OBJECT-ORIENTED PATTERN MATCHING

THÈSE N^{O} 3899 (2007)

PRÉSENTÉE LE 26 OCTOBRE 2007

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Burak EMIR

Dipl.-Inform., RWTH, Aachen, Allemagne
et de nationalité allemande

acceptée sur proposition du jury:

Prof. M. A. Shokrollahi, président du jury
Prof. M. Odersky, directeur de thèse
Prof. V. Kuncak, rapporteur
Dr D. Syme, rapporteur
Dr M. Zenger, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2007

Object-Oriented Pattern Matching

Burak EMIR

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Martin Odersky, for giving me support and advice all these years and for providing the excellent research environment that is the Programming Methods Group. Discussion with past and present colleagues was always enriching and provided ample food for thought, so I would like to thank them as well. This includes Prof. Uwe Nestmann and his staff, whose quest for formalizing properties of programming languages and calculi I always found interesting and inspiring.

I am also grateful for the encouragement I received in discussions with Andrew Kennedy, Claudio Russo, Andrej Rybalchenko, Jan Vitek, Vijay Saraswat, Doug Lea and Don Syme. There were occasions where, each one of them, in their own way, let me feel that my limited experience was valuable, my personal views were informed and interesting, or that they had no doubts that I would be successful in completing my thesis (there were many others, too, to whom I apologize for not mentioning them here). There are times when a grad students hangs on to this form of trust and recognition. Maybe it is this feeling of being trusted that allows one to accept criticism without taking it personal.

Throughout my doctoral studies, I received financial support by the Hasler Foundation, the Swiss National Science Foundation and the PalCom research project funded by the European Union. I should also mention my internship at Microsoft Research Limited, Cambridge. My time there did not only provide me the change of perspective that working in an equally excellent and stimulating research environment brought about, but also some insights that turned out to be directly relevant to this thesis.

Finally, I could never thank my parents enough for their support. This book is dedicated to them.

Abstract

Pattern matching is a programming language construct considered essential in functional programming. Its purpose is to inspect and decompose data. Instead, object-oriented programming languages do not have a dedicated construct for this purpose. A possible reason for this is that pattern matching is useful when data is defined separately from operations on the data - a scenario that clashes with the object-oriented motto of grouping data and operations. However, programmers are frequently confronted with situations where there is no alternative to expressing data and operations separately – because most data is neither stored in nor does it originate from an object-oriented context.

Consequently, object-oriented programmers, too, are in need for elegant and concise solutions to the problem of decomposing data. To this end, we propose a built-in pattern matching construct compatible with object-oriented programming. We claim that it leads to more concise and readable code than standard object-oriented approaches. A pattern in our approach is any computable way of testing and deconstructing an object and binding relevant parts to local names.

We introduce pattern matching in two variants, case classes and extractors. We compare the readability, extensibility and performance of built-in pattern matching in these two variants with standard decomposition techniques. It turns out that standard object-oriented approaches to decomposing data are not extensible. Case classes, which have been studied before, require a low notational overhead, but expose their representation, making them hard to change later. The novel extractor mechanism offers loose coupling and extensibility, but comes with a performance overhead.

We present a formalization of object-oriented pattern matching with extractors. This is done by giving definitions and proving standard properties for a calculus that provides pattern matching as described before. We then give a formal, optimizing translation from the calculus including pattern matching to its fragment without pattern matching, and prove it correct.

Finally, we consider non-obvious interactions between the pattern matching and parametric polymorphism. We review the technique of generalized algebraic data types from functional programming, and show how it can be carried over to the object-oriented style. The main

tool is the extension of the type system with subtype constraints, which leads to a very expressive metatheory. Through this theory, we are able to express patterns that operate on existentially quantified types purely by universally quantified extractors.

Keywords: programming language, type systems, pattern matching, object-oriented programming, data abstraction, semantics, compiler construction

Zusammenfassung

Musterabgleich ist ein essentielles Programmiersprachenkonstrukt der funktionalen Programmierung, welches der Dateninspektion und -dekomposition dient. Im Gegensatz hierzu fehlen objekt-orientierten Programmiersprachen derartige Konstrukte völlig. Ein möglicher Grund hierfür mag in der Tatsache liegen, dass Musterabgleich eine Trennung von Daten und Datenoperationen voraussetzt, welche jedoch nach einem objekt-orientierten Grundsatz gebündelt sein sollten. Trotz dieses Grundsatzes finden sich Programmierer häufig in Situationen wieder, welche das getrennte Definieren von Daten und Datenoperationen erfordern – schliesslich werden die meisten Daten weder in objekt-orientierter Weise gespeichert noch in objekt-orientierten Kontexten erzeugt.

Wir schliessen daraus, dass auch objekt-orientierte Programmierer elegante und konzise Lösungen für das Problem der Datendekomposition benötigen. Zu diesem Zweck definieren wir den Musterabgleich als ein eingebautes Sprachkonstrukt, dass sich mit objekt-orientierter Programmierung verträgt. Wir belegen, dass dies gegenüber den mit Standardtechniken erzielbaren Ergebnissen zu kürzerem und lesbareren Programmcode führt. Ein Muster ist in unserem Ansatz jede berechenbare Methode um Tests und Dekonstruktion auf einem Objekt durchzuführen und relevante Teile an lokale Namen zu binden.

Wir definieren Musterabgleich in zwei Varianten, mittels Fallklassen und mittels Extraktoren. Wir vergleichen beide Varianten mit Standardtechniken in Bezug auf Lesbarkeit, Erweiterbarkeit und Performanz. Es stellt sich heraus, dass Standardtechniken nicht erweiterbar sind. Fallklassen, die schon in der Literatur behandelt wurden, brauchen nur wenig Notation, legen jedoch ihre Darstellung frei und können daher nicht einfach geändert werden. Der neue Extraktormechanismus bietet gute Erweiterbarkeit, führt jedoch zu Performanzeinbussen.

Wir formalisieren objekt-orientierten Musterabgleich mit Extraktoren. Dazu liefern wir Definitionen und Beweise von Standardeigenschaften eines objekt-orientierten Programmiersprachenkalküls, welcher das oben beschriebene Konstrukt des objekt-orientierten Musterabgleichs enthält. Dann geben wir eine formale, optimierende Übersetzung von der Sprache inklusive Musterabgleich auf ihr Fragment ohne Musterabgleich an und beweisen ihre Korrektheit.

Schliesslich wenden wir uns den nicht-trivialen Wechselwirkungen zwischen Musterabgleich und parametrischem Polymorphismus zu. Wir geben eine Übersicht zur Technik der generalisierten algebraischen Datentypen aus der funktionalen Programmierung, und zeigen, wie sie auf den objekt-orientierten Stil übertragen werden kann. Das Hauptwerkzeug dabei wird die Erweiterung des Typsystems mit *subtype constraints* sein, welche zu einer ausdrucksstarken Metatheorie führt. Durch diese Theorie sind wir in der Lage, Muster auf existenziell quantifizierte Typen durch universell quantifizierte Extraktoren auszudrücken.

Stichwörter: Programmiersprachen, Typsysteme, Musterabgleich, objekt-orientiertes Programmieren, Datenabstraktion, Semantik, Übersetzerbau

Contents

List of Figures

Chapter 1

Introduction

Object-oriented programming is a style of software development that lacks a primitive for decomposition. Functional programming languages traditionally offer a pattern matching construct for this purpose.

In this thesis, we propose a pattern matching construct that is compatible with object-oriented programming practices. To this end, we study two approaches based on *case classes* and *extractors*. We claim that pattern matching increases the conciseness and readability of program code and that it can improve type-checking so that more programs can be type-checked. Furthermore, we show that there are situations where these benefits can be obtained efficiently (using case classes) or with improved extensibility characteristics (using extractors).

With the aim of motivating and supplementing the above summary, this introduction is structured as follows. After a brief look at functional pattern matching (Section 1.1), we take a look at the incompatibilities with object-oriented programming (Section 1.2). Following that, we describe the line of work aimed at overcoming these problems (Section 1.3). We then make a case for a pattern matching construct compatible with object-oriented programming (Section 1.4). We conclude with the overview (Section 1.5) and a list of claimed contributions (Section 1.6).

1.1 Functional Pattern Matching

In functional programming languages, pattern matching has been closely related to algebraic data types since its beginning – Burstall [13] is the first to define a pattern matching construct that resembles the one found in statically typed functional languages today.

The pattern matching construct is a high-level operation that performs a multi-way branch based on a case distinction of one or several input values [64, 47]. It is similar to the well-known switch statement, but more general: its branches do not test constants, but *patterns* -

structured terms that contain variables. Performing a match operation consists of checking whether the structure of an input value corresponds to the structure of a pattern. If it does, the pattern variables are bound to their corresponding parts in the input value and the relevant branch is executed in the an environment that is enriched with the pattern variable bindings.

What follows is an example of an algebraic data type. Here and in the following, we use HASKELL [47] for functional programming examples, and mark them with a comment `{-Haskell-}` in the top-right corner in order to distinguish them from object-oriented code.

```
data Expr = Num Int | Add Expr Expr                    {-Haskell-}

eval e = case e of
  (Num i)   -> i
  (Add c d) -> (eval c)+(eval d)
```

These lines define the type `Expr` for simple arithmetic expressions consisting of either a number or a sum. It is defined recursively through its so-called constructors. Instances of an algebraic data type are written as the constructor name together with its arguments, so `(Num 3)` will construct the obvious instance of `Num` whose first argument is `3`. Every instance of `Expr` is either `(Num` i`)` with i an integer, or `(Add` d e`)` with d, e being subexpressions.

Furthermore, a function `eval` is defined to evaluate such an expression. This happens by matching on its argument `e`. Since we know that `e` can only be one of either `(Num` i`)` or `(Add` c d`)`, we make a case distinction:

- if `e = (Num` j`)` for some j, the integer j is returned.
- if `e = (Add` c d`)` for some c, d, then recursive calls are effected on the subexpressions c, d, and the sum of these results is returned.

The case distinction above indicates what is meant by discerning data and deconstructing it into its subcomponents. This style of programming supports *equational reasoning* and is very concise and readable. In fact, it is hard to envision an equally concise but different way to define the data and the operation. The conciseness is also partly due to the fact that *type inference* allows us to omit easily derivable type information from the program source.

However, algebraic data types conflict with object-oriented design principles.

1.2 Why Algebraic Data Types are not Object-Oriented

Standard functional pattern matching depends on algebraic data types. Since algebraic data types suffer from several restrictions that make them unsuitable for an object-oriented pro-

gramming language, pattern matching was never accepted in the object-oriented community. Some arguments held agains pattern matching (see e.g. [67]) are:

Lacking extensibility Algebraic data types cannot be extended with new variants.

Lacking of representation independence Algebraic data types expose their representation, which makes it hard to change them later without affecting the rest of the program. Whereas getter methods can be used (and, if needed, redefined) in place of field access on normal objects, for algebraic data types there is no such option.

Too many ways to define data. Algebraic data types are intended to hold data, but in an object-oriented programming language this task is already fulfilled by classes. Adding non-orthogonal constructs to a programming language make it bulky, hard to implement and hard to learn.

No subtyping Algebraic data types cannot be organized in class hierarchies like normal classes. In particular, a constructor cannot "inherit" from an algebraic data type, no methods can be defined for a particular variant.

These counter-arguments all revolve around the problem of data definition, not the pattern matching construct *per se*. More precisely, object-oriented programmers claim the importance of grouping data and operations in *classes* over other benefits.

The answer seems obvious: turn algebraic data types into classes.

1.3 A Brief History of Case Classes

Adapting algebraic data types to the object-oriented context has been initiated by Wadler and Odersky's PIZZA extension to the JAVA programming language [70]. This language offers generics, closures and also algebraic data types and pattern matching. This extension was the first version of case classes: within the scope of class B, algebraic data type constructors K_i could be defined writing a constructor signature **case**$K_i(T_1 f_1, \ldots, T_n f_n)$. The compiler lifted these, turning them to full classes that extended the containing class B. Thus, classes K_i existed that inherited methods from B. The compiler also recognized calls to the constructor that were not preceded by the **new** keyword.

In his diploma thesis, Zenger describes an extensible compiler framework [96] which revolves around the idea of extensible algebraic data types. He describes an iterative algorithm that represents a match expression, which differed significantly from the existing pattern match translation algorithm in the literature and the first PIZZA implementation [93]. He goes on to show that extensible algebraic datatypes can be applied to solve the problem of simultaneously extending algebraic datatypes and operations defined on these types [97].

In the SCALA programming language [69], case classes turned into classes that did not need to live in the scope of an enclosing class. The **case** has become merely a modifier that can turn any class into a case class – with the sole restriction that case classes could not inherit from case classes.

This restriction was motivated by the implementation: at the time all these systems were designed, the JVM did not have the same aggressive optimizations that they have now. So the designers did not want to commit to slow $instanceof$ checks and thus restricted case classes such that they could not have a direct or indirect parent that is also a case class. This way, a pattern match could always be optimized using integer tags (Chapter 4) takes up the idea or replacing type tests with integer tags.

The present work complements these efforts by recasting them in terms of *extractors* which are user-defined patterns that are referenced in pattern like method calls. The author makes no claims regarding case classes, except the implementation of incompleteness checking for sealed types (see Subsection 1.4.2) and the study of their characteristics in comparison with other techniques. The novel technique of extractors does away with the tight coupling between types and matching behavior. While we will see that the added flexibility comes at a price in terms of performance, it is not so easy anymore to dismiss pattern matching as not extensible and exposing representation – since programmers have the choice and can even combine two different approaches to data definition, a pattern matching construct seems to have become considerably more attractive. In the next section, we will analyze further reasons to study and use pattern matching of a different nature.

1.4 Data in the Object-Oriented Approach

What is the object-oriented way of structuring and operating on data? Object-oriented design is a method and style for software system development that amounts to encapsulating data and operations in abstractions. The abstractions produced by this method are usually implemented in an object-oriented programming language, which supports the object-oriented style through its syntax and semantics. It does so by providing an object model and by offering built-in constructs for object construction, method calls, field access and possibly deletion of objects.

This works great, as long as data and operations can really be grouped together. Classifying abstractions and taking them apart into their subcomponents is *not* a built-in construct of object-oriented programming languages. Yet the problem of defining data separate from operations of course arises, and demands solutions.

In fact, rather than being a rare programming situation, defining data separately from operations is frequent in various applications. For instancem, in his compiler textbook Appel [6, pp.94] contrasts compilers with graphic user interface toolkits, observing two orthogonal directions of modularity: both applications have a matrix of data and operations, but whereas for compilers, the data (syntax trees) is seldom changed but operations (compiler passes) are evolving, for a user interface toolkit, the operations (Redisplay, Move, ...) are fixed and the data (widgets) are unknown. Compilers need to separate operations from the classes that represent syntax trees, since it is inconvenient to change every syntax tree class when a single operation is added. In contrast, graphical user interface toolkits blend well with object-oriented style, since every widget can be implemented as a new class that will implement the interface that contains all the operations it has to support.

The situation would be very frustrating if compiler writers had to embrace the object-oriented style without having a choice to resort to a data-separate approach. There are other situations that are not application specific. Development that spans several organizational units can mean that changes to data classes are not allowed, yet it may be necessary to add an operation.

Maybe the most compelling reason to embrace data-separate-from-operations is the origin of data. The advent of semistructured data in XML and JSON notation on the internet and the persistence of relational database technology are two important indicators that suggest that data does not originate and is not stored in an object-oriented context. While data binding, object-relational mappings and transparent persistence all aim to create the convenient illusion for the programmer that data is accessible in the form of object-oriented abstractions, practical problems prevail. Often, the easiest way is to leave external data as it is and treat it through particular APIs. For instance, XML data is commonly manipulated through the Document Object Model (DOM) API. In these cases, data is already separate and only an API is available to access or traverse it.

1.4.1 A pattern matching use case

There seems to be ample need for solutions that define operations outside of the classes, separately from the data. Let us single out one example and demonstrate how standard object-oriented solutions compare to built-in pattern matching.

Imagine a distributed system where a components communicate via messages. In order to

keep the system extensible, the kinds of possible messages are not fixed in advance. The following classes (in SCALA syntax) model the relevant aspects of messages and components.

```
abstract class Msg(sender:ComponentId)

abstract class Component { ...
  def getId: ComponentId
  def handle(msg:Msg): Unit
}
```

Suppose we wanted to introduce two kinds of messages, one for requests to compute a prime number and one for sending it back. Here are classes to represent these messages.

```
class RequestPrime(sender: ComponentId) extends Msg(sender)
class DeliverPrime(prime:Int, sender: ComponentId) extends Msg(sender)
```

How can we implement the `handle` method of a component? We have to inspect the type and decompose the content of the message in order to take the appropriate action.

If we can change the class hierarchy, or if we had anticipated these concrete message when designing the super class `Msg`, we could add classification methods `isRequest` and `isDeliver` and an accessor method `getPrime`. We call this approach object-oriented decomposition. However, the code for handling messages has to contain a conditional expressions and several calls to these methods.

Another approach is to use double dispatch to discover the type of the message: the `handle` passes a so-called visitor object to an abstract method of class `Msg`, which is implemented in each subclass so that the relevant case of the visitor is called back. This approach corresponds to the Visitor design pattern documented by Gamma *et al* [36]. In their words, visitors are used to "represent an operation to be performed on the elements of an object structure ... without changing the classes of the elements on which it operates". However, visitors can only distinguish on one level, which can be a harsh limitation compared to the flexibility of pattern matching.

In the context of the JAVA virtual machine (and any other object system that supports runtime type information), a much more direct way of obtaining the dynamic type of a value is to use an `instanceof`-check. Although these are considered bad style, programmers make use of them frequently, in order to avoid the overhead of using a Visitor implementation. They are error prone since it is possible to perform a cast without a preceding check.

For the remainder of this thesis, we will refer to any standard object-oriented solutions as an *encoding* of pattern matching, which seems appropriate given that these approach discern and deconstruct data. Encodings provide, through low-level constructs, the same functionality that pattern matching provides in a high-level fashion. The example that shows how this leads to more verbose code.

In contrast, with suitable definitions of either case classes or extractors, we can the decomposition of incoming messages in a straight-forward, high-level manner using a pattern matching operation:

```
def handle(msg:Msg) = msg match {
  case RequestPrime(sender)        => ...
  case DeliverPrime(prime, sender) => ...
}
```

1.4.2 What pattern matching has to offer

Apart from readability and safety, a high-level construct for pattern matching provides opportunities for optimization and for static checks.

A drawback of all object-oriented solutions above is that standard compilers do not check whether such hand-crafted case distinction based on type-tests and type-casts cover all the cases, nor whether all branches can actually be entered. Pattern matching constructs in functional programming languages can be checked statically for *incompleteness* and *redundancy*, which helps catch many programmer mistakes. Incompleteness describes a match expression that does not cover all cases, whereas redundancy indicates a case that can never be entered because of a preceding one being more general.

Redundancy can be detected in any match expression, but incompleteness in functional languages depends on the fact that algebraic data types are closed. The above discussion shows that pattern matching needs to be combined with *standard* object-oriented data definitions, while at the same time offering the option of performing static checks. It turns out that *sealed* types are enough to enable incompleteness checks, therefore we do not give up on extensibility.

A balance needs to be struck between the advantages of the functional pattern matching (like improved checking and optimizations) and the extensibility considerations that have made object-oriented programming succesful. Extensibility comes in different flavors: it should for instance be possible to easily add operations without changing classes. Or the set of classes and operations is fixed. The next section tries to make a more fine-grained assessment of extensibility requirements.

1.4.3 Three Categories of Extensibility for Abstractions

After having given an abstract motivation for an object-oriented pattern matching construct, we should take a step back and reevaluate in which sense extensibility matters. While on the abstract level, it is easy to accept the object-oriented idea that operations should be tied

to data, when we turn to more concrete code problems we find that sticking dogmatically to this idea becomes counter-productive. This becomes clear when considering a few typical data types used in pattern matching under the aspect of extensibility (we could also have used "potential reuse", the point being to measure the relative benefit of following the object-oriented style). Here follow three categories of settings in which the object-oriented pattern matching problem needs to be considered. We shall often take these perspectives for granted in this document.

Standard Library Abstractions. The first category regards basic data types like tuples, the option type and immutable linked lists. These are all realized as algebraic data types in functional programming languages, thus designed to be deconstructed by pattern matching. These data types play basic roles like optional or multiple return values that should be part of any standard library. Also, we would like to avoid having primitive operations for each particular such basic data type, which would make the language unnecessarily large. These basic data types are considered fixed, they are not designed for reuse or extension, but just for use in standard programming situations. It should be noted that, useful as these types are, they do not necessarily correspond to object-oriented "dogma". Although regarded as controversial by proponents of the object-oriented style, classes alone are not appropriate because these types are merely data containers. Providing data containers in a standard Library is useful, yet users of the library can obviously not change standard library classes to add their operations.

Basic, User-Defined Abstraction. Second, programmers that are used to the "mathematic" style of aggregating data in Cartesian products will find it useful sometimes to bundle their data with operations, while still being need for some form of case distinction. So they will want to make up fixed data types designed for use with pattern matching operations (encoded or built-in), which nevertheless can be provided with methods and possibly extended in very limited ways. The immutable stack example cited above falls in this category.

Abstractions designed for Extensibility. The third category relates to data structures that are fully geared towards maintenance and program evolution, in the sense that data which is designed to be split in variants and matched has to allow changes "after the fact", i.e. after uses of data definition have been compiled and deployed. Here, representation independence is crucial and standard functional pattern matching does not offer a solution. However, the standard object-oriented approach only offers pattern matching solutions through encodings and some of these are not extensible either.

1.4.4 Scope

This thesis is concerned with object-oriented pattern matching in its first-order and generic variants as a built-in programming language construct. Its main theme is pattern matching in an object-oriented context: its use, alternatives, formal underpinnings, efficient implementation and interaction with generic types.

Applications and possible extensions are only mentioned in passing, as are connections with the fields of pattern recognition or string matching.

1.5 Outline

We describe the structure of this thesis by giving short summaries of each of the remaining chapters.

Chapter 2 Object-Oriented Pattern Matching

We first present the ideas underlying object-oriented pattern matching in a detailed, but informal manner. We motivate pattern matching as a built-in construct.

We then compare the built-in matching construct with encodings regarding conciseness of expressions as well as maintainability and evolution.

Chapter 3 Formal Semantics and Translation

In this chapter, we formally define a first-order calculus FPAT modeling SCALA. We do this by giving the context-free syntax, typing rules, operational semantics along with the properties of type soundness.

We describe an optimizing translation algorithm from FPAT to its fragment without pattern matching through a set of rewrite rules.

We prove the algorithm correct, thus asserting that the meaning of the program is preserved by each rewrite rule.

Chapter 4 Implementation

In SCALA, several aspects are different from the theoretical presentation of the preceding chapters. SCALA contains more primitives, and quite a few tricks aimed at maximizing

performance. This chapter describes how these trickes can be integrated into the formally defined translation, thus placing the formal translation in the context of the SCALA compiler. In particular, we show how to make use of lower-level primitives like jumps in in order to avoid code duplication.

Also, we discuss some extensions that are not present in the formalization. Among these are *guards*, which are boolean expressions that make execution of a case clause conditional on some property.

Chapter 5 Performance Evaluation

In Chapter 5, we compare the performance of programs that use encodings with the performance of programs that use built-in pattern matching.

Performance is assessed using three micro-benchmarks named BASE, DEPTH and BREADTH, and an application benchmark, the SCALA compiler. The micro-benchmarks test the base performance, performance on deep pattern matches and match expressions with many variants, respectively. The application benchmark shows how the execution times of the SCALA compiler change when some case classes are replaced with extractor calls. We also evaluate some implementation choices, such as replacing type-tests for case classes with operations on integer tags.

Chapter 6 Generic Pattern Matching

In this chapter, we explore interesting and useful interactions of pattern matching with parametric polymorphism. Since pattern matching has long been an essential part of functional programming, this exploration begins by reviewing second-order variants of pattern matching and algebraic data types.

After reviewing parameterized and generalized algebraic data types, we develop the type-systematic consequences of generic type-tests based on *subtyping constraints* [82]. We develop a second-order version of the FPat calculus of Chapter 3 and prove it sound.

Chapter 7 Related Work

In this chapter, we discuss the literature on the various aspects of pattern matching. Note that most research on first-order pattern matching has been carried out in the context of functional programming languages. For the second-order case, some results for object-oriented programming are available, and are put in relation with the present thesis. Additionally, we provide an overview of generalized algebraic data types.

1.6 Contributions

This thesis was elaborated in the context of the SCALA project. SCALA is a programming language aimed at at supporting component-oriented software development throught the unification of idioms from functional and object-oriented programming. The present work was implemented as a phase in the frontend of the SCALA compiler.

The SCALA compiler is the effort of several researchers, and combining pattern matching and object-oriented programming has been approached before [70, 97]. Hence, a phase that translated pattern matching expressions existed beforehand, which in the course of this work has been completely replaced. It seems thus useful to point out the particular contributions of the work described in this thesis. The contributions are summarized by the following points:

1. We precisely define *extractors*, the first approach of combining a built-in pattern matching construct with user-defined patterns that is compatible with object-oriented programming. This satisfies the extensibility requirements sought in object-oriented style and allows programmers to preserve the principle of data abstraction.

2. We formulate *case classes* as *special cases of extractors* where extensibility does not matter – they are merely shortcuts to common programming idioms that can be optimized more easily (previous approaches mainly saw case classes as algebraic data types [70], or restricted case classes so they could not extend other case classes [97]).

3. We provide a *formalization* of an object-oriented language and prove it sound – based on which we define the *formal translation* of extractor-based pattern matching and prove it correct,

4. We introduce a notion of *generic pattern matching* (pattern matching in the presence of parametric polymorphism) and recast generalized algebraic data types from functional programming in object-oriented programming using extractors. This development includes a meta-theory of *subtype constraints*, which is integrated with generic object-oriented pattern matching. We give a soundness proof for a generalized language with pattern matching that uses this meta-theory.

