have proof sketches for the important theorems for the full calculus, as well as a Coq development of progress and preservation for a small subset of the calculus. We will continue this formal development in tandem with the ongoing refinement of the theory.

Chapter 5

Conclusion and Future Research Direction

5.1 Summary of contributions

5.1.1 Type constructor polymorphism

The high-level goal of our research is to improve the type system's ability to assist the programmer in efficiently producing correct software. Our contributions bring this goal closer by providing the Scala library designer with higher-order genericity, or *type constructor polymorphism*. Additionally, we have designed Scalina as a new model for the essence of Scala's type system, including our extension, and we plan to refine it further as part of our ongoing research on a scalable type system.

Haskell has supported higher-order parametric polymorphism since its inception, and the first-order fragment is now common in object-oriented languages. We have shown that the higher-order generalisation of "genericity" is equally useful in object-oriented programming, and that it integrates soundly into the type system of a modern object-oriented language, such as Scala. Moreover, it was implemented in the full Scala compiler with reasonable effort, and has since been put to good use in refactoring Scala's collection library, as well as by other researchers in their work on polymorphically embedding domain-specific languages [HORM08].

The integration of type constructor polymorphism into Scala gave rise to a rich level of kinds, including the type-dependent interval kind that captures lower and upper bounds, and the polarised type-function kind constructor that deals with definition-site variance annotations. Fortunately, the compiler can fully infer these kinds so that the programmer does not have to deal with them. In fact, they are not expressible in the surface syntax. Furthermore, the combination with subtyping and Scala's support for implicit arguments subsumes and even exceeds Haskell's type class construct.

As a concise illustration of the power of higher-order genericity, we presented a novel implementation of the Builder pattern, which makes essential use of type constructor polymorphism. It can be seen as the dual of the Iterator pattern, in the sense that it encapsulates the piecemeal construction of a collection (specified by a type constructor, such as `List`), whereas Iterator captures the consumption of a collection.

5.1.2 The essence of abstraction in Scala

In Chapter 4, we described Scalina, our calculus that takes a first step towards unifying the abstraction mechanisms across different levels. Scalina was originally focussed on proving soundness of type constructor polymorphism. When working on the theoretical underpinnings for type constructor polymorphism, we first experimented with encoding our extension in the νObj calculus [OCRZ03]. However, it turned out that a faithful encoding was just out of reach. Thus, we developed Scalina and incorporated un-members as a fix for the issue that we discovered with the νObj calculus.

Scalina has since evolved to an interesting calculus in its own right due to its uniform treatment of abstraction. The same style of abstraction allows values and types to abstract over values and types, in any combination. Moreover, the functional abstraction mechanisms of system F_{ω}^{sub} can be encoded faithfully.

In fact, Scalina's approach to abstraction is strictly more powerful than functional abstraction in the sense that the input of an abstraction may refer to its result. This may seem like a curiosity, but independently, Dreyer and Rossberg included a similar construct in their MixML calculus, which models a powerful, novel module system for ML [DR08]. In this context it makes perfect sense for a module's imports (the input of the abstraction, its un-members) to refer to its exports (the result of the abstraction, its members). Finally, we suspect un-members to be useful in modelling safe object initialisation, and plan to investigate this in future work.

5.2 Future work

Odersky used the Builder pattern extensively in the refactoring of the collections library. The revised library contains significantly less redundant code, while providing the client of the library with a more consistent interface. This experience strengthens our faith in type constructor polymorphism as an important tool for the library designer. However, it has also shown that there is still room for improvement. Using the library has not become more complicated, but designing it was quite challenging.

More concretely, two opportunities for improvement arose from our work on the collections library. First, even in the fragment of the library that was included in Chapter 3, it was clear that `BoundedIterable` required quite a bit of foresight in order to accommodate collections that require the type of their elements to be bounded.

Second, since variance annotations behave similarly to bounds with respect to kind checking, it is not surprising that they posed a similar challenge in the full library. More specifically, the core implementation classes, such as `IterableTemplate`, (conceptually) exist in a covariant and a nonvariant variety, even though the nonvariant version can be derived from the covariant one by simply dropping the appropriate '+' variance annotations. Thus, to remove this redundancy it should be possible to abstract over variance annotations, similar to how `BoundedIterable` abstracts over the bounds on the type of its elements.

Fortunately, as discussed in Chapter 3, both bounds and variance are captured by kinds, so that we could abstract over both of them by simply abstracting over the kind of the type of the elements. Thus, kind polymorphism emerges as a more powerful abstraction mechanism.

However, it seems that we keep discovering the need for abstracting over things that we formerly did not consider. First, methods allowed abstracting over values (their arguments), then, genericity was introduced, and types (parameterised classes) as well as values (polymorphic methods) could abstract over types. With type constructor polymorphism, types can abstract over type constructors, and kind polymorphism subjects kinds to abstraction. Can we skip another incremental improvement and achieve an increase by an *order of magnitude*?

Scalina lays the foundations for this exploration by aligning the levels of values and types as closely as possible. Although we can abstract over values and types in much the same way, the abstraction mechanisms are modelled by different relations, with subtly different properties.

Instead of incrementally introducing mechanisms that abstract over entities at increasingly high levels of abstraction, we conjecture that the core of the next generation of type systems should be based on a single abstraction mechanism that works at any level. We see level polymorphism as an important source of inspiration. Sheard is investigating it in

Ωmega [She08], a Haskell-like language with type-level computation and user-definable kinds. In the context of the calculus of constructions, this is called "universe polymorphism" [HP89]. The remainder of this section discusses our plans for combining these features into a scalable type system.

5.2.1 A scalable type system

The type system must be able to grow so that it can succinctly specify the correctness of the implementation. The same type system must not interfere during prototyping, gently ramping up its verification power while the implementation matures and thus requires more quality assurance. The required level of quality not only evolves over time, it also varies across the modules that make up a program, as some are less mission-critical than others. Nevertheless, the invariants of each constituent module must be maintained, even when an unverified module in the prototyping phase uses one of the highly verified core modules.

Thus, the type system must scale in its potential for detecting bugs – its expressiveness – as well as in how strictly it enforces correctness – its flexibility.