Publications Parts of the contributions mentioned above have been published in the following papers:

- Burak Emir, Martin Odersky and John Williams. Matching Objects with Patterns. Proceedings of European Conference on Object-Oriented Programming, LNCS, Springer Verlag (2007)

- Burak Emir, Qin Ma and Martin Odersky. Translation Correctness for First-Order Object-Oriented Pattern Matching. Proceedings of Asian Symposium on Programming Languages and Systems, LNCS, Springer Verlag (2007)
- Burak Emir, Andrew Kennedy, Claudio Russo and Dachuan Yu. Generalized Constraints and Variance for C# Generics. Proceedings of European Conference on Object-Oriented Programming, LNCS, Springer Verlag (2006)

Chapter 2

First -Order Pattern Matching

This chapter introduces pattern matching for programs written in object-oriented style. It starts with background information on pattern matching in functional programming languages, using examples in HASKELL, and a brief review of SCALA syntax. Afterwards, we describe standard encodings of pattern matching in object-oriented languages. The characteristics of the encodings are discussed with respect to the aspects of conciseness and maintainability. Then, pattern matching is introduced as a built-in language operation, by giving an informal specification and examples. Its advantages over the encodings are discussed.

2.1 Background

2.1.1 Algebraic Data Types

In typed functional programming languages like HOPE [14], MIRANDA [90], HASKELL [47] and ML [64], users can define concrete data types as disjoint sums of primitive types, tuples and function types. Each variant, or constructor, is identified with a symbolic constant. Such data types can then be discriminated using patterns, which mention the constructor label along with a collection of sub-patterns or variables to bind the constituents of a matching instance. This data definition mechanism should be considered as a building block for the wider goal of functional programming, which is give clear semantics to data and enable equational reasoning about programs. Algebraic data types remain very close to *logical* data models, which in turn allows programmers to derive algebraic properties and operations like equality – these qualities are neither sought, nor realizable in object-oriented programming.

For instance, the HASKELL program in Figure 2.1 defines binary search trees through the type `SrchT`. The values, or instances, of this type are tagged with constructors `Node` or `Leaf`.

```
data SrchT = Node Int SrchT SrchT | Leaf            {-Haskell-}

insert i Leaf           = (Node i Leaf Leaf)
insert i (Node j le ri) = if (i<j)
                      then (Node j (insert i le) ri)
                      else if (i>j)
                             then (Node j le (insert i ri))
                             else (Node j le ri)

{- desugared version -}

insert i tree = case tree of
  Leaf           -> (Node i Leaf Leaf)
  (Node j le ri) -> if (i<j)
                      then (Node j (insert i le) ri)
                      else if (i>j)
                             then (Node j le (insert i ri))
                             else (Node j le ri)
```

Figure 2.1: Binary Search Tree Insertion using Pattern Matching

A `Leaf` does not contain any information, but every instance of `Node` has an integer and two search trees as successors. The `insert` function takes an integer `i` and a tree `tree` as argument and performs a pattern match operation on instances of `SrchT`, in order to produce an updated copy of `tree` containing the integer `i`. The two given versions have exactly the same meaning, except that the former is easier to write and easier to use in equational reasoning. The function maintains the search tree invariant that for every node with key `j`, the left successors only contains smaller keys and the right successor only contains greater keys. The *match expression* **case** ... **of** contains two *case clauses*, each with a pattern to *match* instances tagged with `Node` resp. `Leaf`. If the argument was a node, it is deconstructed by binding its left and right successors to local variables. The *bodies* of the case clause contain several calls to the *constructor function* `Node`, which constructs instances of `SrchT` tagged with `Node`.

Algebraic data types like `SrchT` are defined inductively as the least set closed under their constructor functions. Expressing case distinction on the shapes of `SrchT` values is rooted in the practice of mathematical function definitions and proofs that involve algebraic terms, just like term rewriting and unification (see Wechler [94] and Baader and Nipkow [9] for an introduction in universal algebra and term rewriting techniques).

2.1.2 Other Kinds of Patterns

Apart from testing for constructors, patterns can also test whether a data item is equal to a literal constant, a named constant or, in languages with subtyping, whether it has a certain type. The nesting of patterns can express structural constraints, which can be used to represent information.

For instance, the pattern `(Node 42 Leaf)` matches values of `SrchT` that contains the literals and a leaf in this particular configuration.

Nested patterns make programs very concise and readable, because the shape of a pattern determines the meaning of the program, which leaves many visual clues in the source code. For instance, to a programmer with a mathematical background but no prior exposure to pattern matching, it soon becomes self-evident that a pattern like `(42,y)` matches pairs whose left component is 42 and whose right component can be any value. It is sometimes convenient to consider a tuple with i components as shorthand notation for an algebraic type `Tuple`i.

Systems that deal with nominal subtyping and type-tests, such as the `F`$^{\#}$ language [85], offer patterns like `:? ty as id` which dynamically test an input value for a type `ty` and binds the result to identifier `id` if it matches. This provides a nice alternative to using a type-test followed by a type-cast.

2.1.3 Closed World, Incompleteness and Extensibility

An algebraic data type definition T fixes the structure of the instances of T once and for all. The constructor tags are special cases of T, which provides a relationship between the set of instances tagged with a particular constructor and the set of instances of T that is akin to nominal (explicitly declared) subtyping. One major difference is that an algebraic data type forms a *"closed world"*: the set of constructors and their signature cannot be changed. The reason for this restriction ist that an algebraic data types defines a sum type and allows straightforward reasoning on its fixed structure. A welcome consequence of this restriction is that algebraic data types can be represented efficiently by replacing constructor tags with an integer constant.

Since the set of constructors forms a closed world, an automatic check for *incompleteness* can be performed on match expressions: The compiler can thus warn programmers who by mistake omit a case from their match expressions, which would leave the match expression *incomplete*. This check is very helpful if there are many constructors or when nested patterns allow for combinatorial combinations of algebraic data types (e.g. for a pair of two `SrchT` instances).

However, a closed world does not blend well with extensibility and separate compilation. For instance, if the type `SrchT` was extended with another constructor, then the match expression in function `insert` would become incomplete, since without intervention of the programmer it could not cover all cases of the data type. Moreover, if it was compiled separately from the data type definition, the programmer would never be notified of this fact. These limitations are known and acceptable in the functional paradigm.

A notable exception is OCAML, where subtyping between variant types is allowed [38]. However, it does not blend seamlessly with the rest of the type system, since many explicit type annotations are needed to express that a particular variant is a subtype of another variant. A more complete discussion is provided in Section 7.1.

2.2 Object-Oriented Concepts and Scala

We shall now turn to the important principle of data abstraction, shifting our attention towards data definition in object-oriented style. An abstract review of data abstraction is followed by concrete examples of SCALA syntax.

2.2.1 Data Abstraction and Encapsulation

In concrete data types, the programmer has access to all information that makes up the implementation of the data type – the number, order and type of the data items that are regrouped is completely exposed. There is no separation between specification and implementation.

Concrete types are generally at odds with the well-known software engineering principle of information hiding. The algebraic data types do not hide but *expose* their representation: A program constructed against an algebraic data type depends on the implementation details, making them hard to change later.

This leads to a key concept that is directly related to object-oriented programming, namely *data abstraction*. One way to achieve data abstraction is to introduce a form of indirection such that data types can be specified abstractly and implemented in different ways. This is usually achieved using *encapsulation*, which means regrouping data with (only) the operations that are defined for the data.

Take the example of a function that computes the height of a search tree. A `height` function that works with the definition of the above examples is easily written (and given in Figure 2.2 below). By contrast, it is not possible to change the search tree implementation to use B-trees [22] without changing the `height` function as well. These obvious connection between

```
abstract class SearchTree {
  def height: Int = 0
  def insert(i: Int): SearchTree
}

class Node(item: Int, left: SearchTree, right: SearchTree)
extends SearchTree {
  val _height: Int = {
    val lh = left.height
    val rh = right.height
    1 + { if (lh < rh) rh else lh }
  }
  override def height: Int =
    _height
  def insert(i: Int): SearchTree =
    if(i < item)
      new Node(item, left.insert(i), right)
    else if(item < i)
      new Node(item, left, right.insert(i))
    else
      this
}
object Leaf extends SearchTree {
  def insert(i: Int): SearchTree =
    new Node(i, Leaf, Leaf)
}
```

Figure 2.2: Binary Search Trees Insertion in Object-Oriented Style

data structures and algorithms has an impact on pattern matching, to the extent that pattern matching is dependent on particular data containers.

2.2.2 Class and Object Definitions in Scala

In this section, we recall how specification and implementation are handled in an object-oriented style using examples, before turning to the consequences this has for the applicability of pattern matching.

The object-oriented style of programming presumes aggregation of data and operations in *classes*. Rather than giving an introduction to object-oriented concepts, we will merely recall these concepts and introduce the notation as used in the SCALA programming language.

Figure 2.2 shows the implementation of an immutable binary search tree in SCALA. The implementation follows the technique of object-oriented decomposition, distributing the code across classes. An abstract class `SearchTree` defines the operations defined on immutable search trees. In addition to abstract class, the keyword **`trait`** can be used to define *traits*,

which are interfaces that may contain the same code as classes, but can additionally take part in multiple inheritance.

Our search trees have operations `height` and `insert`. Since search trees are immutable, `insert` returns an updated copy of the tree with the given item inserted in the appropriate place. Insertion is left abstract (there is no body to the declaration), and thus the class has to be marked with an **`abstract`** modifier. The implementation of `height` is an example of a default implementation.

The class definition for type `Node` extends `SearchTree` and contains the declaration of the constructor arguments as well as three member definitions. An instance of `Node` is constructed with an integer `item`, a `left` subtree and a `right` subtree. The definitions in this class serve the following purposes:

- a local immutable variable `_height` contains the eagerly computed height. It is defined through a SCALA block. A block is delimited by braces `{}` and yields a value (which is determined by the last expression in a block). It may also contain local variable declarations. A conditional expression is used to determine the maximum between the height of the left subtree and the height of the right subtree.
- the definition for the method `height` *overrides* the default implementation for the `height` method provided in `SearchTree`. Since the `height` is implemented in the superclass `SearchTree`, this modifier is necessary to indicate that this specific method is to be used for instances of `Node`. This happens through *late binding*, i.e. the method dispatch being deferred to execution time and involving a look-up in a virtual method table. For abstract methods, **`override`** is optional.
- the definition of method `insert` is the implementation of the insert operation.

The definition of `Leaf` illustrates an innovation in SCALA, namely singleton objects (or objects, for short). The keyword **`object`** id is used to define a name id and an unnamed class, with the effect that id is the single instance of the unnamed class. This removes the need for static ("per-class") methods and global functions, through the built-in use of the Singleton design pattern [36].

Singleton objects can be placed in class hierarchies. In the example, **`object`** `Leaf ...` defines an unnamed class which extends class `SearchTree`. It inherits the method `height` and has to implement the abstract method `insert`, like a normal class.

The type of a singleton object can be describe using a *singleton type* `Leaf.`**`type`**. For any value v, the singleton type v`.`**`type`** is inhabited by only value v (and the constant **null**), and a type test w`.isInstanceOf[`v`.`**`type`**`]` succeeds only if w and v are references to the same object.

2.3 Encodings of Pattern Matching

We reviewed pattern matching in functional programming languages, and the mechanisms for encapsulation and data abstraction, namely classes and objects. We can now ask the central question that underlies this thesis: whether and at what cost it is possible to reconcile the readability and maintainability of functional style pattern matching with object-oriented programming.

The above question leads to the problem of *object-oriented pattern matching* which is how to explore a hierarchy of classes "from the outside", i.e. without imposing any unwanted constraints on data abstraction mechanisms. Discriminating amongst objects usually involves inspecting their run-time type, and deconstructing them may take the form of determining some property or accessing some group of constituent objects. For extensibility, which is a particular aspect of maintainability, we would not want to restrict the notion of "constituent" objects to members: these only need to be available in the context of an accepting pattern, and may thus be the result of some computation. For our comparison, a pattern is merely any computable way of testing and deconstructing an object or a group of objects and binding local names to such constituent objects.

In order to make the comparison between the techniques easy, we will now define a running example that is used to demonstrate the difference techniques of implementing object-oriented pattern matching. Each technique will be reviewed with respect to the readability and maintainability of the resulting source code.

2.3.1 A simple example: Simplification of Logic Formulas

Consider symbolic manipulation of expressions. We assume a hierarchy of classes, rooted in a base class `Expr` and containing classes for specific forms of expressions, such as `And` for conjunction, `Var` for variables, and `Lit` for truth value literals. The expression forms have members according to their arity: `And` has two members `left` and `right` denoting its left and right operand, whereas `Lit` has a member `value` denoting an integer. A class hierarchy like this is expressed as follows:

```
class Expr
class Lit(val value: Boolean) extends Expr
class Var(val name: String) extends Expr
class And(val left: Expr, val right: Expr) extends Expr
```

A particular expression would then be constructed as follows

```
new And(new Lit(false), new Lit(true))
```

```
// Class hierarchy:
abstract class Expr {
  def isVar: Boolean  = false
  def isLit: Boolean  = false
  def isAnd: Boolean  = false
  def value: Boolean  = throw new NoSuchMemberError
  def name : String   = throw new NoSuchMemberError
  def left : Expr     = throw new NoSuchMemberError
  def right: Expr     = throw new NoSuchMemberError
}
class Lit(override val value: Boolean) extends Expr {
  override def isLit = true
}
class Var(override val name: String) extends Expr {
  override def isVar = true
}
class And(override val left: Expr, override val right: Expr) extends Expr {
  override def isAnd = true
}

// Simplification rule:

  if (e.isAnd) {
    val r = e.right
    if (r.isLit && r.value) e.left else e
  } else e
```

Figure 2.3: Simplification using Object-Oriented Decomposition

Consider the problem of simplifying logic expressions. A program could perform this task by trying to apply a set of simplification rules, until no more rewrites are possible. For example, looking up the truth table of logical conjunction, we could simplify conjunction involving the constant **true** as a rewrite rule, like so:

`new And(x, new Lit(true))` is replaced with `x` .

The question is how simplification rules like the one above can be expressed. For symbolic computation like the computation at hand, having a concise way of discerning and deconstructing data is essential for readability.

We can now see how this sample problem is solved using various encodings.

2.4 Object-oriented Decomposition

In object-oriented decomposition, the class hierarchy is rooted in a base class that contains *test methods* for testing the dynamic class of an object and *accessor methods* which let one refer

to members of specific subclasses. In each subclass, the relevant methods are overridden. Figure 2.3 demonstrates this technique with the logic simplification example.

We fill the base class `Expr` with test methods `isVar`, `isLit` and `isAnd`, one for each subclass of `Expr`. The default implementations of the test methods all return **`false`**. Each subclass overrides the appropriate test method to return **`true`**. Accessor method for every publicly visible field or every subclass are also put in the base class. By default, these default accessors throw a `NoSuchMemberError` exception. In each subclass, we supply a proper accessors definition for the accessible members it contains. In SCALA, this can be conveniently expressed in one syntactic construct, using the syntax **`override val ...`** in a class parameter.

Note that in dynamically typed language like SMALLTALK, default implementation of accessors are not needed, since `NoSuchMethod` messages are generated automatically for every call to a method that cannot be found. The notational overhead for the object-oriented decomposition technique then diminishes significantly. This may explain why the technique is more used in dynamically typed languages than in statically typed ones. Still, name pollution makes this problematic for large class hierarchies[1].

Object-oriented decomposition also suffers from a lack of extensibility. If one adds another subclass of `Expr`, the base class has to be augmented with new test and accessor methods. Again, dynamically typed languages can alleviate this problem to some degree using metaprogramming facilities where classes can be augmented and extended at run-time.

The second half of Figure 2.3 shows the code of the simplification rule. The rule inspects the given term stepwise, using the test functions and accessors given in class `Expr`.

2.5 Visitors

Visitors [36] perform two method calls (double dispatch) to recover the runtime type of an object. Figure 2.4 shows logic simplification with visitors. We are giving an instance of the Visitor design pattern that allows for incomplete matches, hence a visitors with default [97]. The `Visitor` class contains a *case-method* named *caseX* for each subclass X of `Expr`.

Every *caseX* method takes an argument of type X. The `Visitor` class has a generic type parameter `T`, which is used by each *caseX* method as its return type. In class `Visitor` every case-method has a default implementation which forward to the `otherwise` method.

[1] With regard to this problem, Christian Plesner Hansen writes:"...it often means polluting other objects with methods that may be convenient for you but are completely irrelevant to the object itself and anyone else who might use it. Case in point: in Squeak the Object class has over 400 methods including isSketchMorph, playSoundNamed and, the winner by unanimous decision, hasModelYellowButtonMenuItems." `http://blog.quenta.org/2006/03/match.html`

```
// Class hierarchy:
abstract class Visitor[T] {
  def caseAnd(t: And): T    = otherwise(t)
  def caseLit(t: Lit): T    = otherwise(t)
  def caseVar(t: Var): T    = otherwise(t)
  def otherwise(t: Expr): T = throw new MatchError(t)
}
abstract class Expr {
  def matchWith[T](v: Visitor[T]): T
}
class Lit(val value: Boolean) extends Expr {
  def matchWith[T](v: Visitor[T]): T = v.caseLit(this)
}
class Var(val name: String) extends Expr {
  def matchWith[T](v: Visitor[T]): T = v.caseVar(this)
}
class And(val left: Expr, val right: Expr) extends Expr {
  def matchWith[T](v: Visitor[T]): T = v.caseAnd(this)
}

// Simplification rule:

  e.matchWith {
    new Visitor[Expr] {
      override def caseAnd(m: And) =
        m.right.matchWith {
          new Visitor[Expr] {
            override def caseLit(n: Lit) =
              if (n.value) m.left else e
            override def otherwise(e: Expr) = e
          }
        }
      override def otherwise(e: Expr) = e
    }
```

Figure 2.4: Simplification using Visitors

```
// Class hierarchy:
abstract class Expr
class Lit(val value: Boolean) extends Expr
class Var(val name: String) extends Expr
class And(val left: Expr, val right: Expr) extends Expr

// Simplification rule:

  if (e.isInstanceOf[And]) {
    val m = e.asInstanceOf[And]
    val r = m.right
    if (r.isInstanceOf[Lit]) {
      val n = r.asInstanceOf[Lit]
      if (n.value) m.left else e
    } else e
  } else e
```

Figure 2.5: Simplification using Type-Test/Type-Cast

The `Expr` class declares a generic abstract method `matchWith`, which takes a visitor as argument. Instances of subclasses X implement the method by invoking the corresponding *caseX* method in the visitor object on themselves.

The bottom half of Figure 2.4 shows how the simplification rule is rendered with visitors. Since the pattern has two levels, two visitor objects have to created, one per level. Since the pattern on the third-level involves a primitive boolean, we can use its value directly. Each visitor object defines two methods: the *caseX* method corresponding to the matched class, and the `otherwise` method corresponding to the case where the match fails.

2.6 Type-Test and Type-Cast

The most direct form of discrimination uses the type-test and type-cast instructions available in JAVA or C++ with runtime type information enabled. Figure 2.5 shows logic simplification using this method. In SCALA, the test whether a value x is a non-null instance of some type T is expressed using the pseudo method invocation x`.isInstanceOf[`T`]`, with T as a type parameter. Analogously, the cast of x to T is expressed as x`.asInstanceOf[`T`]`.

2.7 Typecase

The typecase construct accesses run-time type information in much the same way as type-tests and type-casts. It is however more concise and secure since a cast is never done independently of a test. Figure 2.6 shows the logic simplification example using typecase. In

```
// Class hierarchy:
abstract class Expr
class Lit(val value: Boolean) extends Expr
class Var(val name: String) extends Expr
class And(val left: Expr, val right: Expr) extends Expr

// Simplification rule:

  e match {
    case m: And =>
      m.right match {
        case n: Lit =>
          if (n.value) m.left else e
        case _ => e
      }
    case _ => e
  }
```

Figure 2.6: Simplification using Typecase

SCALA, typecase is an instance of a more general pattern matching expression of the form $expr$ **match** { $cases$ }. Each case is of the form **case** p => b; it consists of a pattern p and an expression or list of statements b. There are several kinds of patterns in SCALA. The typecase construct uses patterns of the form $x : T$ where x is a variable and T is a type. This pattern matches all non-null values whose runtime type is (a subtype of) T. The pattern binds the variable x to the matched object. The other pattern in Figure 2.6 is the *wildcard pattern* _, which matches any value.

2.8 Pattern Matching as a Language Construct

In contrast to the encodings, which use method invocation, late binding and conditionals with type-tests to achieve the desired case distinction, pattern matching as a language construct provides a high-level solution that is tailored to the pattern matching problem. In this section, we describe a matching construct with a rich pattern sub-language that is used in SCALA. However, this description merely deals with syntactic convention which are directly inspired by functional pattern matching constructs. The relationship between matching and data, which is particularly important in object-oriented style, is elaborated in the following sections.

A pattern in SCALA is constructed from the following elements:

- Variables such as x or `right`. These match any value, and bind the variable name to the value. The wildcard pattern _ is used as a shorthand if the value need not be named.

- Type patterns such as `x: Int` or `_: String`. These match all values of the given type, and bind the variable name to the value. Type patterns were already introduced in Section 2.7.

- Constant literals such as 1 or "abc". A literal matches only itself.

- Named constants such as `None` or `Nil`, which refer to immutable values. A named constant matches only the value it refers to. The comparison is done via the `equals` method.

- Constructor patterns of the form $C(p_1, \ldots, p_n)$, where C is a pattern constructor and $p_1, \ldots, p_n$ are patterns. Such a pattern can have two meanings, which will both be described below: If C is a case class, it matches all instances which were built from values $v_1, \ldots, v_n$ matching the patterns $p_1, \ldots, p_n$. If C is an extractor, its behavior is defined by the extractor implementation.

  For case classes, it is not required that the class instance is constructed directly by an invocation $C(v_1, \ldots, v_n)$. It is also possible that the value is an instance of a subclass of C, from where a super-call constructor invoked C's constructor with the given arguments. Another possibility is that the value was constructed through a secondary constructor, which in turn called the primary constructor with arguments $v_1, \ldots, v_n$. Thus, there is considerable flexibility for hiding constructor arguments from pattern matching.

- Variable binding patterns of the form `x @` p where `x` is a variable and p is a pattern. Such a pattern matches the same values as p, and in addition binds the variable `x` to the matched value.

To distinguish variable patterns from named constants, we require that variables start with a lower-case letter whereas constants should start with an upper-case letter or special symbol. There exist ways to circumvent these restrictions: To treat a name starting with a lower-case letter as a constant, one can enclose it in back-quotes, as in **case** `` `x` => ... ``. To treat a name starting with an upper-case letter as a variable, one can use it in a variable binding pattern, as in **case** `X @ _ => ...`.

2.9 Case Classes

Case classes in SCALA provide convenient shorthands for constructing and analyzing data. Figure 2.7 presents them in the context of logic simplification.

The programmer signals the intent of turning a class into a case class by using the **case** modifier. This modifier has several effects. First of all, it provides a convenient notation for

```
// Class hierarchy:
abstract class Expr
case class Lit(value: Boolean) extends Expr
case class Var(name: String) extends Expr
case class And(left: Expr, right: Expr) extends Expr

// Simplification rule:

  e match {
    case And(x, Lit(true)) => x
    case _ => e
  }
```

Figure 2.7: Simplification using Case Classes

constructing data without having to write **new**. For instance, the expression `And(Lit(true), Var(x))` would be a shorthand for **new** `And(new Lit(true), new Var(x))`, assuming the class hierarchy of Figure 2.7. Note here the similarity to constructors as seen for algebraic data types in functional programming (Section 2.1.1). Second, case classes allow pattern matching on their constructor. Such patterns are written exactly like constructor expressions, but are interpreted as "templates" that can be filled by matching values. For instance, the pattern `And(x, Lit(true))` matches all values instances of `And`, whose `right` is an instance of `Lit` that has a `value` field equal to **true**. If the pattern matches, the variable `x` is bound the `left` operand of the given value.

2.9.1 Examples of Case Classes

With case classes, binary search trees can be expressed like in Figure 2.8, which is very close to the functional programming code in Figure 2.1.

Case classes are used to express lists, streams, messages, symbols, documents, and XML data in SCALA's libraries. We will explain two groups of case classes in more detail, because they are used in the following sections. First, there are classes representing optional values:

```
sealed abstract class Option[+T]
case class Some[T](value: T) extends Option[T]
case object None extends Option[Nothing]
```

Class `Option[T]` represents optional values of type `T`. The subclass `Some[T]` represents a value which is present whereas the sub-object `None` represents absence of a value. The '$+$' in the type parameter of `Option` indicates that optional values are covariant: if S is a subtype of T, then `Option[`S`]` is a subtype of `Option[`T`]`. The type of `None` is `Option[Nothing]`, where `Nothing` is the bottom in SCALA's type hierarchy. Because of covariance, `None` thus conforms to every option type.

```
// Class hierarchy:
abstract class SearchTree {
  def insert(i: Int): SearchTree = this match {
    case Leaf          =>
      Node(item, Leaf,Leaf)
    case Node(j,le,ri) =>
      if (i < j)
        Node(j, le.insert(i), ri)
      else if (i > j)
        Node(j, le, ri.insert(i))
      else
        this
  }
}

case class Node(i:Int, l:SearchTree, r:SearchTree)
  extends SearchTree
case object Leaf
  extends SearchTree
```

Figure 2.8: Search Tree Insertion using Case Classes

For the purpose of pattern matching, `None` is treated as a named constant, just as any other singleton object. The **case** modifier of the object definition only changes some standard method implementations for `None`, as explained in Section 2.10.5. A typical pattern match on an optional value would be written as follows.

```
v match {
  case Some(x) => "do something with x"
  case None    => "handle missing value"
}
```

Option types are recommended in SCALA as a safer alternative to **null**. Unlike with **null**, it is not possible to accidentally assume that a value is present since an optional type must be matched to access its contents.

Tuples are another group of standard case classes in SCALA. All tuple classes are of the form:

```
case class Tuplei[T1, ..., Ti](_1: T1, ..., _i: Ti)
```

An abbreviated syntax (T_1, ..., T_i) for the tuple type Tuple$i$[T_1, ..., T_i] exists, and analogous abbreviations exist for expressions and patterns.

2.9.2 Case Classes and Class Hierarchies

Algebraic data types correspond closely to case classes in a flat hierarchy, i.e. with one non-case super class. Unlike algebraic data types, case classes are merely normal classes that

have a modifier to indicate that their constructor arguments are public fields. This means we can allow case classes to be arranged in class hierarchies.

However, this raises some issues that we will explain using an example. Consider the following case class hierarchy, which is used to represent points, i.e. members of two-dimensional and three-dimensional vector spaces:

```
case class Point2(x:Float, y:Float)
case class Point3(x:Float, y:Float, z:Float) extends Point2(x,y)
case object Origin extends Point3(0.0, 0.0, 0.0)
```

The use of **extends** indicates that an instance of `Point3` is at the same time a `Point2`. Furthermore, the singleton object `Origin` is a particular `Point3` located at the origin.

The relations between classes `Point2`, `Point3` and `Origin.type` should be respected by the acceptance relationship: A value v matches a case pattern $C(...)$, if the value has been constructed using a constructor D that is a subtype of C (either trivially, directly or indirectly). The following match expression to produce output `(0.0,0.0)`:

```
Origin match { case Point2(x,y) => print (x,y) }
```

2.10 Extractors

We have seen now that case classes provide some of the benefits of functional pattern matching, at the cost of giving up encapsulation. In order to regain encapsulation, it seems necessary to diminish the difference between pattern matching and other operation (i.e. methods, which can be redefined at will and that adhere to the data abstraction mechanisms in the language). An easy way to achieve this is to express patterns via methods, and thus render them user-defined. This approach is inspired by Wadlers's *views* [92], which aim to regain data abstraction by introducing user-defined conversions between data types.

2.10.1 User-Defined Patterns

An *extractor* is a method that can serve as a user-defined pattern. It does so by testing and deconstructing its argument, returning a value in the case of acceptance and returning nothing in the case of rejection. In SCALA, this mechanism is integrated by following some syntactic conventions, that we introduce by example. The following object `Twice` enables patterns of even numbers:

```
object Twice {
  def apply(x:Int) = x*2
  def unapply(z:Int) = if(z%2==0) Some(z/2) else None
}
```

It defines an `apply` function, which provides a new way to write integers: `Twice(x)` is now an alias for `x * 2`. SCALA uniformly treats objects with `apply` methods as functions, inserting the call to `apply` implicitly. Thus, `Twice(x)` is really a shorthand for `Twice.apply(x)`.

The `unapply` method in `Twice` reverses the construction in a pattern match. It tests its integer argument `z`. If `z` is even, it returns `Some(z/2)`. If it is odd, it returns `None`. In a pattern match, ab `unapply` method is invoked implicitly, as in the following example, which prints "42 is two times 21":

```
val x = Twice(21)
x match {
  case Twice(y) => Console.println(x+" is two times "+y)
  case _        => Console.println("x is odd")
}
```

In this example, `apply` is called an *injection*, because it takes an argument and yields an element of a given type. `unapply` is called an *extraction*, because it extracts parts of the given type. Injections and extractions are often grouped together in one object, because then one can use the object's name for both a constructor and a pattern, which simulates the convention for pattern matching with case classes. However, it is also possible to define an extraction in an object without a corresponding injection. The object itself is often called an *extractor*, independently of the fact whether it has an `apply` method or not.

It may be desirable to write injections and extractions that satisfy the equality F`.unapply(`F`.apply(x)) == Some(x)`, but we do not require any such condition on user-defined methods. One is free to write extractions that have no associated injection or that can handle a wider range of data types.

Patterns referring to extractors look just like patterns referring to case classes, but they are implemented differently. Matching against an extractor pattern like `Twice(x)` involves a call to `Twice.unapply(x)`, followed by a test of the resulting optional value. The code in the preceding example would thus be expanded as follows:

```
val x = Twice.apply(21)   // x = 42
Twice.unapply(x) match {
  case Some(y) => Console.println(x+" is two times "+y)
  case None => Console.println("x is odd")
}
```

Extractor patterns can also be defined with numbers of arguments different from one. A nullary pattern corresponds to an `unapply` method returning a Boolean. A pattern with more than one element corresponds to an `unapply` method returning an optional tuple. The result of an extraction plays the role of a "representation-object", whose constituents (if any) can be bound or matched further with nested pattern matches.

2.10.2 Patterns and Types

Pattern matching in SCALA is loosely typed, in the sense that the type of a pattern does not restrict the set of legal types of the corresponding selector value. The same principle applies to extractor patterns. For instance, it would be possible to match a value of SCALA's root type `Any` with the pattern `Twice(y)`. In that case, the call to `Twice.unapply(x)` is preceded by a type test whether the argument `x` has type `Int`. If `x` is not an `Int`, the pattern match would fail without executing the `unapply` method of `Twice`. This choice is convenient, because it avoids many type tests in `unapply` methods which would otherwise be necessary. As we will see later, this also introduces opportunities for optimization (Chapter 4) and allows us to handle parameterized class hierarchies, as will be explained in (Chapter 6).

2.10.3 Representation Independence

Unlike case-classes, extractors can be used to hide data representations. As an example consider the following class of complex numbers, implemented by case class `Cart`, which represents numbers by Cartesian coordinates.

```
abstract class Complex
case class Cart(re: double, im: double) extends Complex
```

Complex numbers can be constructed and matched using the syntax `Cart(r, i)`. The following injector/extractor object provides an alternative access with polar coordinates:

```
object Polar {
  def apply(mod: Double, arg: Double): Complex =
    new Cart(mod * Math.cos(arg), mod * Math.sin(arg))

  def unapply(z: Complex): Option[(Double, Double)] =
    z match {
      case Cart(re, im) =>
        Some(sqrt(re * re + im * im), Math.atan2(im,re))
    }
}
```

With this definition, a client can now alternatively use polar coordinates such as `Polar(m, e)` in value construction and pattern matching.

2.10.4 Logic Simplification Revisited

Figure 2.9 shows the logic simplification example using extractors. The simplification rule is exactly the same as in Figure 2.7. But instead of case classes, we now define normal classes

```
// Class hierarchy:
abstract class Term
class Lit(val value: int) extends Term
class Var(val name: String) extends Term
class And(val left: Term, val right: Term) extends Term

object Lit {
  def apply(value: Boolean) = new Lit(value)
  def unapply(n: Lit) = Some(n.value)
}
object Var {
  def apply(name: String) = new Var(name)
  def unapply(v: Var) = Some(v.name)
}
object And {
  def apply(left: Term, right: Term) = new And(left, right)
  def unapply(m: And) = Some (m.left, m.right)
}

// Simplification rule:

  e match {
    case And(x, Lit(true)) => x
    case _ => e
  }
```

Figure 2.9: Simplification using Extractors

with one injector/extractor object per each class. The injections are not strictly necessary for this example; their purpose is to let one write constructors in the same way as for case classes.

Even though the class hierarchy is the same for extractors and case classes, there is an important difference regarding program evolution. A library interface might expose only the objects `Lit`, `Var`, and `And`, but not the corresponding classes. That way, one can replace or modify any or all of the classes representing logic expressions without affecting client code.

Note that every X`.unapply` extraction method takes an argument of the alternative type X, not the common type `Term`. This is possible because an implicit type test gets added when matching on a term. However, a programmer may choose to provide a type test himself:

```
  def unapply(x: Term) = x match {
    case m:And => Some (m.left, m.right)
    case _     => None
}
```

This removes the target type from the interface, effectively hiding the underlying representation.

2.10.5 Encoding Case Classes using Extractors

For the purposes of type-checking, a case class can be seen as syntactic sugar for a normal class together with an injector/extractor object. This is exemplified in Figure 2.10, where a syntactic desugaring of the following case class is shown:

```
case class And(left: Expr, right: Expr) extends Expr
```

Given a class C, the expansion adds accessor methods for all constructor parameters to C. It also provides specialized implementations of the methods `equals`, `hashCode` and `toString` inherited from class `Object`. Furthermore, the expansion defines an object with the same name as the class (SCALA defines different name spaces for types and terms; so it is legal to use the same name for an object and a class). The object contains an injection method `apply` and an extraction method `unapply`. The injection method serves as a factory; it makes it possible to create objects of class C writing simply $C(\ldots)$ without a preceding **`new`**. The extraction method reverses the construction process. Given an argument of class C, it returns a tuple of all constructor parameters, wrapped in a `Some`.

```
class And(_left: Expr, _right: Expr) extends Expr {
  // Accessors for constructor arguments
  def left  = _left
  def right = _right

  // Standard methods
  override def equals(other: Any) = other match {
    case m: And => left.equals(m.left) && right.equals(m.right)
    case _ => false
  }
  override def hashCode = hash(this.getClass, left.hashCode, right.hashCode)
  override def toString = "And("+left+",␣"+right+")"
}

object And {
  def apply(left: Expr, right: Expr) = new And(left, right)
  def unapply(m: And) = Some(m.left, m.right)
}
```

Figure 2.10: Expansion of Case Class And

2.11 Pattern Matching and Multi-Methods

For completeness, we consider *multiple dispatch* as another method that allows us to encode pattern matching [21]. This method is studied separately, since pattern matching and multiple dispatch are, to some extent, inter-definable.

Considering method invocation $e_0.\mathsf{m}(e_1, \ldots, e_n)$, multiple dispatch refers to the choice of a method among a number of overloaded alternatives based on the run-time types of all arguments (rather than only the run-time type of the receiver e_0). The overloaded alternatives of m are then often considered as being a single method with multiple branches, or multi-methods.

When pattern matching includes typed patterns (run-time type tests), it can be used to implement multiple dispatch. We sketch how such an implementation could be effected, in order to show that some overlap exists between the problems solved by multi-methods and the problems solved by pattern matching.

2.11.1 Defining and Collecting Alternatives

In Figure 2.11 (upper half), an example of multi-methods is given in an imaginary syntax: It contains classes `Shape`, `Rect` and `Square` and a multi-method `intersect`. The method definitions in classes `Rect` and `Square` do not override, but *specialize* the general intersection method provided in class `Shape`. A method call $s_1.\mathsf{intersect}(s_2)$ is dispatched according to

```
class Shape { ...
  def intersect(s: Shape): Boolean = {b1}
}
class Rect   extends Shape { ...
  def intersect(s @: Rect):   Boolean = {b2}
  def intersect(s @: Square): Boolean = {b3}
}
class Square extends Rect { ...
  def intersect(s @: Square): Boolean = {b4}
}
```

```
class Shape { ...
  def intersect(s: Shape): Boolean = {b1}
}
class Rect extends Shape { ...
  override def intersect(s:Shape): Boolean = s match {
    case s:Square => b3
    case s:Rect   => b2
    case _        => super.intersect(s)
  }
}
class Square extends Shape { ...
  override def intersect(s:Shape): Boolean = s match {
    case s:Square => b4
    case _        => super.intersect(s)
  }
}
```

Figure 2.11: Translating Multi-Methods to Match Expressions

the run-time types of s_1 and s_2: If both are instances of `Rect`, then b_2 will be executed; if s_1 is a `Rect` and s_2 a `Square`, then b_3 is executed; if both are squares, b_4 is executed, and b_1 is executed in all other cases. It is easy to see how this generalizes to multi-methods with more than one argument.

The lower part of Figure 2.11 contains an equivalent program that uses pattern matching. All branches in subclasses are replaced with a single method **def** `intersect(s:Shape)` that performs pattern matching (in fact, just a run-time type check) on its first argument. The various branches are all collected in the match expression and cases that are not covered by specializations are deferred using a **super** call. This translation would allow us to use multi-methods in execution environments that only offer single-dispatch, which include the Java Virtual Machine and the .NET Common Language Runtime.

Note the order in match expression of class `Rect`: the more specific type `Square` has to be tested before the type `Rect`, otherwise specializations would not follow the semantics of picking the method with the most specific argument types. Another subtlety in this match

is that intersecting a `Square` with a `Rect` will unnecessarily repeat the type test for `Square`: since the receiver is a `Square`, the presence of the specialization means that the argument has already been tested against the type `Square` and a **super** call was effected.

Matching as defined above is easily generalized to deal with tuples of expressions (which will indeed be done in the formal discussion of the following chapters). This means that at least some multi-methods can be expressed very straightforwardly in terms of pattern matching. On the other hand, it has been proposed to allow general, nested patterns in method signatures and translate them in a fashion that is similar to pattern matching [77].

We end this discussion by mentioning that multi-methods pose challenges to type checking which are independent of pattern matching. Examples include the problem of locating a multi-method such as to allow access to private or protected members of its arguments, or the static detection of specialization conflicts that arise in the presence of multiple inheritance. Certainly, our sketched translation does not imply that all forms and use-cases of multiple dispatch could or should always be translated away to match expressions.

2.12 Discussion of Readability and Maintainability

We shall now discuss readability and maintainability, given the source code of the logic simplification example and the consequences each technique has.

2.12.1 Readability

The best readability is achieved with case classes, which only add a keyword to the class hierarchy while benefitting from concise matches. Next comes typecase, which (like type-tests and type-casts) works for any class hierarchy without the need for annotation. However, the match expressions are less readable than for case classes, considering that children have to be accessed explicitly and deep matches are not supported. Type-test and type-cast are slightly worse because they have to resort to control constructs.

Extractors have a high overhead for the class hierarchy, but in return the match expressions are as concise as for case classes. The readability is therefore at a medium.

Object-oriented decomposition has a high overhead for the hierarchy and pollutes the namespace of all concerned classes with accessors methods. The matching uses control constructs and is thus overall hard to read. Visitors have the worst readability: not only do they require a lot of boilerplate code to be written, but the match expressions involve object constructions and many scopes to be opened.

2.12.2 Maintainability

Extractors have excellent maintainability because they can be changed independently from the types they match. New patterns and new variants can be invented at will, due to the loose coupling that allows extensions without interfering with the class hierarchy.

Object-oriented decomposition is relatively easy to maintain and extend, as long as no new classes are added. The programmer can write new patterns as long as they only depend on existing test and accessor methods. With object-oriented decomposition, it can be possible to control visibility of accessor methods.

Type-test and type-cast (like typecase) are tied to the class hierarchy, which is good for adding new classes, but bad if matching behavior needs to be changed without changing the class hierarchy. They expose the representation of the types they match, which therefore makes program evolution harder.

Visitors are not extensible (in contrast to extensible visitors [54]), since they are tightly coupled with the class hierarchy. Moreover, existing visitors depend on the existing visitor interface, which can therefore not be changed without changing all visitors. Visitors also expose the representation of the class hierarchy they visit.

2.12.3 Multi-Methods are different

Multi-methods are a technique based on method definitions. It thus offers less readable solution when matching expressions are required. On the other hand, when the programmer intends to provide operations through methods, then multi-methods increase the maintainability in a different manner than the techniques presented above (by allowing us to add operations to classes later). Extensibility with respect to patterns is the same as for classes and type-test/type-casts: Since patterns (in this particular presentation of multi-methods) are tied to types, it is not possible to define new patterns. In summary, multi-methods seem to be provide a partial solution to the object-oriented pattern matching problem, while solving problems that cannot be solved by an expression-level pattern matching construct.

2.13 Summary

We have defined the object-oriented pattern matching problem and compared existing solutions with two mechanisms paired with built-in pattern matching, case classes and extractors. Of these two, the former has been proposed before [70] – here, case classes have been presented in a more general manner and furthermore are shown to be definable in terms

of extractors. Moreover, extractor are more amenable to maintenance, since they are user-defined patterns that are resolved like normal method calls. This makes extractors a better choice to formulate a specification and translation algorithm in a rigorous manner.

We also sketched how multi-methods can be considered as a convenient syntax for pattern matching expression, indicating some overlap between multiple dispatch and pattern matching that is purely defined on type tests.

Building on these considerations, we can now turn to a formalization of an object-oriented programming language that offers a pattern matching construct and extractors.

Chapter 3

Formal Semantics and Translation

This chapter contains formal definitions and a soundness proof for a first-order object-oriented language FPAT that offers runtime type inspection and pattern matching. Furthermore, we give a formal, optimizing translation of pattern matching expressions to lower level primitives and prove it correct. The reasoning is centered around a calculus based on Featherweight Java [46].

Through this formalization, we can present possible answers to the question whether algorithms for optimized translation of pattern matching known from functional programming can be applied to extractors, and which conditions (if any) have to be satisfied by extractors in order to prove that an optimized translation preserves the meaning of the program.

3.1 Syntax and Semantics

The syntax and operational semantics of FPAT are given in Figure 3.1 and Figure 3.3. The typing rules and auxiliary definitions are given in Figure 3.5, Figure 3.7, Figure 3.8 and Figure 3.9. The calculus is based on Featherweight Java (FJ), but with the difference that semantics are defined using strict, left-to-right big-step style. We briefly review the definitions. We then follow a recent approach to show type soundness using a coinductive definition of divergent programs.

3.1.1 Syntax

What follows is a short presentation of the notation and rules. The metavariables are consistently chosen from the following alphabets:

A, B, C, D, E	class name	cd	class definition
x	variable name	md	method definition
f, g, h	field name	an	optional annotation
m	method name	$\hat{v}$	open values
a, b, d, e	expression	c	pattern case
p, π	pattern	i, j, k, l, m, n	integer indices
q	result	σ, ρ	substitutions
u, v, w	values	Γ	typing contexts
$\dot{u}, \dot{v}, \dot{w}$	value or **null**	ξ, ζ	target contexts

In order to be precise about length and indices of sequences, a sequence $\alpha_1..\alpha_n$ is abbreviated as ${\alpha_\star}^{1..n}$, where α can be an expression, a name-type binding, or a judgment. The empty sequence is written $\bullet$. Multiple occurrence of the $\star$ indicate that the same index appears at multiple positions. Moreover, we shall need to express sequences with holes, so $\alpha_\star, \beta, {\alpha_\star}^{1..]i[..n}$ stands for ${\alpha_\star}^{1..i-1}, \beta, {\alpha_\star}^{i+1..n}$.

An FPat program ${cd_\star}^{1..n}; e$ is a set of class definitions and a top-level expression. Class definitions are kept in a class table, which we leave implicit throughout the formalization and which satisfies the important properties that inheritance cycles and duplicate entries are absent. Classes have an explicit superclass as well as field declarations and method definitions, all publicly accessible. Methods can have a @**safe** annotation to indicate that they terminate without throwing an exception (more on this later). The class hierarchy induces a subtype relation <:, of which the magic class Obj forms the largest element and the magic class Exc forms the smallest. These two types are magic because they do not have definitions in the class table. We also have a least upper bound $C \sqcup D$ operation, which is the least type E in the hierarchy that satisfies $C <: E$ and $D <: E$.

There are 8 expression forms: **null**, variables x, field selection $e.\mathsf{f}$, method invocation $e.\mathsf{m}({e_\star}^{1..n})$, object construction $C({e_\star}^{1..n})$, exception **throw**, test expressions $a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}$ and match expressions ${e_\star}^{1..n}$ **match** $\{{c_\star}^{1..k}\}$, where each case clause c has the shape **case** ${p_\star}^{1..n} \Rightarrow b$ and we convene that the last clause has only variable patterns. The calculus does not model assignment nor object identity. The free variables fv are defined in the straightforward manner, e.g. the free variables of a test expression $a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}$ is the union of free variables of a, e, and the free variables of d without x.

Note that, unlike the SCALA examples of the previous chapter, we do not use optional values here – instead we will the **null** value to signal rejection of an extractor call.

3.1.2 Semantics

We briefly describe the operational semantics of the fragment without pattern matching, in order to be self-contained. The semantics specific to pattern matching are deferred to a

separate section below.

Terminating computation of meaningful expressions is modeled by a big-step evaluation relation $e \Downarrow q$ that takes expressions e to results q. A result q is either a value v, the **null** result or the exception **throw**. To express that a result may be either a value v or **null**, we use the notation $\dot{v}$ ("optional value").

Substitutions are restricted to map variables only to values or **null**. A value is always the outcome of an object construction $C(\dot{v}_\star^{1..n})$, which is written without **new**. There is no explicitly declared constructor, instead we use the field order determined by the inheritance hierarchy (specified in the auxiliary judgment $\mathsf{fields}(C)$). The following correct program illustrates how arguments in object construction relate to fields in class definitions:

$$\begin{array}{l}\textbf{class}\ D(\mathsf{f}\colon A)\ \triangleleft\ \mathsf{Obj}\ \{\ldots\}\\ \textbf{class}\ C(\mathsf{g}\colon B)\ \triangleleft\ D\ \{\ldots\}\\ C(A(), B())\end{array}$$

Note that in this calculus all classes are case classes: there is no way to restrict visibility and the constructor arguments are at the same time public fields.

Rules (Rfld), (Rinvk), (Rnew) in Figure 3.3 describe field access, method invocation and object construction. The auxiliary judgment $\mathsf{mbody}(\mathsf{m}, C)$ specifies how to lookup method bodies. Rules (Cfld), (Crcv), (Carg), (Cnew) throw or propagate exceptions.

The only significant use of **null** happens in test expressions. Their behavior is specified in rules (Rcst) and (Rskp): if the tested expression, or *scrutinee*, is not **null** and its type is lesser than the required type, it is bound to a local variable and the first branch is evaluated (Rcst). Otherwise, the second branch is evaluated (Rskp). If the scrutinee throws, the exception is propagated (Ctst).

The relation $\Downarrow$ does not specify the behavior of meaningless or non-terminating programs. To show type soundness, divergent programs are defined using a relation $\Uparrow$ in Figure 3.10. Meaningless expressions are then precisely those that neither terminate nor diverge.

3.2 Semantics of Matching

Pattern matching expressions contain one or more case clauses, each of which compares the n input values against n patterns. The last clause may only have variable patterns. This excludes pathological expressions of the form $e_\star^{1..k}$ **match** $\{\}$ and ensures that the behavior is defined for any possible combination of input values. In other words, the pattern match is *complete* (also called *exhaustive*) – we will discuss in Chapter 4 how this property can be checked automatically.

$$
\begin{array}{lcl}
cd & ::= & \textbf{class}\ C(\mathsf{f}_\star : C_\star^{1..n}) \lhd D\ \{md_\star^{1..k}\} \\
md & ::= & an\ \textbf{def}\ \mathsf{m}(\mathsf{x}_\star : C_\star^{1..n}) : C = \{e\} \\
an & ::= & \textbf{@safe} \quad | \quad (\mathsf{empty}) \\
a, b, d, e & ::= & \textbf{null} \\
 & | & \mathsf{x} \\
 & | & e.\mathsf{f} \\
 & | & e.\mathsf{m}(e_\star^{1..n}) \\
 & | & C(e_\star^{1..n}) \\
 & | & \textbf{throw} \\
 & | & e?\{\mathsf{x}\colon C \Rightarrow e\}/\{e\} \\
 & | & e_\star^{1..n}\ \textbf{match}\ \{c_\star^{1..m}\} \\
 & & (\text{convention: } c_m \equiv \textbf{case}\ \mathsf{x}_\star^{1..n} \Rightarrow e) \\
c & ::= & \textbf{case}\ p_\star^{1..n} \Rightarrow e \\
p, \pi & ::= & \mathsf{x} \quad | \quad C(\hat{v}_\star^{1..n}).\mathsf{m}(p_\star^{1..k}) \\
q & ::= & \dot{v} \quad | \quad \textbf{throw} \\
\dot{u}, \dot{v}, \dot{w} & ::= & v \quad | \quad \textbf{null} \\
u, v, w & ::= & C(\dot{v}_\star^{1..k}) \\
\hat{v} & ::= & \mathsf{x} \quad | \quad C(\hat{v}_\star^{1..k})
\end{array}
$$

Figure 3.1: Grammar

Subtyping $C <: D$

$$\frac{}{C <: \mathsf{Obj}}\ (\mathsf{Sobj}) \qquad \frac{}{\mathsf{Exc} <: C}\ (\mathsf{Sthr})$$

$$\frac{}{C <: C}\ (\mathsf{Sref}) \qquad \frac{C <: D \quad D <: E}{C <: E}\ (\mathsf{Stran})$$

$$\frac{\textbf{class}\ C(\mathsf{f}_\star : C_\star^{1..n}) \lhd D\ \{md_\star^{1..m}\}}{C <: D}\ (\mathsf{Sext})$$

$C \sqcup D \stackrel{\textbf{def}}{=}$ smallest E such that $C <: E$ and $D <: E$.