Flexibility

A type system's flexibility can be further decomposed into how strictly types are checked and when these checks are performed, how much missing or erroneous type information can be reconstructed, and whether the type system influences program execution. Another consideration is whether features that improve flexibility are part of the language. More concretely, a compiler that can be configured to treat type errors as warnings is an extra-lingual feature that improves type system flexibility.

Untyped and traditional statically typed languages form the extremes of type checking strictness, as they either perform no type checks at all, or insist on full type checking. Anderson and Drossopoulou introduce a permissive type [AD03] that may be used instead of more precise type information when more lenient type checking is preferred. Recently, Siek and Taha refined this idea and coined the term "gradual typing" [ST07]. An optional type system, such as Strongtalk's [BG93], can be seen as a binary version of a gradual type system, as it can be turned on and off, but types do not scale in precision.

Flexibility in *when* types are checked has recently been explored by Flanagan [Fla06], and he calls it "hybrid type checking". Soft typing [CF91] is a related technique, which turns static type errors into dynamic casts, thus deferring type checking to run time when necessary. In a limited sense, Eclipse's so-called "quickfix" feature makes it easier for the programmer to emulate soft typing for Java by automatically fixing static errors so that the program can be compiled. However, this gives a false impression of increasing programmer productivity, as the programmer is responsible for keeping track of these temporary fixes so that they can eventually be implemented properly.

A flexible type system should have an extra-lingual way of determining how type errors are handled, so that this policy can easily be adjusted to the required level of quality assurance for a certain module. Additionally, the language itself must incorporate types that correspond to varying degrees of strictness. Languages with subtyping already provide this to a certain extent, but Siek et al. point out that it does not suffice [ST07].

Clearly, type inference provides another important source of flexibility, which is closely related to type checking, especially when recovery from type errors is required. Full type inference is not realistic, or even desirable, except perhaps for experimentation, scripting, and prototyping. Types are an important form of documentation, and inference cannot be trusted to recover programmer intent. While local type inference is invaluable in lowering the burden of type annotation, full type inference undermines the verification power of the type system, as fully inferred specifications almost trivially correspond to the implementation.

The type system's guiding role is not limited to documentation, as precise types guide the implementation effort. In fact, types can be leveraged to generate missing implementation code in certain cases. Concrete examples can be found in the work on data-type generic programming [HJ03, HL06, MPJ06, Hin06], where code follows the shape of types. Ad-hoc polymorphism, whether it be in the guise of static overloading in Java, implicit arguments in Scala, or type classes in Haskell, is a more common example of the type system influencing program behaviour.

A more advanced application of reversing the direction of inference, so that values are derived from types, is found in many dependently-typed languages, but also in Djinn, a tool that generates a Haskell implementation for certain kinds of types. By the Curry-Howard isomorphism, this corresponds to automated theorem proving in the logic that corresponds to the type system.

As noted by Bracha [Bra04], a program's execution should not depend on the type system. More concretely, whether the program is allowed to execute, and what this execution comprises, should not depend on the type system. Otherwise, optional typing would not be feasible. It also implies that changing the type inference algorithm could change program behaviour, which would certainly surprise the programmer.

This stance should be nuanced somewhat, since the interaction between types and values also has several useful applications, as already discussed. A compromise could be achieved by interpreting type checking as an optimisation phase that turns dynamic type checks into static ones, discharging the proof obligations that correspond to these run-time test. In this interpretation, turning the type checker off only has an impact on the performance of the program, not its result.

Finally, an important open problem is how to deal with hybrid compositions of modules that have been type checked independently under different typing regimes. For example, while experimenting, some code that is not type checked at all may call into a core module that expresses its specifications as part of the types in its interface. How will these invariants be guaranteed? Clearly, the run-time system must mediate in a scenario like this. Recently, Wadler and Findler's have shown that "well-typed programs can't be blamed" [WF09], and Gray has shown how classes that are defined in a dynamically typed and a statically typed language can safely extend each other [Gra08].

Expressiveness

A type system such as Scala's is at the frontier of what is currently accepted by programmers in OO languages. It is very expressive, but its notion of dependent types is too weak to express specifications such as in JML or Spec$^{\#}$. Nonetheless, there exist programming languages whose type systems rival the expressiveness of these specification languages. For example, Epigram [McB04] has a dependent type system that closely corresponds to constructive type theory, which is also at the core of the Coq proof assistant [BC04].

Chen's work on ATS [CX05], and Sheard's Ωmega [She08] are some of the first indications that theorem proving is (slowly) entering the realm of practical programming languages. Recently, Nanevski et al. have taken another important step in improving the type system's expressiveness by integrating Hoare-style specifications [NMB08].

Fundamentally, a type system can only be truly expressive if it can "grow" [Ste99]. It should be possible to express and encapsulate new abstraction in the type system, as it will never be possible to develop one fixed type system that suits every application or application domain. Thus, the same principles for value-level language design should be applied to the type system.

The subtle difference between the value-level language and the programs written in it becomes even more important at the level of types. For example, Java's `equals` method can be seen as a hook into the language semantics, as it influences a core aspect of the language. Similarly, being able to "rewire" equality at the type level would have far-reaching consequences. Nonetheless, it is essential for the development of type system modules. For example, such an extension could enforce the consistent use of units of measurement [Ken94], which involves extending the notion of type equality to include the laws of the abelian group formed by units and the operations on them (e.g., $N * m$ is the same unit as $m * N$).

Chameleon [WSSR05, SS08] is an important example of a language with an extensible type system, at the value level it is a Haskell-like functional programming language, and the type system can be programmed using constraint handling rules [Frü94]. More recently, Schrijvers et al have proposed an approach in which functional programming is used at both levels [SJCS08]. We are not aware of a language in which object-oriented programming is similarly employed, even though subtyping and encapsulation could provide interesting benefits.

An important open problem in this area is how to modularly prove soundness of a language with an extensible type system. Another challenge is how to bring the ideal notions of theorems and proofs to the real world, that includes effects, from the seemingly innocuous problem of non-termination over the age-old `printf` and `malloc`, and subtle issues such as initialisation and variable binding, to more complex I/O such as network connections and database transactions. An interesting combination of the idea of a pluggable type system and the desire to tame effects using the type system can be found in recent work on representing uniqueness annotations as types of a kind that is distinct from the kind of types of classical expressions [dVPA07]. Even though the notion of a pluggable type system is only present in spirit in their approach, it would serve as an excellent, though truly ambitious, benchmark to challenge the power of an extensible type system.