Figure 3.2: First-Order Subtyping

Computation $e \Downarrow q$

$$\frac{e \Downarrow C(\dot{v}_\star^{1..n}) \quad \mathsf{fields}(C) = \mathsf{f}_\star\colon C_\star^{1..n}}{e.\mathsf{f}_i \Downarrow \dot{v}_i}\ (\mathsf{Rfld})$$

$$\frac{\begin{array}{c} e \Downarrow C(\dot{v}_\star^{1..l}) \quad e_\star \Downarrow \dot{w}_\star^{1..n} \\ \mathsf{mbody}(\mathsf{m}, C) = (\mathsf{x}_\star^{1..n})d \end{array}}{d\,\{\mathsf{this} \mapsto C(\dot{v}_\star^{1..l}),\, \mathsf{x}_\star \mapsto \dot{w}_\star^{1..n}\} \Downarrow q}\ (\mathsf{Rinvk})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..n}}{C(e_\star^{1..n}) \Downarrow C(\dot{v}_\star^{1..n})}\ (\mathsf{Rnew})$$

$$\frac{e \Downarrow C(\dot{v}_\star^{1..l}) \;\; C <: D \;\; e_1\{\mathsf{x} \mapsto C(\dot{v}_\star^{1..l})\} \Downarrow q}{e?\{\mathsf{x}\colon D \Rightarrow e_1\}/\{e_2\} \Downarrow q}\ (\mathsf{Rcst})$$

$$\frac{e \Downarrow \mathbf{null}\ \mathit{or}\ \left[e \Downarrow C(\dot{v}_\star^{1..l}) \;\; C \not<: D\right] \quad e_2 \Downarrow q}{e?\{\mathsf{x}\colon D \Rightarrow e_1\}/\{e_2\} \Downarrow q}\ (\mathsf{Rskp})$$

$$\frac{\begin{array}{c} e_\star \Downarrow \dot{v}_\star^{1..n} \quad \forall j < i\,.\, \dot{v}_\star^{1..n}; c_j \Downarrow \mathbf{reject} \\ \dot{v}_\star^{1..n}; c_i \Downarrow q \end{array}}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star\} \Downarrow q}\ (\mathsf{Rmch})$$

$$\frac{}{\mathbf{throw} \Downarrow \mathbf{throw}}\ (\mathsf{Cthr}) \qquad \frac{}{\mathbf{null} \Downarrow \mathbf{null}}\ (\mathsf{Rnul})$$

$$\frac{e \Downarrow \mathbf{throw}\ \mathit{or}\ e \Downarrow \mathbf{null}}{e.\mathsf{f} \Downarrow \mathbf{throw}}\ (\mathsf{Cfld}) \qquad \frac{e \Downarrow \mathbf{throw}\ \mathit{or}\ e \Downarrow \mathbf{null}}{e.\mathsf{m}(e_\star^{1..n}) \Downarrow \mathbf{throw}}\ (\mathsf{Crcv})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Downarrow \mathbf{throw}}{e.\mathsf{m}(e_\star^{1..n}) \Downarrow \mathbf{throw}}\ (\mathsf{Carg}) \qquad \frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Downarrow \mathbf{throw}}{C(e_\star^{1..n}) \Downarrow \mathbf{throw}}\ (\mathsf{Cnew})$$

$$\frac{e \Downarrow \mathbf{throw}}{e?\{\mathsf{x}\colon C \Rightarrow e_1\}/\{e_2\} \Downarrow \mathbf{throw}}\ (\mathsf{Ctst})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Downarrow \mathbf{throw}}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..k}\} \Downarrow \mathbf{throw}}\ (\mathsf{Cmch})$$

Figure 3.3: FPat Computation Rules

Acceptance $\quad \dot{v}_\star^{1..n}; c \Downarrow q \quad \dot{v} \curvearrowright p \dashv \sigma$

$$\frac{c \equiv \textbf{case}\ p_\star^{1..n} \Rightarrow b \qquad \dot{v}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..n} \quad b\,\sigma_\star^{1..n} \Downarrow q}{\dot{v}_\star^{1..n}; c \Downarrow q}\ \text{(mcase)}$$

$$\frac{}{\dot{v} \curvearrowright \mathsf{x} \dashv \{\mathsf{x} \mapsto \dot{v}\}}\ \text{(mvar)}$$

$$\frac{v.\mathsf{m}(\dot{u}) \Downarrow w \quad \mathsf{xtype}(\bullet, v, \mathsf{m}) = E \qquad \mathsf{casefld}(E, w) = \dot{w}_\star^{1..k} \quad \dot{w}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..k}}{\dot{u} \curvearrowright v.\mathsf{m}(p_\star^{1..k}) \dashv \sigma_\star^{1..k}}\ \text{(mextr)}$$

Rejection $\quad \dot{v}_\star^{1..n}; c \Downarrow \textbf{reject} \quad \dot{v} \curvearrowright p \dashv \textbf{reject}$

$$\frac{c \equiv \textbf{case}\ p_\star^{1..n} \Rightarrow b \qquad \dot{v}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..i-1} \quad \dot{v}_i \curvearrowright p_i \dashv \textbf{reject}}{\dot{v}_\star^{1..n}; c \Downarrow \textbf{reject}}\ \text{(rcase)}$$

$$\frac{v.\mathsf{m}(\dot{u}) \Downarrow \textbf{null}}{\dot{u} \curvearrowright v.\mathsf{m}(p_\star^{1..k}) \dashv \textbf{reject}}\ \text{(rnull)}$$

$$\frac{v.\mathsf{m}(\dot{u}) \Downarrow w \quad \mathsf{xtype}(\bullet, v, \mathsf{m}) = E \qquad \mathsf{casefld}(E, w) = \dot{w}_\star^{1..k} \quad \dot{w}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..i-1} \qquad \dot{w}_i \curvearrowright p_i \dashv \textbf{reject}}{\dot{u} \curvearrowright v.\mathsf{m}(p_\star^{1..k}) \dashv \textbf{reject}}\ \text{(rchild)}$$

Figure 3.4: FPat Acceptance and Rejection

Matching depends on judgments describing acceptance and rejection of patterns and cases, which are given in Figure 3.4. Rule (Rmch) describes evaluation of cases according to the first match policy: an accepting case is evaluated only if all preceding cases rejected the input.

Two separate judgments describe acceptance for cases $\dot{v}_\star^{1..n}; c \Downarrow q$ and patterns $\dot{v} \curvearrowright p \dashv \sigma$. We explain the judgments for case clauses first. A case accepts and evaluates to result q if each input value is accepted by the corresponding pattern (mcase). Analogously, a case rejects its input (written $\dot{v}_\star^{1..n}; c \Downarrow \mathbf{reject}$) if an initial segment of patterns accept and the following pattern rejects its input (rcase). Together, these rules describe a left-to-right evaluation of patterns. If a pattern accepts, it yields a substitution, and if all patterns accept, the substitutions obtained from the accepting patterns are all applied to the body of the case (the merging of substitutions is indicated by juxtaposition).

The judgment $\dot{v} \curvearrowright p \dashv \sigma$ describes that pattern p accepts $\dot{v}$ and yields substitution σ. Analogously, the judgment $\dot{v} \curvearrowright p \dashv \mathbf{reject}$ describes rejection. A variable pattern always accepts its input (mvar), yielding the obvious substitution. An extractor pattern is always of the form $C(\hat{v}_\star^{1..n}).\mathsf{m}(p_\star^{1..k})$ and accepts (mextr) if:

1. evaluation of the extractor call returns a value w, yielding so-called case fields $\dot{w}_\star^{1..k}$
2. all sub-patterns accept the case fields, yielding substitutions $\sigma_\star^{1..k}$

The extractor pattern rejects if the call returns **null** (rnull) or if one of its sub-pattern rejects its input (rchild). Case fields $\mathsf{casefld}(E, w)$ are determined for the return type E of the extractor method, as specified in the auxiliary judgment $\mathsf{xtype}(\bullet, v, \mathsf{m})$. They are the fields declared in the class definition of E itself, which thus serves as the "representation type".

The outcome is undefined when extractors throw exceptions. For this reason, the **@safe** annotation is required on any method that is referenced as an extractor. Safety is of course undecidable, but restrictions and approximations are available to tackle this problem. Our focus in this thesis is on justifying the condition, not checking it.

Discussion In any statically-typed definition of pattern matching, some mechanism has to be in place to derive the order and types of sub-patterns – with case classes and extractor we presented two choices. The benefit of using extractors lies in decoupling the matched type from the representation type. This leads to improved extensibility.

Pattern matching usually includes matching on literals like $42, \mathbf{true}$ and named constants like foo. While literal expressions are constructors without arguments, a corresponding convention for extractors can be assumed. We do not deal with named constants here, but only mention that they can be added to the formalization by allowing tests for singleton types $v.\mathsf{type}$ (structural equality would then ensure that $C(\dot{w}_\star^{1..k}) \in v.\mathsf{type}$ only if $v \equiv C(\dot{w}_\star^{1..k})$).

3.3 Typing

The FPat type system is specified through a set of typing rules in Figure 3.5, which add typing rules for match expressions and case clauses to the FJ type system. The rules are syntax-directed, so they can directly be used implement a type-checker pass of a compiler. The rules specific to matching are described in a separate section below.

Type judgments for expressions have the form $\Gamma \vdash e \in C$ where Γ is a type environment (a finite mapping from variables to types), e an expression and C a class. The judgments $cd \diamond$ and $an\ md\ \diamond\ \text{in}\ C$ assert well-typedness of class and method definitions. Methods annotated with **@safe** are assumed to terminate and never throw exceptions for any input (including **null**). We formally define the assumption expressed by the **@safe** modifier as follows. *Checking* this assumption is, in general, undecidable.

Def 1 *A method in a class C that is annotated with* **@safe** *satisfies the following property: For any sequence of optional values $\dot{v}_\star^{1..m}$ and any arguments $\dot{w}_\star^{1..n}$ such that $e = C(\dot{v}_\star^{1..m}).\mathsf{m}(\dot{w}_\star^{1..n})$ is a well-typed expression, there exists an optional value $\dot{w}$ with a derivation of $e \Downarrow \dot{w}$.*

A class definition is well-typed if all its methods are well-typed, and a method is well-typed if its return expression is well-typed under the appropriate type environment. If the method overrides a method in a superclass, their signatures have to be identical, which is asserted by the judgment $\mathsf{override}(an(B_\star^{1..n})B, \mathsf{m}, D)$. A program is well-typed if all its class definitions are well-typed, and its top-level expression is well-typed in the empty environment.

Typing expressions is straightforward. Rules (Tthr) and (Tnul) give the most specific type Exc to the **throw** and **null** results. (Tvar) takes the type of a variable from the type environment, and field access (Tfld) and object construction (Tnew) is checked against the fields of the class as calculated by the judgment $\mathsf{fields}(C)$. A similar judgment for method signatures $\mathsf{mtype}(\mathsf{m}, C)$ is used to type-check method invocation (Tinvk). Thus, well-typed method calls and objects constructions have the right number and types of arguments.

Test expressions are checked with rule (Ttst), which modifies the type environment for the succeeding branch to account for the new local variable. Binding in test expressions can be used to define a derived form $\mathbf{val}\ \mathsf{x}\colon C = a; b$, which we will introduce in Section 3.7.1.

It is notable here that we allow test expressions that are statically known to fail, such as $C(\ldots)?\{\mathsf{y}\colon D \Rightarrow \ldots\}/\{\ldots\}$ for unrelated C, D. The reason for not checking the static type against the type to be tested is that it would make it impossible to prove a substitution lemma (FJ has "stupid casts" for similar reasons): the expression to be tested might have a more precise type after substitution, and only then it would become apparent that the test expression fails. A real compiler would reject test expressions in source programs if it can derive at compile time that a test expression fails. Similar considerations apply to match expressions.

Expression Typing $\Gamma \vdash e \in C$

$$\frac{}{\Gamma \vdash \mathsf{x} \in \Gamma(\mathsf{x})}\ (\mathsf{Tvar})$$

$$\frac{}{\Gamma \vdash \mathbf{null} \in \mathsf{Exc}}\ (\mathsf{Tnul}) \qquad \frac{}{\Gamma \vdash \mathbf{throw} \in \mathsf{Exc}}\ (\mathsf{Tthr})$$

$$\frac{\Gamma \vdash e \in C \quad \mathsf{fields}(C) = \mathsf{f}_\star : C_\star^{1..n}}{\Gamma \vdash e.\mathsf{f}_i \in C_i}\ (\mathsf{Tfld})$$

$$\frac{\begin{array}{c}\mathsf{fields}(C) = \mathsf{f}_\star : D_\star^{1..n} \\ \Gamma \vdash e_\star \in C_\star^{1..n} \quad C_\star <: D_\star^{1..n}\end{array}}{\Gamma \vdash C(e_\star^{1..n}) \in C}\ (\mathsf{Tnew})$$

$$\frac{\begin{array}{c}\Gamma \vdash e \in A \\ \Gamma, \mathsf{x}: C \vdash a \in D \quad \Gamma \vdash b \in E\end{array}}{\Gamma \vdash e?\{\mathsf{x}: C \Rightarrow a\}/\{b\} \in D \sqcup E}\ (\mathsf{Ttst})$$

$$\frac{\begin{array}{c}\Gamma \vdash e \in C \quad \mathsf{mtype}(\mathsf{m}, C) = an(D_\star^{1..n})E \\ \Gamma \vdash e_\star \in C_\star^{1..n} \quad C_\star <: D_\star^{1..n}\end{array}}{\Gamma \vdash e.\mathsf{m}(e_\star^{1..n}) \in E}\ (\mathsf{Tinvk})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_\star \in C_\star^{1..n} \\ \Gamma; C_\star^{1..n} \vdash c_\star \in D_\star^{1..m}\end{array}}{\Gamma \vdash e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in \bigsqcup D_\star^{1..m}}\ (\mathsf{Tmch})$$

Figure 3.5: FPat Expression Typing Rules

Pattern and Case Typing $\Gamma; C_\star^{1..n} \vdash c \in D$
$\Gamma; C \ni p \dashv \Gamma'$

$$\frac{}{\Gamma; D \ni \mathsf{x} \dashv \mathsf{x}\colon D}\ (\mathsf{TPvar})$$

$$\frac{\begin{array}{c}\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}) = E \\ \mathbf{class}\ E(\mathsf{f}_\star\colon C_\star^{1..m})\ \triangleleft\ E'\ \{an_\star md_\star^{1..k}\} \\ \Gamma; C_\star \ni p_\star \dashv {\Gamma'_\star}^{1..m} \quad \Gamma' \equiv {\Gamma'_\star}^{1..m}\end{array}}{\Gamma; D \ni \hat{v}.\mathsf{m}(p_\star^{1..m}) \dashv \Gamma'}\ (\mathsf{TPext})$$

$$\frac{\begin{array}{c}\Gamma; C_\star \ni p_\star \dashv {\Gamma'_\star}^{1..n} \\ \Gamma, {\Gamma'_\star}^{1..n} \vdash b \in D\end{array}}{\Gamma; C_\star^{1..n} \vdash \mathbf{case}\ p_\star^{1..n} \Rightarrow b \in D}\ (\mathsf{Tcase})$$

Extractor Type $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m})$

$$\frac{\Gamma \vdash \hat{v} \in B \quad \mathsf{mtype}(\mathsf{m}, B) = @\mathbf{safe}(\mathsf{Obj})E}{\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}) = E}$$

Figure 3.6: FPat Pattern and Extractor Typing Rules

Method Typing $md\ \diamond\ \text{in}\ C$

$$\frac{\begin{array}{c}\mathsf{this}\colon C, \mathsf{x}_\star\colon C_\star^{1..n} \vdash e \in E \quad E <: B \\ \mathbf{class}\ C(\mathsf{f}_\star\colon D_\star^{1..m})\ \triangleleft\ D\ \{md_\star^{1..k}\} \\ \mathsf{override}(an(C_\star^{1..n})B, \mathsf{m}, D)\end{array}}{an\ \mathbf{def}\ \mathsf{m}(\mathsf{x}_\star\colon C_\star^{1..n})\colon B = \{e\}\ \diamond\ \text{in}\ C}$$

Class Typing $cd\ \diamond$

$$\frac{an_\star md_\star\ \diamond\ \text{in}\ C^{1..k}}{\mathbf{class}\ C(\mathsf{f}_\star\colon D_\star^{1..n})\ \triangleleft\ D\ \{an_\star md_\star^{1..k}\}\ \diamond}$$

Figure 3.7: FPat Method and Class Typing Rules

Field Lookup $\mathsf{fields}(C)$

$$\frac{}{\mathsf{fields}(\mathsf{Obj}) = \bullet}$$

$$\frac{\mathsf{fields}(D) = \mathsf{f}_\star : C_\star^{1..m} \quad \mathbf{class}\ C(\mathsf{g}_\star : D_\star^{1..n}) \lhd D\ \{an_\star md_\star^{1..k}\}}{\mathsf{fields}(C) = \mathsf{f}_\star : C_\star^{1..m}; \mathsf{g}_\star : D_\star^{1..n}}$$

Case Field Lookup $\mathsf{casefld}(C, v)$

$$\frac{\mathsf{fields}(D) = \mathsf{f}_\star : C_\star^{1..m} \quad \mathbf{class}\ C(\mathsf{g}_\star :\ D_\star^{1..n}) \lhd D\ \{an_\star md_\star^{1..k}\}}{\mathsf{casefld}(C, C(v_\star^{1..m+n})) = v_\star^{m+1..m+n}}$$

$$\frac{E \not\equiv C \quad \mathsf{fields}(D) = \mathsf{f}_\star : C_\star^{1..m} \quad \mathbf{class}\ C(\mathsf{g}_\star :\ D_\star^{1..n}) \lhd D\ \{an_\star md_\star^{1..k}\}}{\mathsf{casefld}(E, C(v_\star^{1..m+n})) = \mathsf{casefld}(E, D(v_\star^{1..m}))}$$

Figure 3.8: FPat Auxiliary Judgments for Field Lookup

Overriding $\mathsf{override}(an(B_\star^{1..n})B, \mathsf{m}, D)$

$$\frac{\mathsf{mtype}(\mathsf{m}, D) = an(B_\star^{1..n})B \text{ or undefined}}{\mathsf{override}(an(B_\star^{1..n})B, \mathsf{m}, D)}$$

Method Lookup $\mathsf{mtype}(\mathsf{m}, C)\ \mathsf{mbody}(\mathsf{m}, C)$

$$\frac{\mathbf{class}\ C(\mathsf{f}_\star : C_\star^{1..m}) \lhd D\ \{an_\star md_\star^{1..k}\} \quad an_i \equiv an \quad md_i \equiv \mathbf{def}\ \mathsf{m}(\mathsf{x}_\star : B_\star^{1..n}) :\ B = \{e\}}{\begin{array}{rl}\mathsf{mtype}(\mathsf{m}, C) & = an(B_\star^{1..n})B \\ \mathsf{mbody}(\mathsf{m}, C) & = (x_\star^{1..n})e\end{array}}$$

$$\frac{\mathbf{class}\ C(\mathsf{f}_\star : C_\star^{1..m}) \lhd D\ \{an_\star md_\star^{1..k}\} \quad \mathsf{m} \not\equiv md_\star^{1..k}}{\begin{array}{rl}\mathsf{mtype}(\mathsf{m}_i, C) & = \mathsf{mtype}(\mathsf{m}, D) \\ \mathsf{mbody}(\mathsf{m}, C) & = \mathsf{mbody}(\mathsf{m}, D)\end{array}}$$

Figure 3.9: FPat Auxiliary Judgments for Method Lookup

3.4 Typing of Match Expressions

Match expressions are well-typed if all their clauses are well-typed (Tmch), using the least upper bound to combine the result types of the case clauses. To type-check a single clause **case** $p_\star^{1..n} \Rightarrow b$ (Tcase), each pattern in $p_\star^{1..n}$ is type-checked w.r.t. the type environment Γ and an "expected type" $C_\star$, yielding a type context $\Gamma'_\star$ as in $\Gamma; C_\star \ni p_\star \dashv \Gamma'_\star{}^{1..n}$. Then, the body is type-checked against the combined type contexts as in $\Gamma, \Gamma'_\star{}^{1..n} \vdash b \in D$. Variables introduced in patterns must be pair-wise different and may not clash with Γ, which is implicit in the juxtaposition of environments.

Pattern typing $\Gamma; E \ni p \dashv \Gamma'$ is type-checked as follows: For variable patterns (TPvar), the expected type E is used to produce a singleton environment. For extractor patterns (TPextr), the judgment $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m})$ looks up the type of receiver and the signature of the extractor method in order to recover the representation type. It also ensures that extractors are **@safe**. The casefield types are then used as expected types to check the sub-patterns. Finally, the environments $\Gamma'_\star{}^{1..m}$ obtained from the sub-patterns are merged into one environment Γ'.

3.5 Divergent Programs

Proving type soundness for big-step semantics necessitates specifying in some way the meaningless programs ruled out by the type system. One possible approach is to define a special value **wrong** and characterize meaningless programs as those that are evaluated to **wrong** [1, pp.86]. This approach is error-prone, because if by mistake, a **wrong** rule is omitted, a misleading statement of "type soundness" is proven that does not actually exclude all meaningless programs.

A safer approach, suggested by Leroy and Grall [56], is to characterize meaningless programs as those that neither terminate nor diverge. A forgotten rule in the specification of divergent programs would then make the proof of the big-step version of the "Progress" lemma impossible. For our purpose, specifying divergent programs has the additional advantage of being relevant to correctness of pattern matching translation (we want to avoid translating divergent programs into terminating ones).

Coinduction is the dual of induction, a statement that can be made very precise: while an inductive definition is the least fixpoint of some equations, the coinductive type is their greatest fixpoint [42]. For use in proofs, it is helpful to think of induction as simply a process of "repeatedly adding new members to a set according to a set of rules", and coinduction as a process of "repeatedly taking away members from a set according to what's excluded by a set of rules, and seeing what's left" [84].

Divergent programs are defined coinductively by the set of divergence rules in Figure 3.10. These rules are tailored to help establishing that any well-typed term that does not terminate necessarily diverges. Their coinductive nature is indicated by horizontal double lines: coinductive derivations are "infinite" trees with the root being the assertion to derive and the successors of each node being determined by a derivation rule.

Let us consider the rules one by one. Rule (Dfld) and (Drcv) express that accessing a field or invoking a method of a divergent expression yields a divergent expression. Rules (Darg) and (Dnew) say that object construction and method invocation diverge if one of their arguments diverge. Note that a strict call-by-value, left-to-right evaluation order is followed also here. Rule (Dinvk) says that calling a method with arguments that make the method body diverge yields a divergent expression. Rules (Dtst), (Dcst) and (Dskp) characterize divergent test expressions by locating divergence in the respective subexpression.

A divergent match expression can be traced back to some (possibly empty) initial segment of rejecting cases and divergent case, and a divergent case can only diverge because its body diverges (Dbdy). Since extractors are assumed to be @**safe**, extractor calls themselves can neither diverge nor throw exceptions.

We give a small example to illustrates how the principle of coinduction is used in divergence proofs. Consider the following class definition and the instance of rule (Dinvk) that is a coinductive proof that expression $\mathsf{Lp}().\mathsf{m}()$ diverges:

```
class Lp {
  def m(): Lp = { this.m() }
}
```

$$\frac{\mathsf{Lp}() \Downarrow \mathsf{Lp}() \quad \mathsf{this}().\mathsf{m}()\{\mathsf{this} \mapsto \mathsf{Lp}()\} \in R}{\mathsf{Lp}().\mathsf{m}() \in R}$$

We pick the set $R = \{\mathsf{Lp}().\mathsf{m}()\}$ – this "guessing" of the initial set is characteristic for coinduction. In order to show that $R \subseteq \Uparrow$ it is enough to show that it is preserved by (any subset of) the divergence rules. A rule is "preserving" the set, if reinterpreting the conclusion as asserting membership in R (rather that in $\Uparrow$) entails the premises (under the same interpretation). Since $\Uparrow$ is defined coinductively (it is the largest possible set preserved by the divergence rules), the set R must be contained in $\Uparrow$.

In the example, for $\mathsf{Lp}().\mathsf{m}()$, we observe that rule (Dinvk) considered backwards only yields the very same expression, because $\mathsf{this}().\mathsf{m}()\{\mathsf{this} \mapsto \mathsf{Lp}\} \equiv \mathsf{Lp}().\mathsf{m}()$. Rule (Dinvk) thus preserves membership in R for $\mathsf{Lp}().\mathsf{m}()$. Since this is the only expression in R, we have shown that the whole set is preserved by the divergence rules. Consequently, $R \subseteq \Uparrow$ and $\mathsf{Lp}().\mathsf{m}()$ is a divergent program.

3.6 Soundness

We now prove type soundness using big-step versions of the standard lemmata.

Divergent Computation $e \Uparrow$

$$\frac{e \Uparrow}{e.\mathsf{f} \Uparrow}\ (\mathsf{Dfld}) \qquad \frac{e \Uparrow}{e.\mathsf{m}(e_\star^{1..n}) \Uparrow}\ (\mathsf{Drcv}) \qquad \frac{e \Downarrow v \quad e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Uparrow}{e.\mathsf{m}(e_\star^{1..n}) \Uparrow}\ (\mathsf{Darg})$$

$$\frac{\begin{array}{c} e \Downarrow C(\dot{v}_\star^{1..m}) \quad e_\star \Downarrow \dot{w}_\star{}^{1..n} \\ \mathsf{mbody}(\mathsf{m}, C) = (\mathsf{x}_\star^{1..n})b \\ b\{\mathsf{this} \mapsto C(\dot{v}_\star^{1..m}), \mathsf{x}_\star \mapsto \dot{w}_\star{}^{1..n}\} \Uparrow \end{array}}{e.\mathsf{m}(e_\star^{1..n}) \Uparrow}\ (\mathsf{Dinvk})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Uparrow}{C(e_\star^{1..n}) \Uparrow}\ (\mathsf{Dnew}) \qquad \frac{e \Uparrow}{e?\{\mathsf{x}\colon C \Rightarrow a\}/\{b\} \Uparrow}\ (\mathsf{Dtst})$$

$$\frac{e \Downarrow D(\dot{v}_\star^{1..n}) \quad D <: C \quad a\{\mathsf{x} \mapsto D(\dot{v}_\star^{1..n})\} \Uparrow}{e?\{\mathsf{x}\colon C \Rightarrow a\}/\{b\} \Uparrow}\ (\mathsf{Dcst})$$

$$\frac{e \Downarrow \mathbf{null}\ or\ [e \Downarrow D(\dot{v}_\star^{1..n}) \quad D \not<: C] \quad b \Uparrow}{e?\{\mathsf{x}\colon C \Rightarrow a\}/\{b\} \Uparrow}\ (\mathsf{Dskp})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Uparrow}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \Uparrow}\ (\mathsf{Dmch})$$

$$\frac{\begin{array}{c} e_\star \Downarrow \dot{v}_\star^{1..n} \quad \forall j < i\ .\ \dot{v}_\star^{1..n} \curvearrowright c_j \dashv \mathbf{reject} \\ \dot{v}_\star^{1..n}; c_i \Uparrow e \end{array}}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \Uparrow}\ (\mathsf{Dcase})$$

Divergent Cases and Patterns $\dot{v}_\star^{1..n}; c \Uparrow e$

$$\frac{c = \mathbf{case}\ p_\star^{1..n} \Rightarrow b \quad \dot{v}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..n} \quad b\,\sigma_\star^{1..n} \Uparrow}{\dot{v}_\star^{1..n}; c \Uparrow b\,\sigma_\star^{1..n}}\ (\mathsf{Dbdy})$$

Figure 3.10: FPat Divergence Rules

Lemma 1 (Uniqueness) *For all a, if $a \Downarrow q$ then for all q', if $a \Downarrow q'$ then $q = q'$.*

Proof By induction on $a \Downarrow q$ and case analysis on q'. □

Lemma 2 (Termination) *For all a and all q, it holds that if $a \Downarrow q$ then $a \not\Uparrow$.*

Proof By induction on $a \Downarrow q$ and inversion of $a \Uparrow$. We only show (Rfld).

Case $a \equiv e.\mathsf{f}$ (Rfld), (Dfld) Assume $a \Downarrow q$, then by (Rfld) $e \Downarrow v$. By i.h. $e \not\Uparrow$, so (Dfld) is not available. Hence $e.\mathsf{f} \not\Uparrow$. □

Lemma 3 (Subtypes have all Fields) *If $C <: D$, $C \neq \mathsf{Exc}$ then $\mathsf{fields}(C) = \mathsf{fields}(D); \mathsf{g}_\star : E_\star^{1..m}$.*

Proof By induction on the derivation of $C <: D$.