Finally, we observe that flexibility and expressiveness are two sides of the same of coin in more than one way. On one hand, a type system can only be scalable if it has both properties; on the other hand, though, an interesting research topic is whether they can be realised in the same way. For example, could flexibility arise from varying how many type system modules are enabled, or whether the strict or lenient variant of a module is used?

5.3 Conclusion

Genericity is a proven technique to reduce code duplication in object-oriented libraries while making them easier to use by clients. The prime example is a collections library, where clients no longer need to cast the elements they retrieve from a generic collection.

Unfortunately, though genericity is extremely useful, the first-order variant is self-defeating in the sense that abstracting over proper types gives rise to type constructors, which cannot be abstracted over. Thus, by using genericity to reduce code duplication, other kinds of boilerplate arise. Type constructor polymorphism allows to further eliminate these redundancies, as it generalises genericity to type constructors. We implemented type constructor polymorphism in Scala 2.5.

As with genericity, most use cases for type constructor polymorphism arise in library design and implementation, where it provides more control over the interfaces that are exposed to clients, while reducing code duplication. Moreover, clients are not exposed to the complexity that is inherent to these advanced abstraction mechanisms. In fact, clients *benefit*, similarly to how genericity reduced the number of casts that clients of a collections library had to write.

Beside the collection libraries, the poster child for genericity, we have experimented with several other applications, such as embedded domain specific languages (DSL's) [CKcS07, HORM08]. Our parser combinator library [MPO08b] is essentially a DSL for writing EBNF grammars in Scala. Datatype-generic programming [Hin06] is another technique that benefits from Scala's support for type constructor polymorphism [MPJ06, Moo07].

While developing the theoretical aspect of type constructor polymorphism, we discovered that existing Scala calculi could not fully encode our extension. Thus, we set out to design Scalina, which provides better support for safe type-level abstraction. More specifically, Scalina is *kind sound*, which means that type-level computation (or abstraction) does not go wrong. To achieve this, we distinguish "input" and "output" members.

Beyond this original motivation, Scalina has evolved into a research vehicle for experimenting with language constructs in support of a scalable type system. From the outset, we strove for uniformity in the abstraction mechanisms at the level of terms and types. We now consider this uniformity a key ingredient for a scalable type system. Furthermore, uniformity in abstraction mechanisms across levels subsumes kind polymorphism, which our practical experience with type constructor polymorphism has shown to be useful. Thus, we plan to continue our research in this direction.

Bibliography

[AC95] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Sci. Comput. Program.*, 25(2-3):81–116, 1995.

[AC96] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Inf. Comput.*, 125(2):78–102, 1996.

[AC01] David Aspinall and Adriana B. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.

[AD03] Christopher Anderson and Sophia Drossopoulou. BabyJ: from object based to class based programming via types. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[Bar91] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BG93] Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

[BGP06] Colin Blundell, Dimitra Giannakopoulou, and Corina S. Pasareanu. Assume-guarantee testing. *ACM SIGSOFT Software Engineering Notes*, 31(2), 2006.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer Berlin / Heidelberg, January 2005.

[BMM90] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. *Inf. Comput.*, 85(1):76–134, 1990.

[BMS05] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In Eric Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549. Springer, 1998.

[Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[Bra07] Gilad Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In Walter G. Olthoff, editor, *ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 1995.

[CA08] Vincent Cremet and Philippe Altherr. Adding type constructor parameterization to Java. *Journal of Object Technology*, 7(5):25–65, June 2008. Special Issue: Workshop on FTfJP, ECOOP 07. http://www.jot.fm/issues/issue_2008_06/article2/.

[Car88a] Luca Cardelli. Structural subtyping and the notion of power type. In *POPL*, pages 70–79, 1988.

[Car88b] Luca Cardelli. Types for data-oriented languages. In Joachim W. Schmidt, Stefano Ceri, and Michele Missikoff, editors, *EDBT*, volume 303 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1988.

[CCH$^+$89] Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280, 1989.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[CG03] Adriana B. Compagnoni and Healfdene Goguen. Typed operational semantics for higher-order subtyping. *Inf. Comput.*, 184(2):242–297, 2003.

[CGLO06] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In Rastislav Kralovic and Pawel Urzyczyn, editors, *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CJSS07] Manuel Chakravarty, Simon L. Peyton Jones, Martin Sulzmann, and Tom Schrijvers. Class families, 2007. On the GHC Developer wiki, `http://hackage.haskell.org/trac/ghc/wiki/TypeFunctions/ClassFamilies`.

[CKcS07] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2007.

[CKJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. Associated types with class. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 1–13. ACM, 2005.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 66–77. ACM, 2005.

[DR08] Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In Hook and Thiemann [HT08], pages 307–320.

[dVPA07] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007.

[EKRY06] Burak Emir, Andrew Kennedy, Claudio V. Russo, and Dachuan Yu. Variance and generalized constraints for C$^{\#}$ generics. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2006.

[Ern99] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[Ern01] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.

[Ern06] Erik Ernst. Reconciling virtual classes with genericity. In David E. Lightfoot and Clemens A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2006.

[Ern07] Erik Ernst, editor. *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*. Springer, 2007.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, pages 171–183, 1998.

[Fla06] Cormac Flanagan. Hybrid type checking. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 245–256. ACM, 2006.

[Frü94] Thom W. Frühwirth. Constraint handling rules. In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 1994.

[Gir72] J.Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These d'Etat, Paris VII, 1972.

[GJ98] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *ICFP*, pages 273–279, 1998.

[GJS+06] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In Peri L. Tarr and William R. Cook, editors, *OOPSLA*, pages 291–310. ACM, 2006.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[Gra08] Kathryn E. Gray. Safe cross-language inheritance. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 52–75. Springer, 2008.

[HHJW07] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *HOPL*, pages 1–55. ACM, 2007.

[Hin06] Ralf Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5):451–483, 2006.

[HJ03] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Carl Backhouse and Jeremy Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003.

[HJW^{+}92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[HL06] Ralf Hinze and Andres Löh. "scrap your boilerplate" revolutions. In Tarmo Uustalu, editor, *MPC*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer, 2006.

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming (reprint). *Commun. ACM*, 26(1):53–56, 1983.

[HORM08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

[How69] W.A. Howard. To H.B. Curry: The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1969.

[HP89] Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions (draft). In Josep Díaz and Fernando Orejas, editors, *TAPSOFT, Vol.2*, volume 352 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 1989.

[HT08] James Hook and Peter Thiemann, editors. *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. ACM, 2008.

[Hug99] John Hughes. Restricted datatypes in Haskell. Technical Report UU-CS-1999-28, Department of Information and Computing Sciences, Utrecht University, 1999.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[Jon94] Mark P. Jones. constructor classes & "set" monad?, 1994. `http://groups.google.com/group/comp.lang.functional/msg/e10290b2511c65f0`.

[Jon95] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35, 1995.

[Ken94] Andrew Kennedy. Dimension types. In Donald Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994.

[Kid07] Eric Kidd. How to make data.set a monad, 2007. `http://www.randomhacks.net/articles/2007/03/15/data-set-monad-haskell-macros`.

[KT92] A. J. Kfoury and Jerzy Tiuryn. Type reconstruction in finite rank fragments of the second-order lambda-calculus. *Inf. Comput.*, 98(2):228–257, 1992.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LDG$^+$07] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, 2007. release 3.10.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[McB04] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[Mei06] Erik Meijer. There is no impedance mismatch: (language integrated query in Visual Basic 9). In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 710–711. ACM, 2006.

[Mei07] Erik Meijer. Confessions of a used programming language salesman. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 677–694. ACM, 2007.

[Moo07] Adriaan Moors. Code-follows-type programming in Scala. Manuscript available from `http://www.cs.kuleuven.be/~adriaan/?q=cft_intro`, 2007.

[MPJ06] Adriaan Moors, Frank Piessens, and Wouter Joosen. An object-oriented approach to datatype-generic programming. In Ralf Hinze, editor, *ICFP-WGP*, pages 96–106. ACM, 2006.

[MPO07] Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. Accepted for the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages at the European Conference on Object-Oriented Programming (ECOOP), 2007.

[MPO08a] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In Gail E. Harris, editor, *OOPSLA*, pages 423–438. ACM, 2008.

[MPO08b] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2008. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html`.

[MPO08c] Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in Scala. In *Proc. FOOL '08*, January 2008. `http://fool08.kuis.kyoto-u.ac.jp/`.

[NMB08] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[OAC$^+$06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.

[Ode06] Martin Odersky. Poor man's type classes, July 2006. Talk at IFIP WG 2.8, Boston.

[Ode07] Martin Odersky. *The Scala Language Specification, Version 2.6*. EPFL, November 2007. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[OH92] Harold Ossher and William H. Harrison. Combination of inheritance hierarchies. In *OOPSLA*, pages 25–40, 1992.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 41–57. ACM, 2005.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *POPL*, pages 41–53, 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PS97] Benjamin C. Pierce and Martin Steffen. Higher-order subtyping. *Theor. Comput. Sci.*, 176(1-2):235–282, 1997.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[Rey74] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[She07] Tim Sheard. Type-level computation using narrowing in Ωmega. *Electr. Notes Theor. Comput. Sci.*, 174(7):105–128, 2007.

[She08] Tim Sheard. *Ωmega Users' Guide.* Portland State University, 2008. `http://web.cecs.pdx.edu/~sheard/Omega/index.html`.

[SJCS08] Tom Schrijvers, Simon L. Peyton Jones, Manuel M. T. Chakravarty, and Martin Sulzmann. Type checking with open type functions. In Hook and Thiemann [HT08], pages 51–62.

[SS08] Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *J. Funct. Program.*, 18(2):251–283, 2008.

[SSW03a] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. The chameleon type debugger (tool demonstration). *CoRR*, cs.PL/0311023, 2003.

[SSW03b] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, New York, NY, USA, 2003. ACM.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In Ernst [Ern07], pages 2–27.

[Ste98] Martin Steffen. *Polarized Higher-Order Subtyping.* PhD thesis, Universität Erlangen-Nürnberg, 1998.

[Ste99] Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[Str67] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in

Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

[TS06] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[TT99] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In Rachid Guerraoui, editor, *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204. Springer, 1999.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, 1987.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[WF09] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, Lecture Notes in Computer Science. Springer, 2009. To appear.

[WLT07] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI : Generalized interfaces for Java. In Ernst [Ern07], pages 347–372.

[WSSR05] Jeremy Wazny, Martin Sulzmann, Peter J. Stuckey, and Andreas Rossberg. The Chameleon home page, 2005.

List of Publications

Conference Papers

[1] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *OOPSLA*, pages 423–438, 2008.

[2] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *GPCE*, pages 137–148, 2008.

Refereed Workshop Papers

[3] Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in Scala. In *Proc. FOOL '08*, January 2008. `http://fool08.kuis.kyoto-u.ac.jp/`.

[4] Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. Accepted for the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages at the European Conference on Object-Oriented Programming (ECOOP), 2007.

[5] Adriaan Moors, Frank Piessens, and Wouter Joosen. An object-oriented approach to datatype-generic programming. In *ICFP-WGP*, pages 96–106, 2006.

[6] Adriaan Moors, Jan Smans, Eddy Truyen, Frank Piessens, and Wouter Joosen. Safe language support for feature composition through feature-based dispatch, October 2005. In the informal proceedings of the OOPSLA workshop on Managing Variabilities Consistently in Design and Code.

Other Manuscripts

[7] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2008. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html.

[8] Adriaan Moors. Code-follows-type programming in Scala. Manuscript available from http://www.cs.kuleuven.be/~adriaan/?q=cft_intro, 2007.

[9] Adriaan Moors. Development of an Object Oriented Language that Integrates several State of the Art Concepts. Master's thesis, KU Leuven, 2004. In Dutch.

Biography

2008 Internship with Don Syme at Microsoft Research, Cambridge (3 months). Internship in the F# team. Among other things, implemented a meta-model for LINQ to allow seamless persistence of F# data structures.