Case (Sobj) Then $\mathsf{fields}(\mathsf{Obj}) = \bullet$

Case (Sthr) Cannot happen

Case (Sref) Trivial

Case (Sext) Then the definition of fields is applied

Case (Stran) The i.h. is applied twice. □

Lemma 4 (Subtypes have all Methods) *If $C <: D$, $C \neq \mathsf{Exc}$ and $\mathsf{mtype}(\mathsf{m}, D) = an(C_\star^{1..n})B$, then $\mathsf{mtype}(\mathsf{m}, C) = an(C_\star^{1..n})B$.*

Proof By induction on the derivation of $\mathsf{mtype}(\mathsf{m}, D) = an(C_\star^{1..n})B$ and case analysis over $C <: D$.

Case (Sobj) Cannot happen, since Obj has no methods.

Case (Sthr) Cannot happen

Case (Sref) Trivial

Case (Sext) If C does not contain a definition for m, then the definition of mtype is applied. Otherwise, class definition of C is well-typed, thus $\mathsf{override}(\ an(C_\star^{1..n})B, \mathsf{m}, D)$ asserts that $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{mtype}(\mathsf{m}, D)$.

Case (Stran) The i.h. is applied twice □

The following two lemmata are needed to prove the substitution lemma for pattern matching expressions. We have to deal with the typing rule for variables which might end up producing a "better" environment for input values whose type has become more precise after substitution. We write $\Gamma' <: \Gamma$ when $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ and $\mathsf{x}\colon B \in \Gamma$ implies $\mathsf{x}\colon A \in \Gamma'$ with $A <: B$.

Lemma 5 (Subtypes yield Refined Environment)
If $C <: D$ and $\Gamma; D \ni p \dashv \Gamma'$ then $\Gamma; C \ni p \dashv \Gamma''$ for some $\Gamma'' <: \Gamma'$.

Proof By induction on $\Gamma; D \ni p \dashv \Gamma'$.

Case (TPvar) Then $\Gamma; D \ni \mathsf{x} \dashv \{\mathsf{x}\colon D\}$ and we can also derive $\Gamma; C \ni \mathsf{x} \dashv \{\mathsf{x}\colon C\}$. From $C <: D$ follows $\{\mathsf{x}\colon C\} <: \{\mathsf{x}\colon D\}$

Case (TPextr)

Then $\Gamma; D \ni \hat{v}.\mathsf{m}({p_\star}^{1..n}) \dashv {\Gamma'}_\star^{1..n}$ and subpatterns have derivations $\Gamma; D_\star \ni p_\star \dashv {\Gamma'_\star}^{1..n}$ for some casefield types ${D_\star}^{1..n}$. The expected type is not used for typing the subpatterns, thus the subderivations can be reused as is, yielding $\Gamma; C \ni \hat{v}.\mathsf{m}({p_\star}^{1..n}) \dashv {\Gamma'}_\star^{1..n}$. □

Lemma 6 (Refined Environment preserves Typing)
If ${C_\star <: D_\star}^{1..n}$ and $\Gamma, \mathsf{x}_\star\colon {D_\star}^{1..n} \vdash e \in B$ then $\Gamma, \mathsf{x}_\star\colon {C_\star}^{1..n} \vdash e \in A$ for $A <: B$.

Proof By straightforward induction on $\Gamma, \mathsf{x}_\star\colon {D_\star}^{1..n} \vdash e \in B$. □

Lemma 7 (Weakening) *If $\Gamma \vdash d \in S$ and $\mathsf{x} \notin fv(d)$, then $\Gamma, \mathsf{x}\colon T \vdash d \in S$ for any T.*

Proof Straightforward induction on $\Gamma \vdash d \in D$. □

Lemma 8 (Substitution Lemma) *If $\Gamma, \mathsf{x}_\star\colon {B_\star}^{1..n} \vdash b \in D$ and $\bullet \vdash \dot{u}_\star \in {A_\star}^{1..n}$ for ${A_\star <: B_\star}^{1..n}$, $\dot{u}_\star \in \mathsf{Values} \cup \{\mathbf{null}\}$ then $\Gamma \vdash b\{\mathsf{x}_\star \mapsto {\dot{u}_\star}^{1..n}\} \in C$, for $C <: D$.*

Proof By induction on the derivation of $\Gamma, \mathsf{x}_\star\colon {B_\star}^{1..m} \vdash b \in D$. Let $\sigma = \{\mathsf{x}_\star \mapsto {\dot{u}_\star}^{1..n}\}$ and $\Gamma' = \Gamma, \mathsf{x}_\star\colon {B_\star}^{1..m}$.

Case (Tvar) $b \equiv \mathsf{x}$

i) If $\mathsf{x} = \mathsf{x}_i$ for some i, then $\mathsf{x}\sigma = \dot{u}_i$ with $\Gamma \vdash \dot{u}_i \in A_i$ and $A_i <: B_i$ by assumption, **(Weakening)**.

ii) Otherwise, $\mathsf{x}\sigma = \mathsf{x}$ and rule (Tvar).

Case (Tthr),(Tnul) trivial because $b\,\sigma \equiv b$

Case (Tfld) $b \equiv e.\mathsf{f}$ We have $\Gamma' \vdash e \in E$ and i.h. yields $\Gamma \vdash e\sigma \in E'$ for $E' <: E$. By **(Subtypes have all Fields)** we have $\mathsf{fields}(E') = \mathsf{fields}(E); \mathsf{g}_\star : D_\star^{1..m}$ and (Tfld) finishes the case.

Case (Tinvk) $b \equiv e.\mathsf{m}(e_\star^{1..n})$ We have $\Gamma' \vdash e \in E$ and $\mathsf{mtype}(\mathsf{m}, E) = an(C_\star^{1..n})D$. The i.h. yields $\Gamma \vdash e\sigma \in E'$ for $E' <: E$. We also have $\Gamma' \vdash e_\star \in E_\star^{1..n}$ for $E_\star <: C_\star^{1..n}$ and i.h. yields $\Gamma \vdash e_\star\sigma \in {E'}_\star^{1..n}$ for $E'_\star <: E_\star^{1..n}$. By **(Subtypes have all Methods)**, we know $\mathsf{mtype}(\mathsf{m}, E') = \mathsf{mtype}(\mathsf{m}, E)$. Transitivity of <: and rule (Tinvk) finishes the case.

Case (Tnew) $b \equiv C(e_\star^{1..n})$ We have $\mathsf{fields}(C) = C_\star^{1..n}$ and $\Gamma' \vdash e_\star \in E_\star^{1..n}$ with $E_\star <: C_\star^{1..n}$. The i.h. yields $\Gamma \vdash e_\star\sigma \in {E'}_\star^{1..n}$ for $E'_\star <: E_\star^{1..n}$. Transitivity of <: and (Tnew) finish the case.

Case (Ttst) $b \equiv \dot{u}?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}$ We have $\Gamma' \vdash \dot{u} \in A$, $\Gamma' \vdash d \in E_1$, and $\Gamma' \vdash e \in E_0$ and $E_\star <: D^{0,1}$. The i.h. yields $\Gamma \vdash \dot{u}\sigma \in A'$, $\Gamma \vdash d\sigma \in E'_1$, and $\Gamma \vdash e\sigma \in E'_0$ with $A' <: A$, $E'_1 <: E_1$ and $E'_0 <: E_0$. Transitivity of <: and (Ttst) finishes the case.

Case (Tmch) $b \equiv e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ We have $\Gamma' \vdash e_\star \in C_\star^{1..n}$ and i.h. yields $\Gamma \vdash e_\star\sigma \in {C'}_\star^{1..n}$ for $C'_\star <: C_\star^{1..n}$.

For each $j \in 1..m$, let $c_j \equiv \mathbf{case}\ p_\star^{1..n} \Rightarrow b_j$ (we omit the extra j index for patterns). We have a case typing $\Gamma'; C_\star^{1..m} \vdash c_j \in D_j$ via $\Gamma'; C_\star \ni p_\star \dashv \Gamma''^{1..n}_\star$ and $\Gamma', \Gamma''_\star{}^{1..n} \vdash b \in D_j$.

By **(Subtypes yield Refined Environment)**, we get $\Gamma'; C'_\star \ni p_\star \dashv \Gamma'''_\star{}^{1..n}$ for $\Gamma'''_\star <: \Gamma''_\star{}^{1..n}$.

By **(Refined Environment preserves Typing)** we get $\Gamma, \Gamma'''^{1..n}_\star \vdash b_j \in D'_j$ for $D'_j <: D_j$.

Applying the i.h. yields $\Gamma, \Gamma'''_\star n \vdash b_j\sigma \in D''_j$ for $D''_j <: D'_j$.

For the combined lubs, we have $\bigsqcup D''^{1..m}_\star <: \bigsqcup D_\star^{1..m}$, and rule (Tmch) finishes the case.

□

Lemma 9 (Preservation)
If $a \Downarrow q$ and $\bullet \vdash a \in C$, then $\bullet \vdash q \in C'$ for some $C' <: C$.

Proof For $q \equiv \mathbf{throw}$ and $q \equiv \mathbf{null}$, rules (Tthr) and (Tnul) yield the proof. Otherwise, induction on $a \Downarrow v$.

Case (Rfld) $a \equiv e.\mathsf{f}_i$

The premises of (Tfld) are $\bullet \vdash e \in C_0$ and $\mathsf{fields}(C_0) = \mathsf{f}_\star : C_\star^{1..m}$ with $C = C_i$.

We have $e \Downarrow D(\dot{w}_\star^{1..n})$ and $v = w_i$.

By i.h. $\bullet \vdash D(\dot{w}_\star^{1..n}) \in D$ for $D <: C_0$.

By **(Subtypes have all Fields)**, we obtain $m \leq n$ and $f_i \in \mathsf{fields}(D)$.

Finally, from $\bullet \vdash D(\dot{w}_\star^{1..n}) \in D$ and rule (Tnew) we know $\bullet \vdash \dot{w}_i \in E_i$ with $E_i <: C_i$.

Case (Rinvk) $a \equiv e.\mathsf{m}(e_\star^{1..n})$

The premises of (Tinvk) are $\Gamma \vdash e \in E$, $\quad \mathsf{mtype}(\mathsf{m}, E) = an(C_\star^{1..n})C_0$, $\quad \bullet \vdash e_\star \in C_\star^{1..n}$.

Then $e \Downarrow D(\dot{w}_\star^{1..m})$, $e_\star \Downarrow \dot{v}_\star^{1..n}$, $\mathsf{mbody}(\mathsf{m}, D) = (\mathsf{x}_\star^{1..n})e_0$ with $\bullet \vdash e_0 \in C$. Under substitution $\sigma = \{\mathsf{this} \mapsto D(\dot{w}_\star^{1..m}), \mathsf{x}_\star \mapsto \dot{v}_\star^{1..n}\}$, the body evaluates as $e_0\,\sigma \Downarrow \dot{v}$.

Applying the i.h. for the receiver yields $D <: E$.

By **(Subtypes have all Methods)** we get $\mathsf{mtype}(\mathsf{m}, D) = \mathsf{mtype}(\mathsf{m}, E)$.

Applying the i.h. for the arguments yields $\bullet \vdash \dot{v}_\star \in C'^{1..n}_\star$ for $C'_\star <: C_\star^{1..n}$.

By **(Substitution Lemma)** we get $\bullet \vdash e_0\sigma \in C'$ for $C' <: C$.

Applying the i.h. for the body then yields $\bullet \vdash \dot{v} \in C''$ for $C'' <: C'$. Transitivity of subtyping finishes the case.

Case (Rnew) $a \equiv D(\dot{w}_\star^{1..n})$ then $a \Downarrow a$, and $D <: D$ by (Sref).

Case (Rcst) $a \equiv e?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}$

We have $e \Downarrow D(\dot{w}_\star^{1..n})$, $D <: C$ and $b\,\{\mathsf{x} \mapsto D(\dot{w}_\star^{1..n})\} \Downarrow v$.

Using typing premises from (Ttst), we apply the **(Substitution Lemma)** and then the i.h.

Case (Rskp) $a \equiv e?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}$

We have $e \Downarrow D(\dot{w}_\star^{1..n})$, $D \not<: C$ and $d \Downarrow v$.

Using typing premises from (Ttst), we apply the i.h. to d, yielding $\bullet \vdash d \in C'$.

Case (Rmch) $a \equiv e_\star^{1..m}\ \mathbf{match}\ \{c_\star^{1..l}\}$. Let i be the index of the matching case.

The premises of (Tmch) include $\bullet \vdash e_\star \in C_\star^{1..m}$ and case typing $\bullet; C_\star^{1..m} \vdash c_i \in D_i$ for $D_i <: C$.

The case typing has premises $\bullet; C_\star \ni p_\star \dashv \Gamma'^{\,1..n}_\star$ and $\bullet; \Gamma'^{\,1..m} \vdash b \in D_i$ where $c_i \equiv \mathbf{case}\, p_\star \Rightarrow b$.

We have $e_\star \Downarrow \dot{v}_\star^{1..m}$ as well as $\dot{v}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..m}$ and $b\,\sigma_\star^{1..m} \Downarrow \dot{v}$.

Applying the i.h. to $e_\star$ yields $\bullet \vdash \dot{v}_\star \in C'^{1..n}_\star$ for $C'_\star <: C_\star^{1..n}$.

By **(Substitution Lemma)**, $\bullet \vdash b\sigma_\star \in D'_i$ with $D'_i <: D_i$.

Applying the i.h. yields $\bullet \vdash \dot{v} \in D_i''$ with $D_i'' <: D_i'$.

This yields the desired type $C' = D_i'' <: \bigsqcup D_\star^{1..m} = C$, by transitivity of <: and properties of the least upper bound operator $\sqcup$. $\square$

We can now show a big-step version of the standard Progress lemma using coinduction. The proof is a straightforward adaptation from Leroy and Grall's proof for the simply-typed lambda calculus [56]. We will say that "rule ... preserves R" instead of Leroy and Grall's locution "applying the coinduction hypothesis" which seems to stem from the implementation of the coinduction proof principle in the COQ proof assistant.

Lemma 10 (Progress)
If $\bullet \vdash a \in C$ *and* $a \not\Downarrow q$ *for all* q*, then* $a \Uparrow$.

Proof By coinduction and case analysis over a. We recall the principle of coinduction: In order to show that $R = \{a \mid \text{for all } q \,.\, a \not\Downarrow q \text{ and } \bullet \vdash a \in C\}$ is included in $\Uparrow$, it suffices to show that R is preserved by the divergence rules. This means, if we replaced each assertion $a \Uparrow$ with $a \in R$ and assumed that the conclusion holds, we have to be able to show that the premises hold as well. To do this, we need proofs for $e \not\Downarrow$ for the subexpressions e of a. These can be obtained from inversion of "blocked" evaluation rules.

Case $a \in \{\mathbf{throw}, \mathbf{null}\}$ and $a \equiv \mathsf{x}$ are not interesting, since $a \notin R$

Case $a \equiv e_0.\mathsf{f}$ and (Rfld), (Cfld) are blocked. By $\bullet \vdash a \in C$ and (Tfld), we also have $\bullet \vdash e_0 \in C_0$. Thus, either

i) $e_0 \not\Downarrow q_0$ for any q_0. This amounts to $e_0 \in R$ and shows that (Dfld) preserves R.

ii) $e_0 \Downarrow \mathbf{throw}$ or $e_0 \Downarrow \mathbf{null}$, but this contradicts (Cfld) blocked.

iii) $e_0 \Downarrow D(\dot{w}_\star^{1..n})$ but by **(Preservation)** and **(Subtypes have all fields)**, this contradicts (Rfld) blocked.

Case $a \equiv e_0.\mathsf{m}(e_\star^{1..n})$ and (Rinvk),(Crcv) and (Carg) are blocked. By $\bullet \vdash a \in C$ and (Tinvk), we also have $\bullet \vdash e_0 \in C_0, \bullet \vdash e_\star \in C_\star^{1..n}$, $\mathsf{mtype}(\mathsf{m}, C_0) = (D_\star^{1..n})$ and $C_\star <: D_\star^{1..n}$.

Thus, either

i) $e_0 \not\Downarrow q_0$ for any q_0. This amounts to $e_0 \in R$ and shows that (Drcv) preserves R.

ii) $e_0 \Downarrow \mathbf{throw}$ or $e_0 \Downarrow \mathbf{null}$ but this contradicts (Crcv) blocked.

iii) $e_0 \Downarrow D(w_\star^{1..n})$. By **(Preservation)**, $D <: C_0$ and by **(Subtypes have all methods)**, $\mathsf{mbody}(\mathsf{m}, D) = (\mathsf{x}_\star^{1..n})b$. We can distinguish further

a) There exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \not\Downarrow q_0$ for any q_0. Then $e_i \in R$ and (Darg) preserves R.

b) There exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$, but this contradicts (Carg) blocked

c) $e_\star \Downarrow \dot{v}_\star^{1..n}$ and **(Preservation)** yields $\bullet \vdash v_\star \in D_\star^{1..n}$ for $D_\star <: C_\star^{1..n}$. Then let $\sigma = \{\mathsf{this} \mapsto D(w_\star^{1..m})\ \mathsf{x}_\star \mapsto v_\star^{1..n}\}$ and consider $b\sigma$. Either

1. $b\,\sigma \not\Downarrow q$ for any q. This amounts to $b\,\sigma \in R$ and shows that (Dinvk) preserves R.
2. $b\,\sigma \Downarrow q$, but this contradicts (Rinvk) blocked.

Case $a \equiv C(e_\star^{1..n})$ and (Rnew), (Cnew) are blocked. By $\bullet \vdash a \in C$ and (Tnew), we also have $\bullet \vdash e_\star \in A_\star^{1..n}$, $\mathsf{fields}(C) = B_\star^{1..n}$ and $A_\star <: B_\star^{1..n}$. Thus, either

i) there exists i with $e_\star \Downarrow v_\star^{1..i-1}$ and $e_i \not\Downarrow q$ for any q. Then $e_i \in R$ and (Dnew) preserves R

ii) there exists i with $e_\star \Downarrow v_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$, but this contradicts (Cnew) blocked.

iii) $e_\star \Downarrow v_\star^{1..n}$, but this contradicts (Rnew) blocked.

Case $a \equiv e?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}$ and (Rcst),(Rskp),(Ctst) are blocked. By $\bullet \vdash a \in C$ and (Ttst), we have all premises of the rule (Ttst). Thus either

i) $e \not\Downarrow q$ for any q, then $e \in R$ and (Dtst) preserves R.

ii) $e \Downarrow \mathbf{throw}$, but this contradicts (Ctst) blocked.

iii) $e \Downarrow D(\dot{v}_\star^{1..n})$. There are several subcases to consider:

a) $D <: C$, and for $\sigma = \{\mathsf{x} \mapsto D(\dot{v}_\star^{1..n})\}$, $b\,\sigma \not\Downarrow q$ for any q. Then $b\,\sigma \in R$ and (Dcst) preserves R.

b) $D <: C$, and for $\sigma = \{\mathsf{x} \mapsto D(\dot{v}_\star^{1..n})\}$, $b\,\sigma \not\Downarrow \mathbf{throw}$ but this contradicts (Rcst) blocked.

c) $D \not<: C$, and $d \not\Downarrow q$ for any q. Then $d \in R$ and (Dskp) preserves R.

Case $a \equiv e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ and (Rmch),(Cmch) are blocked.

By $\bullet \vdash a \in C$ and (Tmch), we have $\bullet \vdash e_\star \in A_\star^{1..n}$, for all j a case typing $\bullet; A_\star^{1..n} \vdash c_j \in D_j$, and for each body b_j a typing $\Gamma'_j \vdash b_j \in D_j$.

Thus, either

i) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \not\Downarrow q$ for any q. Then $e_i \in R$ and (Dmch) preserves R.

ii) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$ or $e_i \Downarrow \mathbf{null}$, but this contradicts (Cmch) blocked.

iii) $e_\star \Downarrow \dot{v}_\star^{1..n}$. Then we distinguish these cases:

a) if all cases reject, this contradicts that the last case always accepts.

b) There exists an i such that $\forall j < i \;.\; v_\star^{1..n}; c_j \Downarrow \mathbf{reject}$, $c_i = \mathbf{case}\ p_\star^{1..n} \Rightarrow b$ and $\dot{v}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..n}$. Then, either

1) $b\,\sigma_\star^{1..n} \not\Downarrow q$ for any q, then $b\,\sigma_\star^{1..n} \in R$ and (Dbdy), (Dcase) preserve R

2) $b\,\sigma_\star^{1..n} \Downarrow q$, which contradicts (Rmch) blocked.

Thus, R is preserved by all divergence rules, so $R \subseteq \Uparrow$.

□

Thm 1 (Type Soundness)
If $\bullet \vdash a \in C$ *then either* $a \Uparrow$ *or* $a \Downarrow q$ *for some* q *with* $\bullet \vdash q \in C'$, $C' <: C$.

Proof Consequence of **(Progress)** and **(Termination)**.

3.7 Optimizing Translation

In this section, we define and prove correct an optimizing translation from pattern matching expressions to decision-trees.

3.7.1 Rewriting Match Expressions

An elegant way to describe translation of match expressions is to give a set of rewrite rules, which are applied successively until all match expressions are replaced with lower-level operations. Apart from being easy to understand and implement, correctness can then be established for each rule separately.

There are two approaches to the compilation of pattern matching, one based on decision-trees and the other based on backtracking automata [33, 79]. We chose the translation to decision trees, which in the functional setting guarantees that no input value is tested more than once. Our presentation of the algorithm follows Pettersson's [73].

The central idea is to remove a top-level pattern of a case clause, lifting its sub-patterns to the top-level. Consider the two expressions in Figure 3.11. Inspecting their structure reveals

```
//we know                                        List().cons(x) ? { y: Cons ⇒
//xtype(Γ, List(), cons) = Cons                    (x, y.hd, y.tl) match {
x match {                                            case y1  , π1  , π2  ⇒ a
  case List().cons(π1, π2)        ⇒ a                case y0  , y2  , y3  ⇒ b
  case y0                         ⇒ b              }}/{
}                                                    x match {case y0 ⇒ b}
                                                   }
```

Figure 3.11: Rewriting a Nested Pattern

that they are actually equivalent (they will evaluate to the same result for all values that are substituted for x, z). The extractor of the first pattern $\mathsf{z.cons}(\pi_1, \pi_2)$ in the first case has been pulled out and a test is done on the outcome: if it is non-null, it is bound to the fresh variable y and the sub-patterns are matched against the case fields $\mathsf{y.hd}, \mathsf{y.tl}$. Note that the width of the original match is augmented by lifting the nested patterns to the top-level. Since π_1, π_2 can potentially reject the input, all cases of the original match are copied to the new one. Some entries need to be *expanded* to match the arity of the new match, which is done by using fresh variable patterns $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$. If the extractor returns **null**, the first clause rejects and so the second branch of the test expression deals with the remaining cases of the match.

It is convenient to treat match expression as a matrix of patterns and bodies, with the last column containing the bodies. This should be clear from the layout of the following match expression:

$$
\begin{array}{lllll}
e_\star^{1..n} \textbf{ match } & \{ & & & \\
\quad \textbf{case } p_{11}, & \ldots, & p_{1n} & \Rightarrow b_1 \\
\quad \textbf{case } p_{m1}, & \ldots, & p_{mn} & \Rightarrow b_m \\
\} & & & &
\end{array}
$$

This facilitates reasoning on the optimization that the algorithm performs by reusing results of an extractor call. It consists of replacing calls to the same extractor in the same column, but in different rows of the matrix with clauses that match the sub-patterns, in case the call succeeded. Likewise, if the result of the extractor call is **null**, all extractor patterns are discarded at the same time, instead of repeating the extractor call. We illustrate the optimization with an example. For simplicity, we omit the input values and only show the case clauses in Figure 3.14.

Here, the first and third case (on the left) test the same extractor $\mathsf{z.cons}$. This extractor call has been pulled out into a test expression. If it succeeds (then-branch), the resulting value is deconstructed and matched against the sub-patterns. Again, since patterns π_1, π_2 may fail, we include all other cases, but we do not need to repeat the extractor call. If the extractor call returns **null**, then (else-branch) the remaining test cases are those that have extractor patterns

Translation $[\![\Gamma \vdash a \ \mathbf{match}\ \{M\} \in D]\!] = e$

(Tmp)

$$[\![\ \frac{\ldots \quad \Gamma \vdash a_\star \in C_\star^{1..n}}{\Gamma \vdash a_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in D}\ ]\!] = \mathbf{val}\ \mathsf{z}_\star \colon C_\star = a_\star^{1..n}; \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$$

condition:
- $a_\star$ are not all variables

(Var)

$$[\![\ \frac{\ldots \quad \Gamma \vdash \mathsf{z}_\star \in C_\star^{1..n}}{\Gamma \vdash \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in D}\ ]\!] = b\,\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}$$

condition:
- c_1 has the shape $\mathbf{case}\ \mathsf{x}_\star^{1..n} \Rightarrow b$

(Mix)

$$[\![\ \frac{\ldots \quad \Gamma \vdash \mathsf{z}_\star \in C_\star^{1..n}}{\Gamma \vdash \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in D}\ ]\!] = \hat{v}.\mathsf{m}(\mathsf{z}_i)?\{\mathsf{y} \colon C \Rightarrow d\}/\{e\}$$

condition:
- c_1 has the shape $\mathbf{case}\ \mathsf{x}_\star^{1..i-1}\ \hat{v}.\mathsf{m}(p_\star'^{1..k})\ p_\star{}^{i+1..n} \Rightarrow b$

translation steps:

- $\hat{v}.\mathsf{m}(\mathsf{z}_i)$ has pattern typing $\dfrac{\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}) = C \quad \ldots}{\Gamma; C_i \ni \hat{v}.\mathsf{m}(p_\star'^{1..k}) \dashv \Delta}$ (TPext)
- the definition of C is $\mathbf{class}\ C(\mathsf{f}_\star \colon D_\star^{1..k}) \lhd E\ \{md_\star^{1..n}\}$
- $\mathsf{y}, \mathsf{y}_\star^{1..k}$ are fresh variables
- $d = \mathbf{val}\ \mathsf{y}_\star \colon D_\star = \mathsf{y}.\mathsf{f}_\star^{1..k}; (\mathsf{z}_\star, \mathsf{y}, \mathsf{y}_\star^{1..k}, \mathsf{z}_\star^{1..]i[..n})\ \mathbf{match}\ \{expand_{\hat{v}.m}(c_\star^{1..m})\}$
- $e = \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{other_{\hat{v}.m}(c_\star^{1..m})\}$

Figure 3.12: FPat Translation Rules

$$expand_{\hat{v}.\mathsf{m}}(\bullet) = \bullet$$

$$expand_{\hat{v}.\mathsf{m}}(\mathbf{case}\ p_\star\ \hat{v}.\mathsf{m}(p_\star^{1..k})\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}) =$$
$$\mathbf{case}\ p_\star\ \mathsf{z}'\ p'_1\ \cdots p'_k\ p_\star^{1..]i[..n} \Rightarrow b; expand_{\hat{v}.\mathsf{m}}(c_\star^{1..m})$$
$$\text{where } \mathsf{z}' fresh$$

$$expand_{\hat{v}.\mathsf{m}}(\mathbf{case}\ p_\star\ p\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}) =$$
$$\mathbf{case}\ p_\star\ p\ \mathsf{z}'_1\ \cdots \mathsf{z}'_k\ p_\star^{1..]i[..n} \Rightarrow b; expand_{\hat{v}.\mathsf{m}}(c_\star^{1..m})$$
$$\text{where } \mathsf{z}'^{1..k}_\star fresh, \quad p \neq \hat{v}.\mathsf{m}(p_\star^{1..k})$$

$$other_{\hat{v}.\mathsf{m}}(\bullet) = \bullet$$

$$other_{\hat{v}.\mathsf{m}}(\mathbf{case}\ p_\star\ \hat{v}.\mathsf{m}(p'^{1..k}_\star)\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}) = other_{\hat{v}.\mathsf{m}}(c_\star^{1..m})$$

$$other_{\hat{v}.\mathsf{m}}(\mathbf{case}\ p_\star\ p\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}) =$$
$$\mathbf{case}\ p_\star\ p\ p_\star^{1..]i[..n} \Rightarrow b; other_{\hat{v}.\mathsf{m}}(c_\star^{1..m})$$
$$\text{where } p \neq \hat{v}.\mathsf{m}(p_\star^{1..k})$$

Figure 3.13: Definitions of $expand_{\hat{v}.\mathsf{m}}$ and $other_{\hat{v}.\mathsf{m}}$

```
x match {
  case List().cons(π1, π2)  ⇒ a
  case List().nil()         ⇒ b
  case List().cons(π3, π4)  ⇒ d
  case y0                   ⇒ e
}
```

```
List().cons(x)?{y : Cons ⇒
  (x, y.hd, y.tl) match {
  case y1            , π1   , π2   ⇒ a
  case List().nil()  , y2   , y3   ⇒ b
  case y4            , π3   , π4   ⇒ d
  case y0            , y5   , y6   ⇒ e
}}/{
  x match {
  case List().nil()  ⇒ b
  case y0            ⇒ e
}}
```

Figure 3.14: Optimization of Extractor Calls

other than z.cons. This suggests an iterative algorithm that identifies common patterns and translates them into test expressions and new, smaller match expressions.