2008 Served on the program committee of the Workshop on Generic Programming 2008

2008 Visited Klaus Ostermann at Aarhus University (2 weeks). Co-authored “Polymorphic Embedding of DSLs”.

2007 Internship with Martin Odersky at EPFL (3 months). Implemented support for higher-kinded types and higher-order subtyping in the Scala compiler. Included in the distribution as of version 2.5.

2006 Reviewer for the JFP special issue for MSFP 2006 (since then, sub-reviewed for AOSD, ICFP, and OOPSLA)

2006 – 2009 Personal 4-year research grant from the *Institute for the promotion of Innovation by Science and Technology* in Flanders (IWT-Flanders)

2004 – 2006 K.U.Leuven doctoral scholarship (15 months).

1999 – 2004 B.Sc.E & M.Sc.E. in Computer Science (Magna Cum Laude), K.U.Leuven.

1981 Crashed space ship on planet Earth. Oops.

Type constructor polymorfisme voor Scala: theorie en praktijk

A.1 Samenvatting

Een statisch type systeem is een belangrijk hulpmiddel bij het efficiënt ontwikkelen van correcte software. De recente invoering van “genericity” in object-gerichte programmeertalen heeft hun type systeem een stuk krachtiger gemaakt. Genericity, ook wel “parametrisch polymorfisme” genoemd, is zeer nuttig, omdat het bijvoorbeeld de definitie van polymorfe lijsten mogelijk maakt, waarbij een type parameter gebruikt wordt om abstractie te maken van het type van de elementen. Aangezien genericity echter beperkt is tot eerste orde, kunnen we niet opnieuw abstractie maken van het type van deze geparametriseerde lijsten.

We veralgemeenden Scala’s ondersteuning voor parametrisch polymorfisme tot de hogere orde, aangezien dit van praktisch nut blijkt te zijn. We noemen het resultaat “type constructor polymorfisme”, aangezien Scala programmeurs nu veilig abstractie kunnen maken van type constructors, zoals het type van polymorfe lijsten. We beschrijven de theoretische onderbouw en de praktische kant van onze uitbreiding van Scala’s type systeem.

Onze veralgemening, versterkt door de synergie met Scala’s bestaande concepten, zoals implicits, vormt een belangrijke troef voor de library-ontwerper, terwijl de gebruiker van deze abstracties zich geen zorgen hoeft te maken over hun interne werking. De theoretische kant van het verhaal richt zich op de lacunes in de bestaande Scala formalismes, en presenteert Scalina, onze calculus die deze oplost. Ten slotte gaan we kort in op onze visie voor toekomstige verbeteringen van het type systeem, op basis van onze ervaring met Scalina en type constructor polymorfisme.

A.2 Type constructor polymorfisme

A.2.1 Inleiding

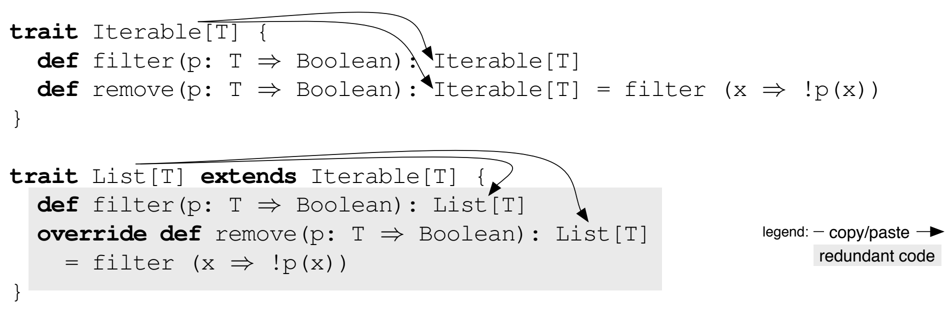
Eerste-orde parametrisch polymorfisme wordt ondertussen ondersteund door de meeste statisch getypeerde programmeertalen. Het concept vond zijn oorsprong in system F [Gir72, Rey74] en werd voor het eerst in de praktijk toegepast in functionele programmeertalen. Meer recent werd het geïntegreerd in object-gerichte talen zoals Java, C$^{\#}$, en vele andere. In de context van deze talen noemt men eerste-orde parametrisch polymorfisme doorgaans *generics*. Abadi en Cardelli [AC96, AC95], Igarashi et al. [IPW01], en vele anderen hebben deze concepten ten gronde bestudeerd.

Collecties zijn een standaard toepassing van generics. Het type `List[ A]`, bijvoorbeeld, stelt lijsten voor met elementen van het type `A`, dat vrij gekozen kan worden. In deze zin kunnen generics gezien worden als de veralgemening van het type van arrays, dat in Java altijd al geparameteriseerd is geweest in het type van zijn elementen.

Eerste orde parametrisch polymorfisme heeft echter een aantal beperkingen. Hoewel het toelaat om abstractie te maken van types, wat leidt tot *type constructors* zoals `List`, werkt het abstractie mechanisme niet meer voor deze type constructors. Hierdoor kan men een type constructor niet doorgeven als een type argument aan een andere type constructor. Abstracties vereisen dit echter vaak, zelfs in object-gericht programmeren, en deze beperking leidt dus tot onnodige code duplicatie.

De veralgemening van eerste orde polymorpfisme tot het hogere-orde systeem was een natuurlijke stap in de lambda calculus [Gir72, Rey74, BMM90]. Deze theoretische vooruitgang is sindsdien opgenomen in functie-gerichte programmeertalen. De programmeertaal Haskell [HJW^{+}92] ondersteunt type constructor polymorfisme, dat ook geïntegreerd is met het “type class” concept [Jon95]. Deze veralgemening naar types die abstractie maken van types die abstractie maken van types (“higher-kinded types”) heeft vele praktische toepassing, zoals comprehensions [Wad92], parser combinators [HM96, LM01], en meer recent werk rond embedded Domain Specific Languages (DSL’s) [CKcS07, HORM08].

Dezelfde noden en toepassingen bestaan in object-gericht programmeren. LINQ bracht rechtstreekse ondersteuning voor for comprehensions naar het .NET platform [BMS05, Mei07], Scala [OAC^{+}06] heeft altijd een dergelijke constructie ondersteund, en Java 5 introduceerde een lichtgewicht versie [GJSB05, Sec. 14.14.2]. Parser combinators zijn eveneens meer populair aan worden: Bracha gebruikt hen als de onderliggende technologie voor zijn Executable Grammars [Bra07], en de Scala distributie bevat a li-



```
trait Iterable[T] {
  def filter(p: T ⇒ Boolean): Iterable[T]
  def remove(p: T ⇒ Boolean): Iterable[T] = filter (x ⇒ !p(x))
}

trait List[T] extends Iterable[T] {
  def filter(p: T ⇒ Boolean): List[T]
  override def remove(p: T ⇒ Boolean): List[T]
    = filter (x ⇒ !p(x))
}
```

Listing A.1: De beperkingen van genericity

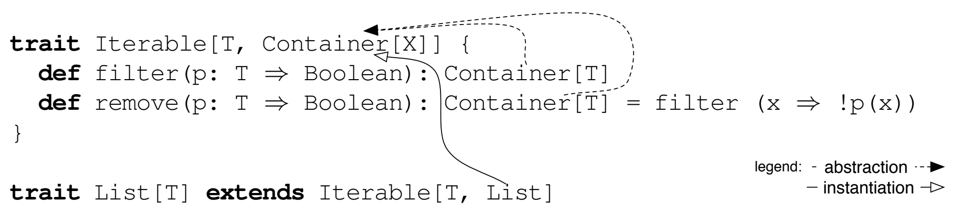
brary [MPO08b] die een embedded DSL voor parsing implementeert, zodat gebruikers parsers rechtstreeks in Scala uit kunnen drukken, in een notatie die dicht bij EBNF aanleunt. Type constructor polymorfisme is essentieel voor het definiëren van een gemeenschappelijk parser interface dat door verschillende back-ends geïmplementeerd wordt.

A.2.2 Code duplicatie verminderen met type constructor polymorfisme

Deze paragraaf illustreert de voordelen van het veralgemenen van genericity tot type constructor polymorfisme aan de hand van de gekende `Iterable` abstractie. Het eerste voorbeeld, gesuggereerd door Lex Spoon, illustreert de essentie van het problem. Paragraaf 3.2.1 breidt het uit tot realistischere proporties.

Listing A.1 toont een Scala implementatie van de `Iterable[T]` trait, die bestaat uit een abstracte `filter` methode en een concrete `remove` methode. Subklassen horen `filter` te implementeren zodat het een nieuwe collectie afleidt van de huidige door enkel de elementen te behouden die voldoen aan een gegeven predicaat `p`. Dit predicaat wordt voorgesteld als een functie die een element van de collectie neemt, van het type `T`, en die een `Boolean` teruggeeft. Aangezien `remove` enkel de betekenis van dit predicaat inverteert, is deze methode geïmplementeerd in termen van `filter`.

Als men een lijst filtert, verwacht men vanzelfsprekend opnieuw een lijst. Vandaar overschrijft `List` de `filter` methode en verfijnt hij covariant het type van het resultaat. `remove` hoort uiteraard hetzelfde type resultaat te hebben, maar de enige manier om dit te bekomen is om de volledige



```
trait Iterable[T, Container[X]] {
  def filter(p: T ⇒ Boolean): Container[T]
  def remove(p: T ⇒ Boolean): Container[T] = filter (x ⇒ !p(x))
}

trait List[T] extends Iterable[T, List]
```

Listing A.2: Code duplicatie verwijderen

methode te hernemen. De code duplicatie die hieruit volgt is een duidelijke aanwijzing van een beperking van het type systeem: beide methodes zijn redundant in `List`, maar het type systeem is niet krachtig genoeg om deze abstractie uit te drukken.

De oplossing die we voorstellen in Listing A.2 is om abstractie te maken van de type constructor die de container voorstelt van het resultaat van `filter` en `remove`. De verbeterde `Iterable` neemt nu twee type parameters: de eerste, `T`, staat voor het type van zijn elementen, en de tweede, `Container`, staat voor de *type constructor* van de container van het resultaat van `filter` en `remove`. `Container` is een type parameter die zelf een type parameter verwacht. Hoewel de naam van deze hogere-orde type parameter (`X`) hier niet gebruikt wordt, zullen meer gesofisticeerde voorbeelden het nut van expliciete naamgeving aantonen.

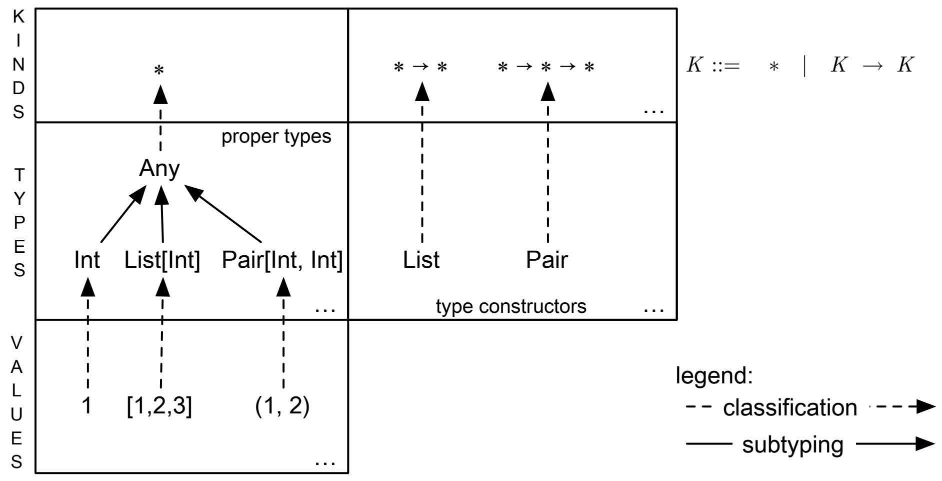
Nu, om aan te geven dat het toepassen van `filter` of `remove` op een `List[T]` een `List[T]` teruggeeft, instantieert `List` `Iterable`'s type constructor parameter als de `List` type constructor.

In dit eenvoudige voorbeeld had men ook gebruik kunnen maken van een constructie als Bruce's `MyType` [BSvG95]. Dit veralgemeent echter niet naar meer ingewikkelde gevallen.