Figure 3.12 contains the rewrite rules used by the algorithm. The translation relies on the static types of expressions, and is thus expressed as a translation of type derivations. The rules use a derived form $\mathbf{val}\ \mathsf{x}\colon C = a; b$ which has the double purpose of simplifying the presentation and catching divergent and exception-throwing input values. The derived form is only used when a is of static type C, and abbreviates $a?\{\mathsf{x}\colon C \Rightarrow b\}/\{b'\}$ where $b' = b\,\{\mathsf{x} \mapsto \mathbf{null}\}$.

Rule (Tmp) introduces **val** definitions, so that input values are always variables. Rule (Var) handles matches that are known to accept. The essential rule is (Mix) which performs the optimizing translation described above. If the extractor returns value w, then the sub-values $\mathsf{casefld}(C, w)$ can be obtained with field accesses $w.\mathsf{f}_\star^{1..k}$, and the return value as well as the sub-values are bound to fresh local variables $\mathsf{y}, \mathsf{y}_\star^{1..k}$. The function $expand$ adapts the width of case clauses as mentioned before. If the extractor returns **null**, we continue matching on those clauses that have a different extractor, computed by function $other$.

In contrast to functional pattern matching, we cannot assume that e.g. a rejecting extractor cons means that nil will necessarily accept the input value. The user-defined methods could be annotated to supply this information, an extension that we do not pursue in this thesis.

3.7.2 Why must extractors be @safe?

Recall the example above. Suppose π_1 was a variable pattern and π_2, π_4 test the same extractor. Optimizing for the failing pattern π_2 causes omission of the entire third case clause.

When omitting this case clause, we are already assuming that π_3 will either accept or reject its input. However, if π_3 were allowed to throw an exception or diverge, it would not be possible to omit its evaluation without changing the meaning of the program. For this reason, the semantics does not cover these anomalous situations (if we included them, we could not prove our optimization correct). Any semantics for pattern matching that involves user-defined code depends on this assumption if optimized translation of matching is to preserve the meaning of programs, since we usually do not expect divergent or exception throwing programs to turn into normally terminating ones.

The assumption that extractors are **@safe** complements the assumptions formulated by Syme, Neverov and Margetson [87] and Okasaki [71] that informally require extractors to be side-effect free and return the same result in all execution contexts in order for optimization to work. Of course in this calculus, absence of side-effects is guaranteed by the absence of assignment. We do not discuss checking absence of side-effects here. In the ML language,

$$
\begin{array}{ll}
\mathit{transform}(\mathbf{null}) & = \mathbf{null} \\
\mathit{transform}(\mathsf{x}) & = \mathsf{x} \\
\mathit{transform}(e.\mathsf{f}) & = \mathit{transform}(e).\mathsf{f} \\
\mathit{transform}(e.\mathsf{m}({e_\star}^{1..n})) & = \mathit{transform}(e).\mathsf{m}({e'_\star}^{1..n}) \\
 & \qquad \text{where } e'_\star = \mathit{transform}(e_\star)^{1..n} \\
\mathit{transform}(\mathbf{throw}) & = \mathbf{throw} \\
\mathit{transform}(a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}) & = a'?\{\mathsf{x}\colon C \Rightarrow d'\}/\{e'\} \\
 & \qquad \text{where } a' = \mathit{transform}(a) \\
 & \qquad \text{where } d' = \mathit{transform}(d) \\
 & \qquad \text{and } e' = \mathit{transform}(e) \\
 & \\
\mathit{transform}({e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}) & = \mathit{transform}(\mathit{rewrite}({e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}))
\end{array}
$$

Figure 3.15: The *transform* function

which has a tradition of offering references and side-effecting computation alongside functional programming, the absence of automatic checks is compensated by the rigorous specification: programmers understand in which situations they better not rely on side-effects. For SCALA, we adopt a similar approach with the contention that such a compromise would work equally well in the object-oriented setting.

3.7.3 The Algorithm

We define a function *transform* in Figure 3.15 that recursively traverses expressions, rewriting any match statements it finds.

The *transform* function is then naturally extended to method definitions and class definitions. A program is translated by translating all class definitions and the top-level expression. Note that a single application of a rewrite rule takes place in one of the following contexts:

Def 2 (Target Context) *A target context is defined by the following grammar:*

$$
\begin{array}{rcl}
\xi, \zeta & ::= & [\,] \mid \xi.\mathsf{f} \mid \xi.\mathsf{m}({b_\star}^{1..n}) \mid a.\mathsf{m}(b_\star, \xi, {b_\star}^{1..]i[..n}) \\
 & \mid & \xi?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\} \mid a?\{\mathsf{x}\colon C \Rightarrow \xi\}/\{e\} \mid a?\{\mathsf{x}\colon C \Rightarrow d\}/\{\xi\}
\end{array}
$$

By the reasoning in the next section, this rewrite preserves the meaning of the program. A subsequent call of *transform* performs the same for subexpressions of a', until all match expressions are translated away.

3.8 Correctness of the Translation

We define a formal notion of equivalence. Recall that a substitution always satisfies $\mathsf{x}\sigma \equiv$ **null** or $\mathsf{x}\sigma \in \mathsf{Values}$ for all $\mathsf{x} \in \mathsf{dom}(\sigma)$. We proceed in two steps, following the *démarche* of Ma [59]: we define a notion of equivalence and show that it is preserved by target contexts. Then we show that an expression is equivalent to its translation.

3.8.1 Equivalence and Open Equivalence

Def 3 (Equivalence) *For d, e expressions with $fv(d) = fv(e) = \emptyset$, d is equivalent to e (written $d \approx e$), if both of these conditions hold: 1. for all q, if $d \Downarrow q$ then $e \Downarrow q$, and 2. if $d \Uparrow$ then $e \Uparrow$.*

Showing that $\approx$ is an equivalence relation is easy using **(Uniqueness)**, **(Termination)**. Equivalence alone is not enough for our purpose, since rewrite rules take place in context. We now define an equivalence on open terms and show it is stable under contexts.

Def 4 (Open Equivalence) *For expressions d, e with $fv(d) \cup fv(e) \subseteq X$, d is open-equivalent to e (written $X \Vdash d \approx e$) if $d\sigma \approx e\sigma$ for all substitutions σ with $X \subseteq \mathsf{dom}(\sigma)$.*

Lemma 11 (Substitution preserves Equivalence) *If $X \Vdash d \approx e$, then for any substitution σ with $\mathsf{dom}(\sigma) \subseteq X$, it holds that $X \backslash \mathsf{dom}(\sigma) \Vdash d\sigma \approx e\sigma$.*

Proof Let $X \Vdash d \approx e$ and σ be a substitution. We have to show that for any substitution ρ with $\mathsf{dom}(\rho) = X \backslash \mathsf{dom}(\sigma)$, it holds that $(d\sigma)\rho \approx (e\sigma)\rho$. Considering that substitution is associative, this amounts to $d(\sigma\rho) \approx e(\sigma\rho)$. We observe that $\sigma\rho$ is a substitution with $X = \mathsf{dom}(\sigma\rho)$, and the equivalence follows from $X \Vdash d \approx e$. □

Def 5 (Derived Form for Value Definition) *The expression form* **val** $\mathsf{x}\colon C = a; d$ *abbreviates* $a?\{\mathsf{x}\colon C \Rightarrow d\}/\{d\{\mathsf{x} \mapsto \mathbf{null}\}\}$ *and is typed according to the scheme*

$$\frac{\Gamma \vdash a \in A \qquad \Gamma, \mathsf{x}\colon A \vdash d \in D \qquad \Gamma \vdash d\{\mathsf{x} \mapsto \mathbf{null}\} \in D'}{\Gamma \vdash a?\{\mathsf{x}\colon C \Rightarrow d\}/\{d\{x \mapsto \mathbf{null}\}\} \in D} \text{(Ttst)}$$

With **(Substitution Lemma)**, it is easy to see that $D' <: D$ and thus $D \sqcup D' = D$

Lemma 12 (ValDef Equivalences) *Let $b \equiv$* **val** $\mathsf{x}\colon A = a; d$ *where $\Gamma \vdash a \in A$ and σ some substitution that agrees with Γ. Then*

$$\begin{array}{rlll} I. & \text{If} & a\sigma \Downarrow w & \text{then } b\,\sigma \approx d\{\mathsf{x} \mapsto w\}\sigma. \\ II. & \text{If} & a\sigma \Downarrow \mathbf{null} & \text{then } b\,\sigma \approx d\{\mathsf{x} \mapsto \mathbf{null}\}\sigma. \\ III. & \text{If} & a\sigma \Downarrow \mathbf{throw} & \text{then } b\,\sigma \Downarrow \mathbf{throw}. \\ IV. & \text{If} & a\sigma \Uparrow & \text{then } b\,\sigma \Uparrow . \end{array}$$

Proof (Sketch) We note that by **(Substitution Lemma)** and **(Preservation)**, the type test cannot fail except for **null**. With this observation, I. follows from (Rcst), II. from (Rskp), III. from (Ctst) and IV. from (Drcv). □

Def 6 (Translation) *For* $a \equiv e_\star^{1..n}$ **match** $\{c_\star^{1..m}\}$ *with* $\Gamma \vdash a \in C$*, the translation* $[\![\Gamma \vdash a \in C]\!]$ *is defined as the application of a suitable rule in Fig. 3.12.*

Thm 2 (Congruence) *If* $X \Vdash d \approx e$*, then* $Y \Vdash \xi[d] \approx \xi[e]$ *for* $Y = fv(\xi[d]) \cup fv(\xi[e])$*.*

Proof By induction on ξ. For each context shape, we consider possible terminating and divergent computations under a value substitution σ.

Case $\xi \equiv [\,]$ then $Y = X$ and $X \Vdash d \approx e$ by assumption.

Case $\xi \equiv \zeta.\mathsf{f}$

i) $(\zeta[d].\mathsf{f})\sigma \Downarrow \mathbf{throw}$ by (Cfld). Then, we have $\zeta[d]\sigma \Downarrow \mathbf{throw}$ or $\zeta[d]\sigma \Downarrow \mathbf{null}$. By i.h., we obtain $\zeta[e]\sigma \Downarrow \mathbf{throw}$ (resp. $\zeta[e]\sigma \Downarrow \mathbf{null}$) and $(\zeta[e].\mathsf{f})\sigma \Downarrow \mathbf{throw}$ by (Cfld).

ii) $(\zeta[d].\mathsf{f})\sigma \Downarrow w$ by rule (Rfld). Then, we have $\zeta[d]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ and $\mathsf{f}\colon D \in \mathsf{fields}(C)$. By the i.h., we obtain $\zeta[e]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ and $(\zeta[e].\mathsf{f})\sigma \Downarrow w$ by (Rfld).

iii) $(\zeta[d].\mathsf{f})\sigma \Uparrow$ by (Dfld). Then $\zeta[d]\sigma \Uparrow$, by i.h. $\zeta[e]\sigma \Uparrow$ and $(\zeta[e].\mathsf{f})\sigma \Uparrow$ by (Dfld)

Case $\xi \equiv \zeta.\mathsf{m}(b_\star^{1..n})$

i) $(\zeta[d].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow \mathbf{throw}$ by (Crcv). Then $\zeta[d]\sigma \Downarrow \mathbf{throw}$ or $\zeta[d]\sigma \Downarrow \mathbf{null}$. By i.h. $\zeta[e]\sigma \Downarrow \mathbf{throw}$ (resp. $\zeta[e]\sigma \Downarrow \mathbf{null}$) and $(\zeta[e].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow \mathbf{throw}$ by (Crcv).

ii) $(\zeta[d].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow \mathbf{throw}$ by (Carg). Then $b_i\,\sigma \Downarrow \mathbf{throw}$ or $b_i\,\sigma \Downarrow \mathbf{null}$ and $\zeta[e].\mathsf{m}(b_\star^{1..n}) \Downarrow \mathbf{throw}$ by (Carg).

iii) $(\zeta[d].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow q$ by (Rinvk). Then $\zeta[d]\sigma \Downarrow C(\dot{v}_\star^{1..m})$. By i.h., $\zeta[e]\sigma \Downarrow C(\dot{v}_\star^{1..m})$ and by (Rinvk), $\zeta[e].\mathsf{m}(b_\star^{1..n}) \Downarrow q$.

Case $\xi \equiv a.\mathsf{m}(b_\star, \zeta, b_\star^{1..]i[..n})$

This case is similar to the preceding case.

Case $\xi \equiv \zeta?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\}$

i) $(\zeta[d]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow \mathbf{throw}$ by (Ctst). Then $\zeta[d]\sigma \Downarrow \mathbf{throw}$. By i.h. $\zeta[e]\sigma \Downarrow \mathbf{throw}$ and $(\zeta[e]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow \mathbf{throw}$ by (Ctst).

ii) $(\zeta[d]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rcst). Then $\zeta[d]\sigma \Downarrow E_0(\dot{v}_\star^{1..n})$ for some $E_0 <: E$ and $a\sigma\{\mathsf{x} \mapsto E_0(\dot{v}_\star^{1..n})\} \Downarrow q$. The i.h. yields $(\zeta[e])\sigma \Downarrow E_0(\dot{v}_\star^{1..n})$ and $(\zeta[e]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rcst).

iii) $(\zeta[d]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rskp). Then we either have $\zeta[d]\sigma \Downarrow \mathbf{null}$ or $\zeta[d]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ for some $C \not<: E$, and $b\sigma \Downarrow q$. The i.h. yields $\zeta[e]\sigma \Downarrow \mathbf{null}$ resp. $\zeta[e]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ and $(\zeta[e]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rskp).

Case $\xi \equiv a?\{\mathsf{x}\colon E \Rightarrow \zeta\}/\{b\}$

This case is similar to the preceding case, with i.h. applied with $\sigma\{\mathsf{x} \mapsto w\}$ for (Rcst).

Case $\xi \equiv a?\{\mathsf{x}\colon E \Rightarrow b\}/\{\zeta\}$

This case is similar to the preceding case.

□

The following lemma is an essential part of the correctness theorem and shows the equivalence of a match expression and the test expressions where the relevant extractor call is "pulled out".

Lemma 13 (Split) *Let* $a \equiv \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ *with* $\Gamma \vdash a \in A$,

with $c_1 = \mathbf{case}\ \mathsf{x}_\star^{1..i-1}\ \hat{v}.\mathsf{m}(\pi_\star^{1..k})\ p_\star^{i+1..n} \Rightarrow b$,
and $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}) = @\mathbf{safe}(\mathsf{Obj})C$.
For any substitution σ it holds that :

$I.$ $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow w$ implies $a\sigma \approx (\mathsf{z}_\star, \mathsf{z}_i,\ \mathsf{w}_\star^{1..k}, \mathsf{z}_\star^{1..]i[..n}\ \mathbf{match}\ \{expand_{\hat{v}.m}(c_\star^{1..m})\})\sigma$
$II.$ $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow \mathbf{null}$ implies $a\sigma \approx (\mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{other_{\hat{v}.m}(c_\star^{1..m})\})\sigma$

Proof *I.* Let $a' \equiv (\mathsf{z}_\star, \mathsf{z}_i,\ \mathsf{w}_\star^{1..k}, \mathsf{z}_\star^{1..]i[..n}\ \mathbf{match}\ \{expand_{\hat{v}.m}(c_\star^{1..m})\})\sigma$.

Terminating computation $a\sigma \Downarrow q$ can only happen through (Rmch).

We show pattern acceptance and rejection coincides in a and a'.

Let $\mathsf{z}_\star\sigma^{1..n}; c_j\sigma \Downarrow \mathbf{reject}$. Then acceptance as well as rejection was established to the left of column i, in column i, or to the right of column i of the original match in a.

Patterns to the left of i are not changed by *expand*. Patterns to the right of i are merely moved to index $i + k$, but test the same input values.

- If in clause c_j a pattern in column i is of the form $\hat{v}.\mathsf{m}(\pi'^{1..k}_\star)$ for some $\pi'^{1..k}_\star$, the function *expand* lifts patterns $\pi'^{1..k}_\star$ appear in c'_j, to be matched against $w_\star^{1..k}$.

The same derivations for acceptance and rejection can be reused: whenever acceptance by (mextr) is derived with $w_\star \curvearrowright \pi'_\star \dashv \rho_\star{}^{1..k}$ the corresponding $\rho_\star{}^{1..k}$ are also obtained in c'_j. Moreover, whenever rejection is derived through (rchild), then $w_h \curvearrowright \pi'_h \dashv \mathbf{reject}$ for some h can be also be derived in c'_j. The additional variable pattern at position i does not affect the outcome, since it was chosen fresh.

- If in clause c_j, a pattern with a different extractor appears in column i, then the outcome is obviously the same. The additional variable patterns in columns $i+1..i+k$ act as dummy patterns for discarded input values $w_\star{}^{1..k}$.

Divergent computation $a\sigma \Uparrow$ is only derivable with (Dcase), and (Dbdy). By the same reasoning as above, patterns in c'_j have the same acceptance/rejection behavior as those in c_j. So the matching case c_i in (Dbdy) produces the same substitution ρ that makes the body diverge.

Proof *II.* Let $a' \equiv (\mathsf{z}_\star{}^{1..n}\ \mathbf{match}\ \{other_{\hat{v}.m}(c_\star{}^{1..m})\})\sigma$.

The hypothesis is enough to derive pattern rejection by (rnull). It is clear that this rejection judgment for column i causes all c_j with the same $\hat{v}.\mathsf{m}(\pi'^{1..k}_\star)$ in column i to reject input values $\mathsf{z}_\star{}^{1..n}\sigma$. For a formal proof, this is not enough, so we argue that we can actually produce a proof of case rejection for every such c_j.

To this end, we have to look at all patterns to the left of column i. It is important to note that for each such pattern, we can derive either an acceptance or a rejection judgment: First of all, the syntax allows only extractor patterns $C(\hat{v}_\star{}^{1..l}).\mathsf{m}(\ldots)$, so no null dereferencing can occur. Furthermore, **by condition "@safe" divergent patterns and exceptions in patterns are ruled out**. An inductive argument is then applied to each extractor pattern in column $l < i$ to derive either an acceptance or a rejection judgement: Every extractor call of terminates normally (guaranteed by **@safe**), yielding (so we stop with a rejection judgment) or a value w, whose case-fields are used for the remaining subpatterns (with the induction hypothesis yielding acceptance or rejection of subpatterns, which is the used to derive acceptance or rejection for the current pattern). So every pattern in column l on the top-level will either accept or reject z_l.

Now, a sequence of acceptance judgments followed by a rejection judgment in input z_l for $l < i$ yields the desired case rejection for c_j. If instead all patterns to the left of colum i accept, then we use rejection of $\hat{v}.\mathsf{m}(\pi'^{1..k}_\star)$ to derive rejection of the whole case clause.

Putting it all together, we conclude that omitting these case clauses will not alter the behavior of a. The expression a' uses $other$ to omit exactly these cases, and will contain only those cases that still need to be tested for acceptance or rejection (with at least one case always accepting, by convention). Thus, every evaluation involving (Rmch) (which involves one crucial pattern acceptance judgment) for a can be simulated by a corresponding evaluation involving (Rmch) for a'. □

We now have everything we need for proving the correctness theorem. Since equivalence is a congruence, it is enough to show correctness of the rewrite rules.

Thm 3 (Correctness of $[\![\,]\!]$) *For $a \equiv a_\star^{1..n}$ $\mathbf{match}$ $\{c_\star^{1..m}\}$, $fv(a) = X$, typing $\Gamma \vdash a \in A$, translation $a' = [\![\Gamma \vdash a \in A]\!]$ it holds that $X \Vdash a \approx a'$.*

Proof By case distinction on $[\![\Gamma \vdash a \in C]\!]$. Let σ be a substitution that closes a.

Case (Tmp) Let σ be a substitution with $X \subseteq \mathsf{dom}(\sigma)$. Then **(ValDef Equivalences)** yields the desired result.

Case (Var) $a = \mathsf{z}_\star^{1..n}$ $\mathbf{match}$ $\{c_\star^{1..m}\}$ and $a' = b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}$

where $c_1 = \mathbf{case}\ \mathsf{x}_\star^{1..n} \Rightarrow b$

Since $\mathsf{z}_\star\sigma \in \mathsf{Values} \cup \{\mathbf{null}\}$, we can ignore (Cmch),(Dmch). By (mcase) and (mvar), the first row matches, yielding $\rho = \{\mathsf{x} \mapsto \mathsf{z}\sigma\}$. We then exploit that $b\,\rho \equiv b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}\sigma$

i) $a\sigma \Downarrow q$ by (Rmch). Then $b\rho \Downarrow q$ thus $b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}\sigma \Downarrow q$

ii) $a\sigma \Uparrow$ by (Dcase), (Dbdy). Then $b\rho \Uparrow$ thus $b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}\sigma \Uparrow$

Case (Mix) $a = \mathsf{z}_\star^{1..n}$ $\mathbf{match}$ $\{c_\star^{1..m}\}$ and $\hat{v}.\mathsf{m}(\mathsf{z}_i)?\{\mathsf{x}\colon B \Rightarrow d\}/\{e\}$

where $c_1 = \mathbf{case}\ \mathsf{x}_\star^{1..i-1}\hat{v}.\mathsf{m}(p_\star^{1..k})\ p_\star^{i+1..n} \Rightarrow b$ and d, e as described in Figure 3.12.

Since $\mathsf{z}_\star\sigma \in \mathsf{Values} \cup \{\mathbf{null}\}$, we can ignore (Cmch),(Dmch). We distinguish two cases.

- $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow w$. Then by **(Preservation)**, (Rcst), we get $a'\sigma \approx d\sigma$. We finish the case with Lemma **(Split)** and transitity of $\approx$.
- $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow \mathbf{null}$. Then by (Rskp), $a'\sigma \approx e\sigma$. We finish the case with Lemma **(Split)** and transitity of $\approx$.

□

Corollary 1 (Complete Algorithm) *The algorithm described in Section 3.7.3 is correct.*

Proof Consequence of the above theorem, applied sequentially to every application of a rewrite rule $[\![\,]\!]$, and transitivity of $\approx$. For termination, observe that each match expression produced by a rewriting rule is smaller than the original match expression using the lexicographically ordered tuples $\langle i, j, k\rangle$ where i is the number of non-variable input values, j the number of case clauses, and k the number of extractor patterns in ${c_\star}^{1..j}$. This ordering shows that for any e, all chains of dependency-pairs $\langle transform(e), transform(rewrite(e))\rangle$ must be finite. □

Using dependency-pairs [40] seems necessary, since the mix rules creates *two* new match expressions which complicates finding a suitable simplification order that only compares expressions before and after application of a rewrite rule.

3.9 Summary

In this chapter, we explored the formal underpinnings of object-oriented pattern matching based on extractors. The development uses the fact that pattern matching expressions can be considered as matrices regrouping the patterns and bodies of the case clauses, which can be broken down in smaller matrices by identifying common tests and translating them step by step. The formal development proved useful in that we have motivated important restrictions that must hold in order for optimizing translation of extractor-based pattern matching to be semantics-preserving. Namely, extractors have to be implemented with terminating expressions, that are moreover guaranteed to not throw any exceptions. The modular nature of this formal development in the form of rewrite rules makes it a suitable basis for implementation. However, a real language implementation like SCALA contains several pattern forms for syntactic convenience, which have some interaction and sometimes new opportunities for optimization. These aspects will be described in the next chapter.

Chapter 4

Implementation

When implementing pattern matching in a compiler for a realistic programming language, the theoretical model of Chapter 3 is modified: on the one hand, we introduce extensions to pattern matching that increase programmer convenience, on the other hand, we adapt translation to the available target instructions in order to help efficient code generation. This chapter deals with implementation issues, based on experiences from implementing pattern matching in the SCALA compiler.

The SCALA compiler translates an industrial-strength high-level language to a low-level language interpreted the JVM virtual execution environment. It is reasonable to expect that our results can be used to extend other realistic languages with an object-oriented pattern matching construct.