A.2.3 Kinds

Hoewel gewone types en type constructors op gelijke voet geplaatst worden wat betreft parametrisch polymorfisme, moeten ze toch onderscheiden worden. Een type parameter die staat voor een gewoon type kan geen argumenten ontvangen, terwijl een type constructor parameter op zich dan weer geen waardes kan classificeren. Enkel wanneer een dergelijke type parameter de nodige type argumenten heeft ontvangen, en dus een gewoon type is geworden, kan er gesproken worden van instanties van dit type.

We gebruiken "kinds" om een type parameter die staat voor een ge-



Figuur A.1: Diagram van levels

woon type, zoals `List[Int]`, te onderscheiden van een type parameter die abstractie maakt van een type constructor, zoals `List`. Het diagram in Fig. A.1 illustreert een eerste, simplistische, kind systeem, dat we verder zullen verfijnen. De figuur toont de drie "levels" van classificatie, waar entiteiten in lagere levels geclassificeerd worden door entiteiten in de laag er net boven.

Kinds bevinden zich in de bovenste laag. Het kind $\star$ classificeert types die waarden classificeren, en de $\rightarrow$ kind constructor wordt gebruikt om kinds te construeren die type constructors classificeren. Merk op dat de compiler deze kinds infereert: de programmeur wordt hier volledig van afgeschermd.

We verfijnen het kind van type constructors tot een *dependent* function kind, aangezien hogere-orde type parameters in hun bounds mogen voorkomen, of in de bounds van de omsluitende type parameter. Het kind dat gewone types classificeert wordt veralgemeend tot het kind $\star$`(T, U)`, dat het subtyping interval aangeeft van de types die het classificeert: het omvat alle types die een subtype zijn van `U` en een supertype van `T`. Ten slotte moet het function kind verder uitgebreid worden om correct om te gaan met variance annotaties. We gaan hier in deze samenvatting niet verder op in.

A.3 Veilige abstracties op het niveau van types

A.3.1 Inleiding

Scalina is a strikt object-gerichte calculus die onze uitbreiding van Scala met type constructor polymorfisme formeel onderbouwt. Scalina introduceert een aantal nieuwigheden ten opzichte van eerdere object-gerichte calculi [IPW01, OCRZ03, CGLO06]. De belangrijkste verbetering tov. de νObj calculus is dat kind checking er voor zorgt dat type applicatie nooit mis loopt: we noemen deze eigenschap *kind soundness*.

Traditioneel gebruikt de formele onderbouw van object-gerichte talen een combinatie van functionele (FP) en object-gerichte (OO) abstractie mechanismes. De FP stijl is gebaseerd op lambda abstractie and functie applicatie, en OO stijl abstracties worden uitgedrukt mbv. abstracte members en compositie (via subclassing of mixin composition).

Java, bijvoorbeeld, gebruikt FP abstractie voor methodes en polymorfe klassen, die parametrisch kunnen zijn in types en waarden. Java ondersteunt natuurlijk ook OO stijl abstractie: een klasse met een abstracte methode maakt abstractie van de implementatie van deze methode. Een subklasse wordt geacht om de concrete implementatie te voorzien.

Net als νObj maakt Scalina enkel gebruik van OO abstractie: er zijn geen constructies voor parameterisatie. Niettemin zullen we aantonen dat Scalina dezelfde abstracties als, bijvoorbeeld, system F_{ω}^{sub} [Car88a, PS97, CG03], uit te drukken, met dezelfde garanties.

A.3.2 Kind soundness

Scala ondersteunt twee soorten abstractie mechanismen: de FP stijl gebruikt parameterisatie, terwijl abstracte members de OO manier mogelijk maken. Een voor de hand liggende vraag is of een enkele stijl volstaat. Aangezien Scala intrinsiek object-gericht is, zullen we ons concentreren op de OO stijl, en hoe die de FP stijl kan encoderen.

Scala's abstract type members closely correspond to type parameters, and abstract type member refinement can be seen as het object-gerichte counterpart of type application. Abstract type member refinement is a restricted form of mixin composition that can be used to override abstract type members with concrete ones. However, it turns out that this encoding does not preserve het safety properties that are ensured by parameterisation.

Om dit meer concreet te maken, gebruikt Listing 4.1 parameterisatie om de gekende `Iterable` abstractie uit te drukken in Scala. De `Iterable`

```
trait Iterable[A, Container[X]] {
  def map[B](f: A ⇒ B): Container[B]
}

trait List[A] extends Iterable[A, List]
```

Listing A.3: `Iterable` uitgedrukt mbv. type parameters

```
trait TypeFunction1 { type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container{type A = B}
}

trait List extends Iterable { type Container = List }
```

Listing A.4: `Iterable`'s type parameters encoderen als members

trait (een abstract klasse) neemt twee type parameters: de eerste stelt het type van elementen voor, en de tweede maakt abstractie van de type constructor van de container.

Listing 4.2 illustreert de object-gerichte aanpak. `Iterable` abstraheert van het type van zijn elementen en de container ahv. abstracte members. De `A` type member wordt geërfd van `TypeFunction1`, en de `Container` type constructor parameter wordt een abstracte type member die begrensd is tot een subtype van `TypeFunction1`. `map`'s resultaat wordt uitgedrukt door `Container`'s abstracte type member `A` te verfijnen tot `B`.

Voorlopig bleef de encodering getrouw aan het origineel. Er treedt echter een discrepantie op wanneer we een ongeldig programma encoderen. De type applicatie `Iterable[A, NumericList]` van Listing 4.3 wordt — terecht — verboden door de compiler, terwijl zal blijken dat de encodering zonder waarschuwing wordt aanvaard. Hierdoor wordt het echter mogelijk om later in `map`'s resultaat type gelijk welk type `B` door te geven aan

```
trait NumericList[A <: Number]
    extends Iterable[A, NumericList]
```

Listing A.5: `NumericList`: een ongeldige subklasse van `Iterable`

```
trait NumericList extends Iterable {
  type A <: Number
  type Container = NumericList // Incorrect, maar geen
    foutmelding!
}
```
Listing A.6: De foutieve encoding van `NumericList` ontsnapt de type checker

`NumericList`, terwijl het enkel subtypes van `Number` aanvaardt. Door `Iterable[A, NumericList]` te verbieden zorgen we ervoor dat deze fout nooit kan optreden.