We will describe type patterns (and other extensions) in isolation, but give no formal description of the entire system. The reason for this is that more extensions are added later, and the interactions hold no theoretical insights. The basis for these extensions is rooted in the engineering perspective of improving efficiency and readability. In contrast, type patterns do hold new theoretical insights when *generic* patterns are considered in Chapter 6.

Thus, we describe the implementation semi-formally. We give a rewrite rule that is tailored to match expressions that contain a single column of type patterns. The interactions that arise in the interplay of type patterns, case classes and extractors is highlighted in prose, which should be sufficient for the reader interested in reconstructing the whole translation algorithm from the parts that described here.

4.1 The Scala Language and Compiler

Object-oriented pattern matching is fully implemented in the reference SCALA compiler, including many extensions that we will only mention in passing but which should be obvious

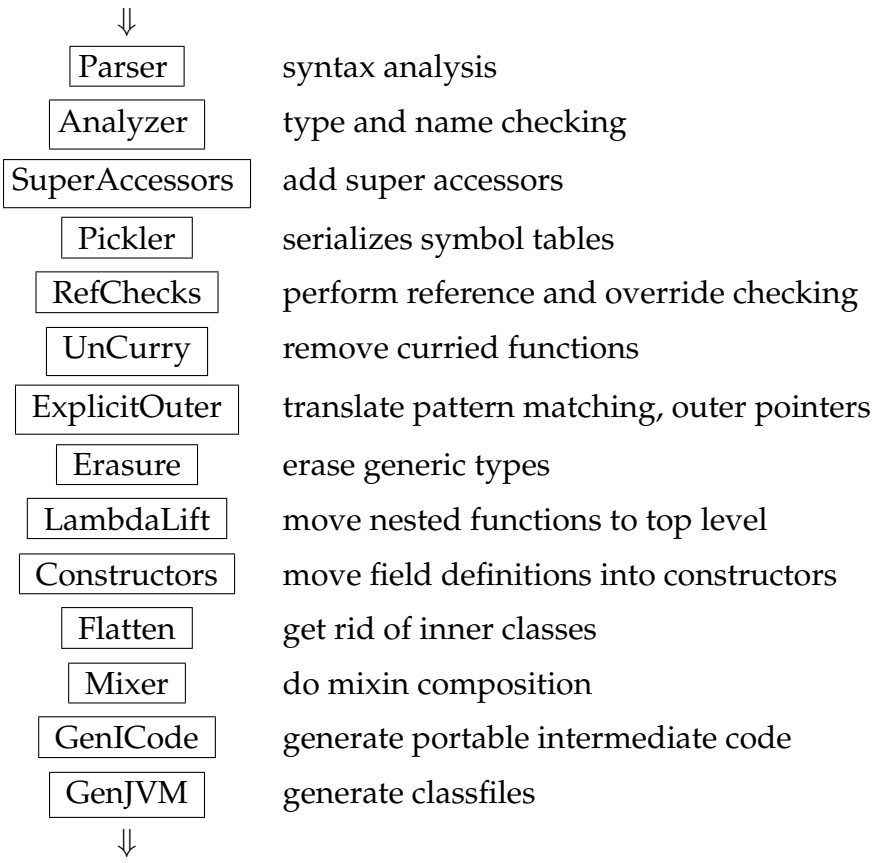


Figure 4.1: Phases of the `Scala` compiler

after following through the material contained in this thesis.

SCALA is a statically typed language that blends object-oriented programming constructs with higher-order functions, pattern matching and a component-oriented type system providing type members, nested types and multiple inheritance. The extensions are compiled down to the JVM in order to ensure compatibility with existing JAVA libraries and foster real-world adoption.

The reference SCALA compiler rewrites source programs in several phases, each time taking a higher-level construct or combination of expressions and yielding one that is closer to the JVM class-file format. Its phases (except specific ones that are experimental or deal with optimization) are depicted in Figure 4.1. The architecture of the compiler has been described by Schinz [78], Altherr [5] and Zenger and Odersky [98].

4.1.1 Applications of Pattern Matching

Pattern matching is widely used in the compiler and the standard library of the SCALA programming language. There is a variety of programming language elements that are syntactic sugar for some form of pattern matching. Among these language elements are

- value definitions through patterns, as in `val (min,max) = computeMinMax()` which involves a tuple pattern,
- for-comprehensions, as in `for(`p `<- items) {...}` which will filter those items that match pattern p, and
- partial functions, which consist of just the case clauses of a pattern matching expressions, and
- catch-blocks for exceptions.

We focus on pattern matching itself and do not dwell on these derived constructs, which are explained in detail in the SCALA language specification [69]. Odersky claims that compared to JAVA, the primitives and libraries of the SCALA language can reduce code size of equivalent programs by a factor of 2 [68]. Given the above list, it is clear that pattern matching plays an important part in this reduction.

Application of pattern matching to semistructured data like XML is not discussed in this thesis, as it would blur the main problem of combining pattern matching with object-oriented programming. The interested reader is referred to [27, 87].

4.2 Type Patterns

The formal calculus of the preceding chapter offered a minimal pattern language consisting of extractor patterns and variable patterns. Type tests could be encoded using extractor methods that have a type test expressions as their body. However, in the context of a real-world programming language, it is usually preferred not to force programmers to use encodings. Moreover, extending the kinds of pattern and adapting the translation algorithm offers more opportunities for optimization. In this section, we will elaborate on extending the pattern language with type patterns.

Type patterns such as `x: Int` or `_: String` match all values of the given type, and bind the variable name to the value. Pattern matching with type patterns immediately corresponds to the "typecase" construct discussed in Chapter 2.

In a first-order object-oriented language, the principle challenge in translating match expressions with type patterns lies in subtyping. Consider the class declarations and the two expressions in Figure 4.2: a variable `v:Any` and a class hierarchy `One <: List <: Seq`. We like to keep our strategy of successively rewriting match expression into simpler ones that are guarded by tests, so the outcome of rewriting (on the right) contains a type tests and two branches for each possible result. The outcome of a type test for T can be used to process remaining patterns "modulo" subtyping with respect to T. There are two ways in which we

```
class  Seq  extends Any
class  List extends Seq  {...}
class  One  extends List {...} //singleton lists


(v:Any) match {
  case x:List => b1
  case y:Seq  => b2
  case z:One  => b3
}
```

```
if(v.isInstanceOf[List]) {
  val tmp = v.asInstanceOf[List]
  tmp match {
    case x      => b1
    case y      => b2
    case z:One  => b3
  }
} else v match {
    case y:Seq  => b2
}
```

Figure 4.2: Example of Rewriting Type Patterns modulo Subtyping

can make use of the knowledge that an input value is of a "better" type, both of which are applied in the example. First, in the then-branch of the **if**-expressions, there is no need to test for `Seq`, since it is a more general type than `List`. The type pattern `y:Seq` is replaced with a pattern `y`. Second, in the else-branch, we can omit the case clause for `z:One`, since it is a more specific type pattern than `x:List` and we know statically that it will reject its input.

The rewrite rule that describes translation of type patterns is given in Figure 4.3. For a type pattern that tests for T, The function mod_T retains exactly those c_j which test for a more general or more precise pattern ("modulo"). More general patterns are replaced with variable patterns, and more specific patterns are kept as they are. Incompatible patterns are removed. In contrast, the function $incmp_T$ retains exactly those patterns that test for some type S that is either more general or incompatible with T. It is straightforward to generalize this rule to deal with multiple columns. Extractor patterns are always retained since we do not know whether the extractor method tests for a type.

4.2.1 Extractors and Type Patterns

Having type patterns suggests a very convenient shortcut for type tests in extractor methods, as introduced in Chapter 2: an extractor method can be given an argument type that is more precise that the expected type of a pattern. The semantics is then defined using a type test prior to the extractor call.

In the formalization, this can be described as allowing not only extractor signatures $@\mathbf{safe}(\mathsf{Obj})E$, but also $@\mathbf{safe}(C)E$ and change the semantics so that a type test for C is added before calling the extractor.

$$\texttt{z match \{ } c_\star^{1..k} \texttt{ \}} \quad \Longrightarrow \quad \begin{array}{l} \texttt{if(z.isInstanceOf[}C\texttt{]) \{} \\ \quad \texttt{val tmp = z.asInstanceOf[}C\texttt{]} \\ \quad \texttt{tmp match \{}{c'}_\star^{1..k}\texttt{\}} \\ \texttt{\} else} \\ \quad \texttt{z match \{}{c''}_\star^{1..k}\texttt{\}} \end{array}$$

where $c_1 \equiv \textbf{case}\ \mathsf{x}\colon C \Rightarrow b_1$, $c' = mod(c_\star)^{1..k}$ and $c'' = incmp(c_\star)^{2..k}$

$$mod_T(\textbf{case}\ \mathsf{y}\colon S \Rightarrow b; c_\star^{j..k}) = \begin{cases} \textbf{case}\ \mathsf{y} \Rightarrow b; mod_T(c_\star^{j..k}) & \text{if } T <: S \\ \textbf{case}\ \mathsf{y}\colon S \Rightarrow b; mod_T(c_\star^{j..k}) & \\ \qquad\qquad \text{if } S <: T, S \neq T & \\ mod_T(c_\star^{j..k}) \qquad \text{if } S \not<: T, T \not<: S & \end{cases}$$

$$incmp(\textbf{case}\ \mathsf{y}\colon S \Rightarrow b; c_\star^{j..k}) = \begin{cases} \textbf{case}\ \mathsf{y}\colon S \Rightarrow b \ ; incmp(c_\star^{j..k}) & \text{if } S \not<: T \\ \qquad\qquad incmp(c_\star^{j..k}) & \text{if } S <: C \end{cases}$$

Figure 4.3: Rewrite Rule for Type Patterns

This has an important effect on readability. Consider the following extractor methods, for some type C having an integer-valued method `foo`. These are equivalent when more precise argument types are allowed:

```
def unapply(x:Any): Option[Int] =
  if(x.isInstanceOf[C]) Some(x.asInstanceOf[C].foo) else None

def unapply(x:C): Some[Int] = Some(x.foo)
```

This innocent-looking convention for type tests is the key to reasoning about the static types of generic values. It also permits to write result types that use type arguments of the tested type.

4.2.2 Static Types and Null

If we have an expression of static type C and type test for type D, and we know that $C <: D$, then it seems that the type test is unnecessary. Unfortunately, this is not the case in SCALA, since it is still possible that the expression evaluates to the **`null`** value. By definitions, **`null`** does not count as an instance of type C, hence the type test should not succeed. Still, replacing a dynamic type test for C with a non-null check promises a small efficiency gain.

4.3 Case Classes

In this section, we discuss implementation aspects that are specific to case classes and case class patterns. We will see that, in many respects, case class patterns are merely elaborated forms of type patterns.

4.3.1 Direct Translation of Case Patterns

When the flexibility of extractor methods is not needed, the overhead paid for extractor calls becomes a source of inefficiency. In Chapter 2, we saw that case classes can simulate algebraic datatypes and at the same time be encoded using extractors. In this encoding, an extractor of a case class pattern was always a type test followed by accesses to its case fields. This way, we could ensure acceptance of all values that were constructed with a constructor D that is a subtype of C.

With the extension of type patterns introduced in the last subsection, it is important to note that almost all machinery is in place for a direct implementation of case patterns, i.e. one that avoids calling an extractor method at runtime. While this does not make a difference for correctness, it can make a difference for performance that goes further than simple inlining: In combinations with other type tests, and especially if the set of case alternatives is *closed*, additional optimizations become available. Consider the following SCALA definitions for lists (here specialized to integer lists), which use a new modifier, **sealed**.

```
sealed abstract IntList
case class  IntCons(head: Int, tail:List) extends IntList
case object IntNil extends IntList
```

Recall that the meaning of **sealed** is that no other case class definition may be added to `IntList`. This means, then, that in the following match expressions, only *one* type test is necessary to discriminate among `IntCons` and `IntNil` instances (if it is not the instance of one, it must be instance of the other).

```
list match {
  case IntCons(x,xs) => "handle non-empty list"
  case Nil           => "handle empty list"
}
```

For the purpose of proving such an optimized translation correct, a rule has to be added that deals with sealed case classes: If the input value of a column i has modifier **sealed** and contains only type tests, this rule can simultaneously inspect all rows and ensure that the one is replaced by a simple cast that is guaranteed to succeed (Scott and Ramsey apply the same idea is to algebraic data types [79]). The concrete shape of the translation is not very different from the optimization for typed patterns, so we omit it here.

```
if(x.isInstanceOf[C1]) {                    x.$tag match {
                                              case C1.$tag =>
} else if(x.isInstanceOf[C2]) {                 if(x.isInstanceOf[C1]) ...

} else ...                                    case C2.$tag =>
                                                if(x.isInstanceOf[C2]) ...
... if(x.isInstanceOf[Cn]) {
                                              ...
} else {                                      case Cn.$tag =>
                                                if(x.isInstanceOf[Cn]) ...
}                                           }
```

Figure 4.4: If vs. Switch

4.3.2 Using tags for case classes

For case classes that simulate an algebraic data type, type tests can be avoided by adding an integer tag to each case class and performing a switch. The idea is depicted in Figure 4.4, which shows that the linear search through cascading **if**-expressions can be avoided. Due to separate compilation, it is not possible to ensure uniqueness of tags, so that the succeeding integer test does not guarantee that the object is of the right type. As a consequence, an `isInstanceOf` check has to be done anyway. Only for class hierarchies that are **sealed**, this additional type test can be omitted. The evaluation of this implementation choice is deferred to the next Chapter.

4.3.3 Using Reference Equality on Singleton Objects

In the case of SCALA lists, we additionally take into consideration that `Nil` is a named constant, another important savings become possible: `Nil` can be tested using a fast check for reference equality, so it this test should be preferred over testing for type `Cons`. This obviously works for any pattern that tests for a singleton object. Such a special case in the translation is useful for performance regardless of whether the type is sealed or not. However, it complicates the formal treatment significantly, without adding new insights.

4.4 Incomplete Matches and Redundant Clauses

Incompleteness checking is a useful feature of functional pattern matching that is easy to achieve with the matrix-based translation algorithm and which could be useful also in object-oriented settings. The SCALA language offers case classes and the **sealed** keyword

to this end. We hope to formalize incompleteness checking in future work, and study how to reintroduce the optimizations possible for algebraic datatypes.

The central notion is the one of "combination". The combination of patterns in a case clause collectively tests a particular property of the input values. When reasoning about the set of accepted input values, it is helpful to think of case clauses and patterns as partially-defined functions. The *domain of a case clause* (resp. *pattern*) is then exactly the subset of input values that is accepted by the case clause or pattern. For all combinations of sealed type, there must be exactly one pattern row that matches (non-sealed are ignored). In practice, programmers tend to introduce two kinds of mistakes when writing match expressions:

1. The union of the domains of the case clauses does not cover the whole range of input values, i.e. there exists a combination of input values rejected by all case clauses. The match expression is said to be *incomplete*.

2. There exists a *redundant* case clause whose domain is entirely contained in the domain of a preceding case clause, so that the clause is never applied to input values it can accept.

In the formalization, we artificially made every match exception *complete* using a convention, namely that the last case clause contains only variable patterns. This catch-all clause at the end has a domain that obviously covers the entire range of input values, but it would be too constraining to force programmers to follow this convention. In functional programming languages, programmers have come to expect that the compiler gives appropriate warnings if a pattern matching expression is encountered that suffers from such a mistake [60]. In the following, we show how this detection is integrated into pattern matching translation.

4.4.1 Match Exceptions

In practice, a catch-all clause can easily be added automatically. The body of this automatically added case clause is a single statement that throws a match exception. This follows the philosophy that certain abnormal situations that occur at runtime are signaled by throwing an exception. Figure 4.5 shows how match exceptions and temporary variables are added (the input value can be used to display a diagnostic exception message)

4.4.2 Incompleteness Checking on Sealed Types

We argued before that extractor-based pattern matching allows us to preserve encapsulation by decoupling the type of the accepted values from the behavior of the extractor, or, put

```
e match {
  case p1 => b1
  ...
  case pn => bn
}
```

```
val tmp : T = e
tmp match {
  case p1 => b1
  ...
  case pn => bn
  case _  =>
    throw new MatchException(tmp)
}
```

Figure 4.5: Temps and Match Exceptions

another way, the structure of its domain. Also, extractors can contain arbitrary code. This flexibility makes incompleteness checking impossible without further user annotations.

On the other hand, if a domain is completely defined through fixed data definitions, so that it may be covered by a fixed set of patterns, it seems appropriate to accept the coupling between types and domains, as is done for case classes. To this end, we introduce a **sealed** modifier in order to indicate that the set of direct subclasses is known and fixed (i.e. cannot be extended - more on this below). Examples from the SCALA library include:

```
sealed abstract class List[+A]
case object Nil extends List[Nothing]
case class ::[A](hd:A,tl:List[A]) extends List[A]

sealed abstract class Option[+A]
case object None extends Option[Nothing]
case class Some[A](x:A) extends Option[A]
```

With these definitions, we can be more selective when adding default cases that throw match exceptions. Using the information that the types of some input values are sealed, we can check completeness for a case clause matrix by testing it against each tuple in the Cartesian product of sealed input types. Input values that are of non-sealed types can simply be ignored. Here is an example of a match expressions and such a Cartesian product, for `xs:List[String]` and `ys:List[Int]`

```
(xs, ys) match {
  case ( Nil,   _) => 1
  case (_::_, Nil) => 2
  case ( Nil, Nil) => 3
}
```

```
   Nil        Nil    (1)
   Nil      ::[Int]  (1)
::[String]    Nil    (2)
::[String]  ::[Int]  (?)
```

The example shows that this match is incomplete, because of the combination in the last row. Note that in SCALA, this is a match expression operation on a single input value (a pair). The deconstruction of the tuple is guaranteed to succeed, leaving us with two sub-values and a proper case clause matrix.

The incompleteness check is not necessarily carried out on the top-level. The following

match expression is similar to the one above, but instead of a tuple instance being matched against tuple patterns, it has a case class instance `Foo(xs,ys)` as input value and slightly more refined patterns (we give a simplified version of the output to help readability)

```
f match {
  case Foo(Nil, _::_) => 1
  case Foo(_::_, Nil) => 2
  case Foo(Nil,  Nil) => 3
  case z:Foo          => 4
}
```

```
if(f.isInstanceOf[Foo]) {
  (f.xs,f.ys) match {
    case ( Nil,_::_) => 1
    case (_::_, Nil) => 2
    case ( Nil, Nil) => 3
    case (   _,   _) => 4
  }
} else throw new MatchException(f)
```

In deconstructing a case class and typed patterns, we are tacitly assuming a generalization of the formal translation algorithm that deals with these patterns. Since such patterns mainly test types and the formal calculus already assumed that every class is a case class, little insights are to be gained from the detailed translation rules and adapted correctness proof, which is therefore omitted.

Rewriting the clause matrix lifts the sub-patterns to the top-level and generates the "dummy patterns" in the last row. It is when translating the match `(f.xs,f.ys) match { ... }` that incompleteness checking is activated (because it has input values of sealed type).

4.4.3 Computing Sealed Type Candidates

Above we used a working definition of candidate types for a given input value of type C that comprised the set $D_\star^{1..n}$ of direct subclasses of C. Since sealed types can be part of a hierarchy, this definition can be extended, such that direct subclasses of a sealed type D_i can be recursively taken into account. Here is the complete, recursive definition of the candidate closure:

$$
\begin{array}{lll}
cand(C) & = \emptyset & \text{if } C \text{ is not sealed} \\
cand(C) & = \bigcup_{1..n} cand(D_\star) & \text{if } C \text{ is sealed, abstract} \\
cand(C) & = C \cup \bigcup_{1..n} cand(D_\star) & \text{if } C \text{ is sealed, not abstract} \\
 & & \text{where } D_\star^{1..n} \text{ direct subclasses of } C
\end{array}
$$

It is important to note here that the presented mechanism of sealed types only limits the *direct* descendants of a type C. In practice, separate compilation makes such a requirement is hard to define and hard to check, which requires all subclasses to be defined in the same compilation unit. So far, the prime motivation for sealed types was to allow convenient features from algebraic data types. Another benefit of **`sealed`** will become clear when implementation of case class is discussed: when the set of candidate types is known to be of size n, than a complete match does not need to perform more than $n - 1$ tests.

4.4.4 Detection of Redundant Clauses

As mentioned above, a case clause is redundant if its domain is entirely contained in the domain of a preceding case class, causing the body to be unreachable.

Detecting redundancy can be considered as a variation of detecting incompleteness [60]: in this method, the Cartesian product of types that are tested by the given case clause is tested against the clause matrix of the preceding cases. If the preceding cases cover the combination, then the case is redundant. This check would have to be done each time a case clause matrix is built, in other words, for each match expression that is in the source or generated by a rewrite rule.

However, there is a simpler way: the translation algorithm, through the (Var) rule, only generates code for the body of a case clause if this case clause is actually reached. Redundant cases are simply dropped, which can be used to detect redundancy. For this technique, we merely keep a hash-table of generated case clause bodies, and at the end of the translation compare its entries with the bodies of the original match expressions. The ones that are not in the hash-table, are redundant. This redundancy detection technique is the same as in Petterson's algorithm devised for algebraic data types [73].

4.4.5 Disabling Incompleteness Checking

In practice, incompleteness checking is a very useful help for working with a sealed data type implemented by case classes. However, there arise occasional situations where the incompleteness warning is unnecessary, because the program conforms to some invariant that is not represented in the type system. In such a case, we allow the programmer to annotate the type of the input value with `@unchecked`, such that these unnecessary warnings are suppressed.

4.5 Code Generation

In this section, we summarize the low-level operations that are used in the pattern matching implementation. The essential point here is to show that high-level constructs like **`if`**-expressions and method calls are used side-by-side with low-level operations like jumps. This is of course due to balancing expressivity of pattern matching with considerations regarding performance and code size.

4.5.1 Target Operations

In order to address efficient code generation for pattern matching statements, we need to know the available target instructions are. The discussion so far assumed a built-in operation $a?\{\texttt{x}\colon C \Rightarrow b\}/\{d\}$. This operation can be easily realized with lower level operations as follows:

```
val tmp1 = a
if(tmp1.isInstanceOf[C])
  { val x = tmp1.asInstanceOf[C]; b }
 else
  { d }
```

Besides **if**-expressions, local variable declarations, type-tests and type-casts, there are two more operations that are relevant:

- **switch**-expressions can be more efficient than cascaded **if**-expressions in terms of performance, provided that multi branch case distinctions can be mapped to integer keys.
- using direct jumps opens the possibility of code-sharing, which is more efficient in terms of code size than duplicating bodies of case clauses. This is useful for alternative patterns.

This combination of high-level and low-level control flow operations suggests that pattern matching is translated in an early phase of the compiler (as seen in Figure 4.1 in the introduction of this thesis). At the same time, the intermediate representation at this phase should already allow low-level operations such as **switch** and jumps. The next sections contain a detailed discussion of how these operations are used in practice.

4.5.2 Equality Operations

By supporting literal patterns and named constants, pattern matching can be used to replace ubiquitous cascades of equality checks on literals and named constants. At the same time, three different notions of equality exists, namely comparison between primitive data (e.g. two integers), reference equality between objects and user-defined equality (calling the `equals` method on an object). For expressiveness, it is preferable to expose all three to the programmer.

With this knowledge, we can now describe more precisely some patterns from Section 2.8: Once we statically ensure that the type of the input value corresponds to the type of the pattern (either by looking at its static type or performing an `instanceOf`, a literal pattern like `1` or `'a'` is ultimately compared to the input value via primitive equality. String literals

like `"abc"` are an exception to this rule, since string are not primitive values on the JVM and string equality is accessible to the `equals` method.

We also expose reference equality via SCALA's singleton types which were introduced in Section 2.2.2. For a given input value v, the typed pattern `_: x.type` and the expression v`.isInstanceOf[x.type]` express the same condition, with pattern match translation replacing the former with the latter. At a later phase, the compiler then translates all occurrences of singleton type tests to reference equality instructions v `eq x`. Since the algorithm of the preceding chapter works with type tests, it seems preferable to do this *after* matching – this allows the translation algorithm to deal with type test (since a straightforward extension would be able to optimize multiple occurrences of typed patterns of the form `_: x.type`).

4.5.3 Switch Operations and avoiding `instanceof`

A special case that is worth taking into consideration is the comparison of an input value of type `Int` (or a convertible type) against *several* literal patterns. These can be translated to more efficient switch operations. Moreover, we have seen in Section 4.3.2 that switching on integer tags can replace cascades of `isInstanceOf` operations.

In the context of the JVM, two particular strategies for multi-way branch statements are available through the `tableswitch` and `lookupswitch` opcodes. Both are used with a list of key-target pairs and a default target, with the former using a jump-table and the latter a static search to find the target for a given key at runtime. While the appropriate switch instruction is chosen by the backend, translation of pattern matching can affect this choice through the sparseness of its key range: the more efficient `tableswitch` instruction can only be used when its key range is dense (like e.g. `213,214,215`), because otherwise, the size of the jump table would become too large.

When generating native code, platform specific features and strategies for multi-way branch compilation would also be relevant for efficient implementation. We refer the reader to Spuler [24] and Korobeynikov [53].

4.5.4 Jumps

As we have seen before, the recursive nature of the translation process leads to nested patterns bubbling up to the top-level. In this process, code duplication may take place whenever a case clause appears in both the succeeding branch and the failing branch of a test operation.

Consider the rewrite step of the example from Section 3.7.1:

$$
\begin{array}{lll}
\textbf{case } \mathsf{z.cons}(\pi_1, \pi_2) & \Rightarrow a \\
\textbf{case } \mathsf{z.nil}() & \Rightarrow b \\
\textbf{case } \mathsf{z.cons}(\pi_3, \pi_4) & \Rightarrow d \\
\textbf{case } \mathsf{y}_0 & \Rightarrow e
\end{array}
\qquad
\begin{array}{llll}
\textbf{case } \mathsf{y}_1 & , \pi_1 & , \pi_2 & \Rightarrow a \\
\textbf{case } \mathsf{z.nil}() & , \mathsf{y}_2 & , \mathsf{y}_3 & \Rightarrow b \\
\textbf{case } \mathsf{y}_4 & , \pi_3 & , \pi_4 & \Rightarrow d \\
\textbf{case } \mathsf{y}_0 & , \mathsf{y}_5 & , \mathsf{y}_6 & \Rightarrow e \\
\quad - - - & & & \\
\textbf{case } \mathsf{z.nil}() & \Rightarrow b & & \\
\textbf{case } \mathsf{y}_0 & \Rightarrow e & &
\end{array}
$$

To avoid duplication, a jump expression $\mathbf{jmp}(e_\star^{1..n})$ and a labeled block $l(\mathsf{x}_\star^{1..n}) : e$ are needed. The variables introduced in the labeling instruction are assumed mutable and used to bind the arguments which are given to the jump. This way, variable bindings obtained by evaluating a pattern in one line of the matrix can be communicated when deferring control to the appropriate case clause body. The backend can then translate this high-level jump to a normal jump followed by n assignment operations.