Deze vroege foutopsporing gaat verloren in de vertaling. De encodering verliest de “kind soundness” eigenschap. Dit wordt geïllustreerd door Listing 4.4, wat een geldig Scala programma is. De compiler aanvaardt dit programma, hoewel we de implementatie nooit zullen kunnen vervolledigen (uiteindelijk zullen we een `NumericList` moeten instantiëren voor een arbitrair type van elementen, waarop de compiler onze vergissing alsnog zal detecteren). Ter vergelijking met type soundness, het equivalent op het niveau van waardes van deze vergetelheid zou zijn om toe te laten om een functie van het type, e.g., `Number` ⇒ **`Any`** door te geven wanneer een **`Any`** ⇒ **`Any`** wordt verwacht.

Deze kind unsoundness vindt zijn oorsprong in de νObj calculus [OCRZ03], die toelaat om abstracte type members *covariant* te overschrijven, waardoor `NumericList <: TypeFunction1`, zodat de encodering van de ongeldige type toepassing resulteert in een geldig programma.

We maken vroege foutopsporing mogelijk in Scalina door covariante en contravariante members te onderscheiden, in plaats van te veronderstellen dat ze zich all covariant gedragen. Dit komt overeen met de observatie dat sommige members abstractie maken van invoer, terwijl andere de uitvoer, het resultaat, van de abstractie voorstellen. Invoer hoort zich contravariant te gedragen, zoals het type van de argumenten van een functie, terwijl covariantie gepast is voor uitvoer, wat overeenkomt met het resultaattype van een functie. Dankzij dit onderscheid kan een strikt object-gerichte calculus de FP stijl van abstractie encoderen met dezelfde garanties.

Vanuit het standpunt van de gebruikers van een abstractie, onderscheiden we externe en interne klanten. Externe klanten bieden informatie aan aan een abstractie zonder te weten met welke subtype van deze abstractie ze te maken hebben. De beperkingen op deze ontbrekende informatie mogen dus enkel verzwakt worden in de subtypes die de abstractie imple-

```
trait TypeFunction1 { deferred type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container◁{type A = B}
}

trait List extends Iterable { type Container = List }

trait NumericList extends Iterable {
  deferred type A <: Number // ongeldig: covariante
    verandering niet toegestaan
  type Container = NumericList
}
```

Listing A.7: Kind soundness herstellen via un-members

menteren. Interne klanten, die het exacte type kennen van de abstractie die ze verfijnen, kunnen zich permitteren om de beperkingen op het resultaat te verstrengen.

Scalina vervolledigt Scala's covariante type members met een contravariant versie, die we "un-members" noemen. Listing 4.5 geeft een pseudo-Scala voorstelling van de encodering weer, waar un-members aangegeven worden met het `deferred` keyword. Ze worden concreet gemaakt door externe gebruikers met de `... ◁{ ... }` constructie.

Aangezien de `A` type member invoer is, moet het zich contravariant gedragen, zodat het `NumericList` niet toegestaan is om de bound op de `A` un-member die het erfde van `Iterable`, te verstrengen.

A.4 Conclusie

Genericity is een bewezen techniek om code duplicatie in object-gerichte bibliotheken tegen te gaan, en ze tegelijkertijd gebruiksvriendelijker te maken. Het belangrijkste voorbeeld is een library van collecties, waar gebruikers de elementen die zij uit een generische collectie halen, niet meer hoeven te casten, zoals wel het geval is voor niet-generische collecties.

Helaas, hoewel genericity uiterst nuttig is, is de eerste orde variant te beperkt, in de zin dat abstraheren over gewone types leidt tot type constructors, waarvan geen abstractie gemaakt kan worden. Door gebruik te maken van genericity om code duplicatie te verminderen, ontstaan zo andere soorten redundantie. Type constructor polymorfisme laat toe om deze verder te elimineren, omdat zij genericity veralgemeent tot type constructors. We implementeerden type constructor polymorfisme in Scala 2.5.

Zoals met genericity, is type constructor polymorfisme vooral nuttig voor het ontwerpen en implementeren van libraries. De ontwerper kan rijkere abstracties definiëren en zo code duplicatie vermijden. Bovendien worden gebruikers niet blootgesteld aan de complexiteit die inherent is aan deze geavanceerde abstractie mechanismen. In tegendeel, type constructor polymorfisme maakt libraries meer gebruiksvriendelijk, net als genericity ervoor zorgde dat gebruikers van een collectie library minder casts moesten schrijven.

Naast libraries van collecties, hebben we geëxperimenteerd met verschillende andere toepassingen, zoals embedded domein specifieke talen (DSL's) [CKcS07, HORM08]. Onze parser combinator library [MPO08b] is een concreet voorbeeld van een DSL voor het schrijven van EBNF grammatica's in Scala. Datatype-generisch programmeren [Hin06] is een andere techniek die afhangt van Scala's ondersteuning voor type constructor polymorfisme [MPJ06, Moo07].

Tijdens de ontwikkeling van de theoretische onderbouw van type constructor polymorfisme, ontdekten we dat de bestaande Scala calculi zich niet leenden tot het encoderen van onze extensie. Daarom ontwierpen we Scalina, die betere ondersteuning biedt voor veilige type-niveau abstractie. Meer specifiek, is Scalina *kind sound*, wat betekent dat type-niveau computatie niet mis kan gaan. Om dit te bereiken voerden we, we onderscheid in tussen "invoer" en "uitvoer" members.

Afgezien van deze oorspronkelijke motivatie, heeft Scalina zich ontwikkeld tot een onderzoeksvehikel voor het experimenteren met taalconstructies ter ondersteuning van een schaalbaar type systeem. Vanaf het begin hebben we gestreefd naar uniformiteit in de abstractie mechanismen op het niveau van waarden en types. We beschouwen deze uniformiteit als een belangrijk ingrediënt van een schaalbaar type systeem. Bovendien kan kind polymorfisme, wat uit onze praktijkervaring met type constructor polymorfisme zeer wenselijk blijkt te zijn, gezien worden als een specialisatie van een uniform abstractie mechanisme over alle niveaus. Uit deze observaties concluderen we dat het zinvol is om ons onderzoek naar een uniform abstractie mechanisme verder te zetten.