With the help of these instructions, the rewrite rule will yield two match expressions whose case clauses are shown below:

$$
\begin{array}{llll}
\textbf{case } \mathsf{y}_1 & , \pi_1 & , \pi_2 & \Rightarrow a \\
\textbf{case } \mathsf{z.nil}() & , \mathsf{y}_2 & , \mathsf{y}_3 & \Rightarrow \mathbf{jmp}\ l_1(); l_1() : b \\
\textbf{case } \mathsf{y}_4 & , \pi_3 & , \pi_4 & \Rightarrow d \\
\textbf{case } \mathsf{y}_0 & , \mathsf{y}_5 & , \mathsf{y}_6 & \Rightarrow \mathbf{jmp}\ l_2(\mathsf{y}_0); l_2(\mathsf{y}'_0) : e\{\mathsf{y}_0 \mapsto \mathsf{y}'_0\} \\
\quad - - - & & & \\
\textbf{case } \mathsf{z.nil}() & \Rightarrow \mathbf{jmp}\ l_1() & & \\
\textbf{case } \mathsf{y}_0 & \Rightarrow \mathbf{jmp}\ l_2(\mathsf{y}_0) & &
\end{array}
$$

In this example, only binding y_0 needs to be communicated. However, it is not wise to generate such jumps directly: it might be that the case clause matrix on the succeeding branch (the upper part) will optimize away the clause that contains the jump target, yielding an inconsistent program. We therefore delay the generation of jumps, using a hash-table of abstract syntax trees. Whenever all patterns of a clause are tested, i.e. in the implementation of rewrite rule (Var), the code of the *labeled* body is inserted if the body has not previously been generated, and a jump is inserted otherwise.

We will briefly describe another application of code sharing, namely the extension of alternative patterns $p_{ij} = \pi_1 | \ldots | \pi_n$ to discuss this point. Alternative patterns can be removed using a simple preprocessing step which is illustrated with the following transformation of the case clause matrix (letters $\mathcal{A} - \mathcal{F}$ being unchanged parts of the matrix). Note here that the leftmost column in sub-matrix $\mathcal{E}$ contains the body of the (single) case clause containing

the alternative pattern.

$$\begin{bmatrix} \mathcal{A} & \mathcal{B} & \mathcal{C} \\ \mathcal{D} & (\pi_1|\ldots|\pi_n) & \mathcal{E} \\ \mathcal{E} & \mathcal{F} & \mathcal{G} \end{bmatrix} \Longrightarrow \begin{bmatrix} \mathcal{A} & \mathcal{B} & \mathcal{C} \\ \mathcal{D} & \pi_1 & \mathcal{E} \\ \vdots & \vdots & \vdots \\ \mathcal{D} & \pi_n & \mathcal{E} \\ \mathcal{E} & \mathcal{F} & \mathcal{G} \end{bmatrix}$$

The duplication of the body increases the code size. By using jumps, the blowup in code size can be avoided.

4.6 Guards

While the above extensions aimed at increasing the expressivity of patterns, there is a point where using pattern matching is not enough to express the constraints relating to a case. For this reason, the SCALA programming language provides the possibility to annotate a case clause with a *guard*. A guard is simply a boolean expression that may make use of the variables bound in the pattern. If it evaluates to true, then the body is evaluated. If, however, the guard evaluates to false, then the case clause is ignored and the remaining cases are tested.

We illustrate the use of guards with the search tree example from Chapter 2.

```
def insert(i: Int): SearchTree = this match {
  case Leaf                    =>
    Node(item, Leaf,Leaf)
  case Node(i,l,r) if i < item =>
    Node(item, left.insert(i), right)
  case Node(i,l,r) if i > item =>
    Node(item, left, right.insert(i))
  case Node(i,l,r)             =>
    this
}
```

Guards can lead to subtle interactions, which makes it difficult to share guards the same way that we might want to share case clause bodies. Consider the following code fragment, which shows a match expression operating on integers in the presence of a guard.

```
i match {
  case 1 | 2 if cond => b1
  case 1             => b2
}
```

After preprocessing, this pattern becomes

```
i match {
  case 1 if cond => b1
  case 2 if cond => b1
  case 1         => b2
}
```

Clearly, body b_1 can be shared between the first and second case clause. However, it is also clear that the condition cannot be shared: if it is executed in the first clause and fails, evaluation needs to continue with the second clause, whereas failing evaluation of the guard in the second clause leads to evaluation of the third clause.

The use of guards also has consequences for incompleteness checking. While checking a number of patterns for completeness involves construction of Cartesian products of finite sets, the static detection of satisfiability of boolean conditions in guards is obviously undecidable. We can possibly resort to approximations and static analyzes as a remedy [25].

4.7 Summary

We discussed implementation aspects of pattern matching in the context of the SCALA compiler. While some of the changes that we added here enhanced the expressivity (such as case classes and type patterns) of the matching construct present formally, others implementation issues can be characterized by the motivation to enhance runtime performance and code size. Sometimes, there are choices to be made, and an overall assessment of pattern matching as a high-level construct that is translated to low-level instructions remains to be made. This is the subject of the next chapter.

Chapter 5

Performance Evaluation

When introducing pattern matching in Chapter 2, we compared conciseness of expressions as well as maintainability and evolution of the built-in pattern matching construct and its standard object-oriented encodings. Now we turn to the aspects of performance and scalability, aiming to answer the question of the cost of built-in pattern matching in terms of execution time and code size. We carry out several micro-benchmarks and an application benchmark. Particular attention will be paid to the implementation choice of replacing type-tests for case class tests with integer tags.

5.1 Questions regarding Performance

Pattern matching is a high-level construct for which lower-level encodings exist, and programmers have an (often correct) intuition that they have a better control over performance characteristics of their programs if they use low-level operations and optimize by hand, applying knowledge of the underlying execution environment. If a primitive pattern matching is to be a viable alternative to the common and known object-oriented encodings, the relative improvements in readability and maintainability should not carry too high a cost in performance.

In order to make precise statements about performance and scalability, we should first define what precisely we intend to measure by these terms. Based on this definition, we perform a quantitative comparison and sketch a general picture of performance characteristics. We distinguish three dimensions along which performance can be measured in order to evaluate and compare pattern matching with the other techniques.

- How much time is spent for a single match expressions whose purpose is to decompose a structured datum with a single case?

- How well does the technique scale for traversals of deep data structures, where repeated use of a match expression with multiple cases is applied many times?
- How does the technique scale when input is drawn from many alternatives?

Each of these dimensions is relevant for the overall performance of applications that make heavy use of some form of pattern matching, for instance compilers performing abstract syntax tree manipulations. For each dimension, a sample program with a variable number of constructs can be given to find a data point. We are thus only concerned with the relative differences of pattern matching and its encodings.

5.2 Method

We describe the environment and benchmarks that are used to assess the runtime performance of the code generated built-in pattern matching vs. the use of standard encodings. The implementation we study is of course the production SCALA compiler, with all optimization and extensions to the formalization that were presented in the last chapter.

5.2.1 Environment

In order to compare built-in pattern matching performance with other approaches to object-oriented pattern matching, we measure execution times for typical operations involving these approaches. These measurements come from three micro-benchmarks. We then carry out an application benchmark in order to compare extractors to case classes.

The benchmarks presented here were carried out on three configurations using the SCALA distribution v2.3.1 with local modifications to benchmark case class tags. The configurations are

- a portable computer running the Mac OS X operating system on and Intel Core Duo 2.2 Ghz and Java HotSpot(TM) Server VM (build `1.4.2-76`, mixed mode) which we will call "Mac-1.4"
- *ditto*, with the Java HotSpot(TM) Server VM (build `1.5.0_07-87`, mixed mode, sharing), herafter called "Mac-1.5"
- a Pentium 4 machine running the Ubuntu GNU/Linux operating system and Java HotSpot(TM) Server VM (build `1.5.0_07-87`, mixed mode, sharing), herafter called "Linux"

The server variant is chosen by means of the `-server` option, since we are interested in the effects of JIT compilation. We run each benchmark several times, to ensure that no extra JIT-compilation takes place after we started measuring. This choice is not the only possible one, but it allows us to mask the effect of the JIT-compilation itself (which takes place in parallel with execution).

5.2.2 Limitation of Micro-Benchmarks

Micro-benchmarks have a limitation which is that they cannot accurately reflect the impact on realistic programs by themselves. This is explained by the relative importance of the idiom that is measured is not known (e.g. a match statement is only executed very rarely), and that interactions with important optimizations like inlining and register allocation depend on the remainder of the program. Therefore, they can only serve as a general indication of the contribution to overall performance. An extensive performance analysis with the goal of maximizing overall performance would take into account the generated native code, and platform characteristics, which would however mean abandoning platform independence. We use three micro-benchmarks to measure relative performance of the approaches presented in Chapter 2. The benchmarks are called BASE, DEPTH and BREADTH, and are described along with their results in a separate section for each.

5.2.3 Application Benchmark

We complement this micro-benchmark with a measurement on a real application that makes heavy use of pattern matching: scalac, the Scala compiler. Like most compilers for high-level languages, the Scala compiler works on abstract syntax trees (ASTs). Since scalac is written in Scala, these ASTs are represented using case classes and operations are expressed using match operations on these case classes. In order to evaluate the overhead caused by extractors, we take advantage of the fact that extractors are more general than case classes. We replaced all cases classes representing ASTs with suitable classes and accompanying extractors. As we mentioned earlier, this is possible without changing the meaning of the program - in particular, all match expressions used in the source code could be left as is.

The application benchmark on the compiler is measured by performing a given task (compilation of a Scala program) 10 times and taking the average of the compilation time. We use version 2.6.0 of scalac as the reference and call the version that uses extractors as its AST representation the “extractified” version. Here, factors like just-in-time compilation are not singled out, instead the wholesale performance of the compilation task is measured.

		Linux (ms)	Mac-1.5 (ms)
oo	oo decomposition	503	725
ooabs	oo with abstract class	504	1241
vis	visitor	790	1605
cast	test-and-cast	514	707
ccls	case classes	504	710
ext	extractor	1611	2675

Table 5.1: Results of the BASE benchmark on Linux and Mac-1.5

5.3 BASE Performance

The BASE benchmark establishes how the techniques perform for a single pattern. We use the logic simplification examples that was used throughout Chapter 2. As a reminder, the pattern matching version consists of the following lines:

```
t match {
  case And(l, Lit(true)) => l
  case _ => t
}
```

This match expression (and its equivalents in the different encodings) is executed against an argument `And(Var("x"), Lit(true))`. The benchmark measures execution time of $2 * 10^7$ successful matches, in milliseconds. The simplification is not applied recursively.

The BASE benchmark led to the following results in our configurations: The raw numbers are shown in Table 5.1 The graphs for the "Linux" configuration are shown in the left half of Figure 5.3. In the graph, we use abbreviations oo for object-oriented decomposition, vis for visitor, cast for test-and-cast, ccls for case classes, ext for extractors returning tuples.

Discussion: No difference is observed between the object-oriented, test-and-cast and case class approaches. The visitor and the extractor approaches suffer from having to create new objects.

5.4 DEPTH Performance

The DEPTH benchmark shows how factoring out common cases affects performance. To this end, a more realistic simplification is done, which is depicted in its pattern matching version in Figure 5.1.

In addition to the above expression and its encodings, we measure a naive attempt to improve readability of object-oriented decompostion that discards factorization of common

		Linux (ms)	Mac-1.5 (ms)
ooNaive	naive oo	1816	1891
oo	oo decomposition	327	517
ooabs	oo with abstract class	230	362
vis	visitor	735	837
cast	test-and-cast	203	271
ccls	case classes	178	263
ext	extractor	361	698

Table 5.2: Results of the DEPTH benchmark on Linux and Mac-1.5

```
t match {
  case And(Lit(true), x)  => simplify(x)
  case And(Lit(false), x) => Lit(false)
  case And(x, Lit(true))  => simplify(x)
  case And(x, Lit(false)) => Num(0)
  case And(x, y)          => And(simplify(x),simplify(y))
  case Lit(i)             => Lit(i)
  case Var(x)             => Var(x)
}
```

Figure 5.1: Recursive Transformation (Pattern Matching Version)

tests. The code of for "naive" object-oriented decomposition is contained in Figure 5.2 and is more regular and arguably more maintainable. Moreover, we measure a variant of factored object-oriented decomposition where methods are located in an abstract class rather than a trait. It is well-known that interface methods are more costly to lookup than class methods.

When several patterns are tested side-by-side, a lot of time can be saved by factoring out common tests in nested patterns. If this is done by hand, the resulting code becomes hard to read and hard to maintain.

This benchmark measures execution time of 10^5 applications of several arithmetic simplification rules that are applied side-by-side and recursively. The numbers are given in Table 5.2, with the graphical representation contained on the right side of Figure 5.3.

Discussion: The ooNaive column shows that a readable, semantically equivalent program with redundant type tests can be 6 times slower than the hand-optimized oo version. But cast and ccls improve on both. Again, vis and ext suffer from object construction. In an experimental benchmark not shown here [28], we were able to show that avoiding object construction can make ext as fast as vis.

```
def simplify(t:Term): Term = {
  if(t.isAnd) {
    val left = t.left
    if(left.isAnd && left.value)
      return simplify(t.right)
  }
  if(t.isAnd) {
    val left = t.left
    if(left.isAnd && !left.value)
      return new Lit(false)
  }
  ...
  if(t.isVar)
    return new Var(t.name)
  throw new RuntimeException
}
```

Figure 5.2: Naive, Readable Version of Object-oriented Decomposition

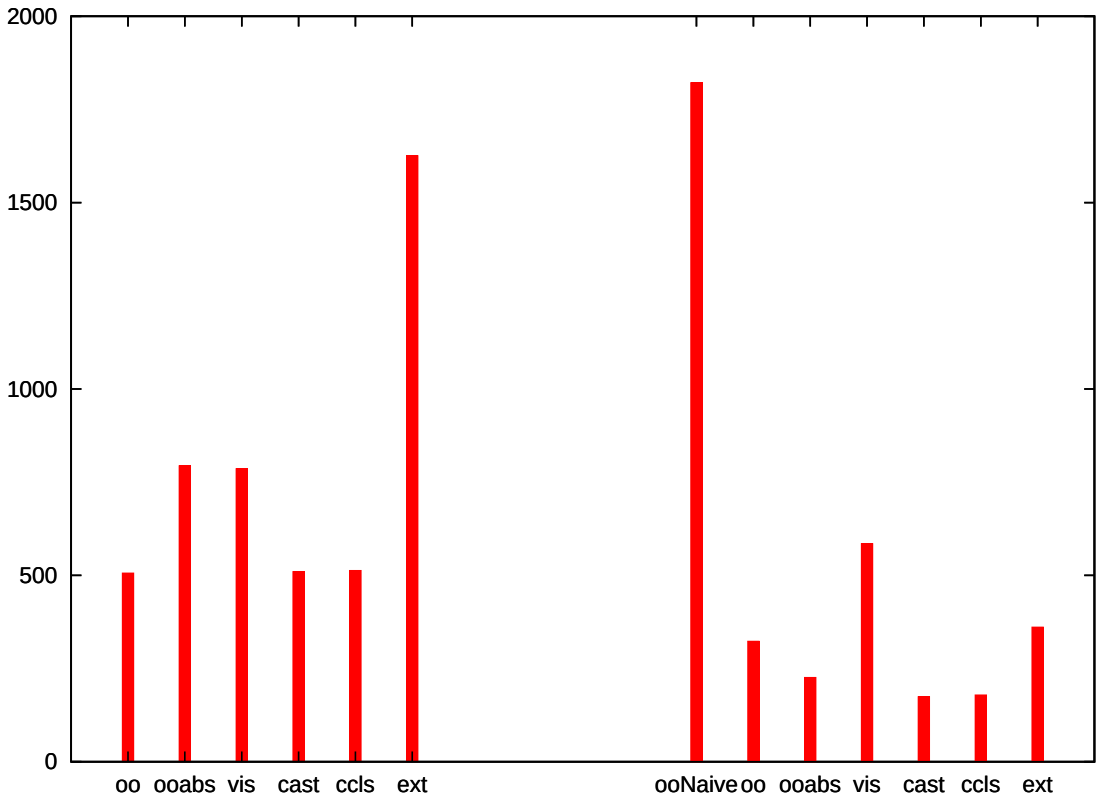
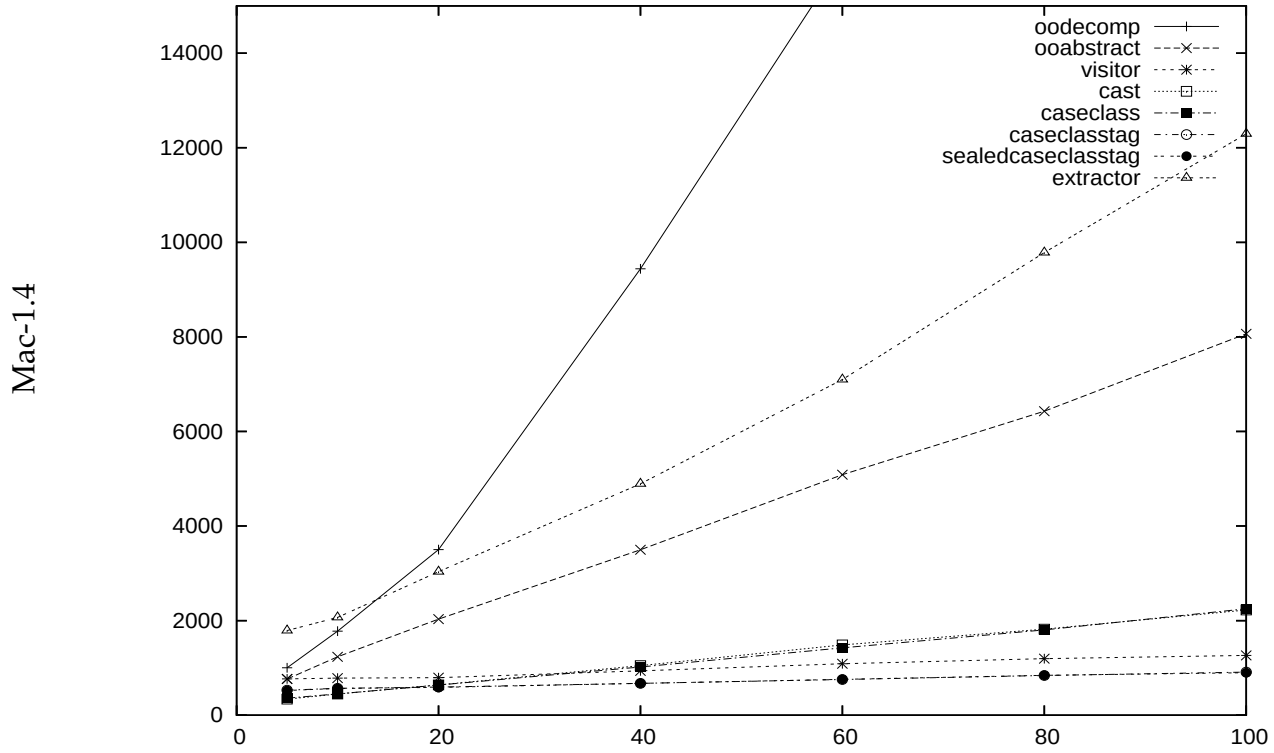


Figure 5.3: Results on BASE and DEPTH benchmarks, Linux

Figure 5.4: Diagrams for BREADTH benchmark, Mac-1.4



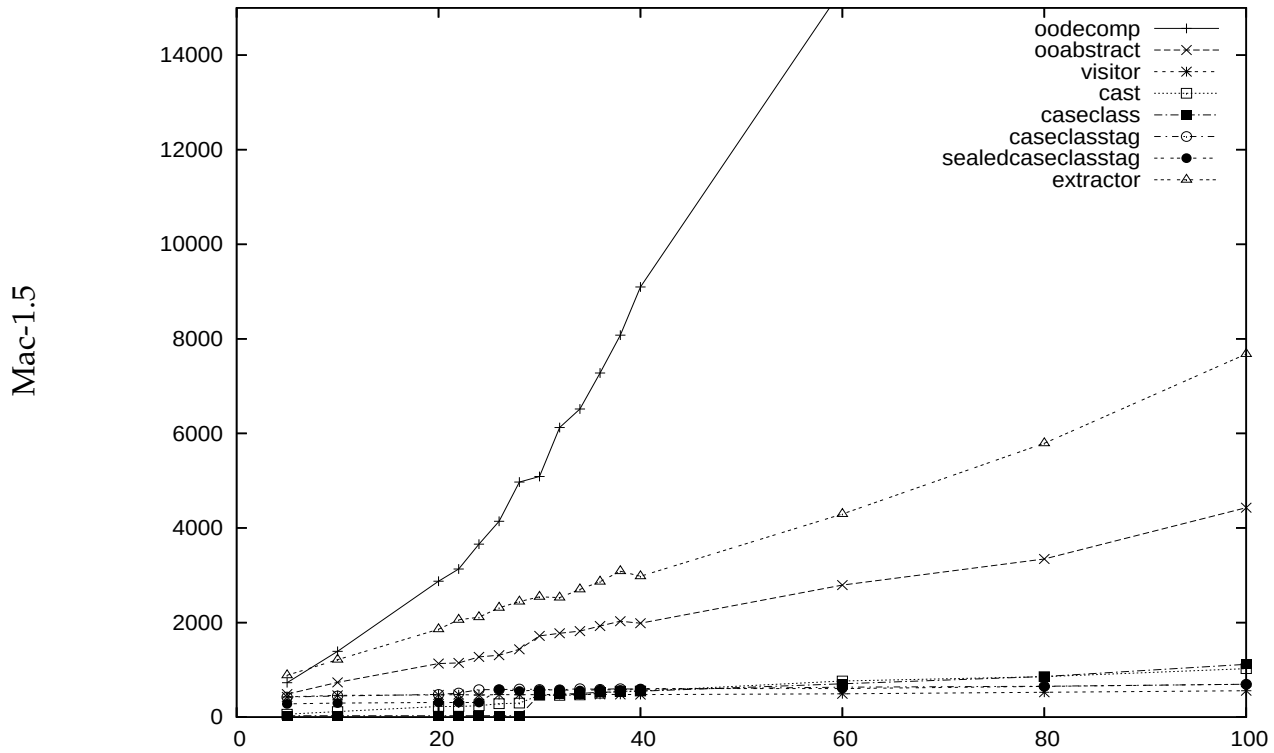
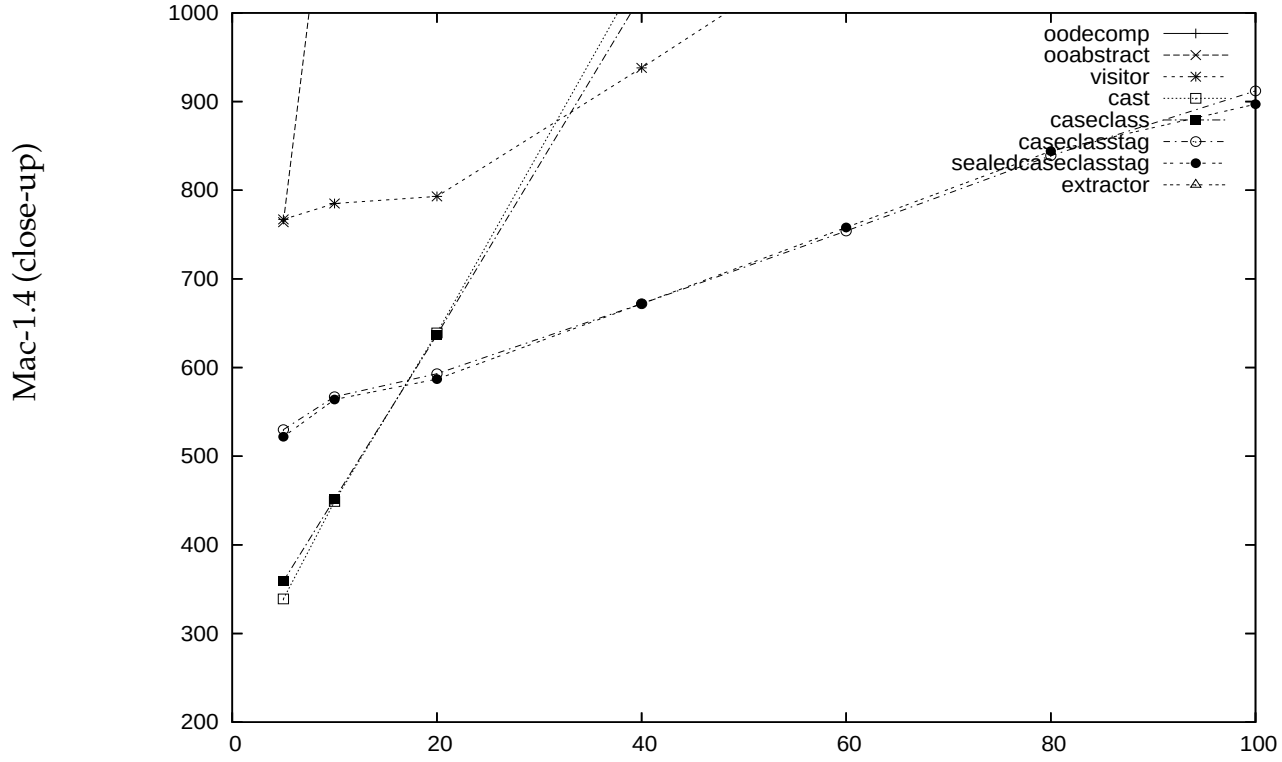


Figure 5.5: Diagrams for BREADTH benchmark, Mac-1.5

Figure 5.6: Close-up for BREADTH benchmark, Mac-1.4



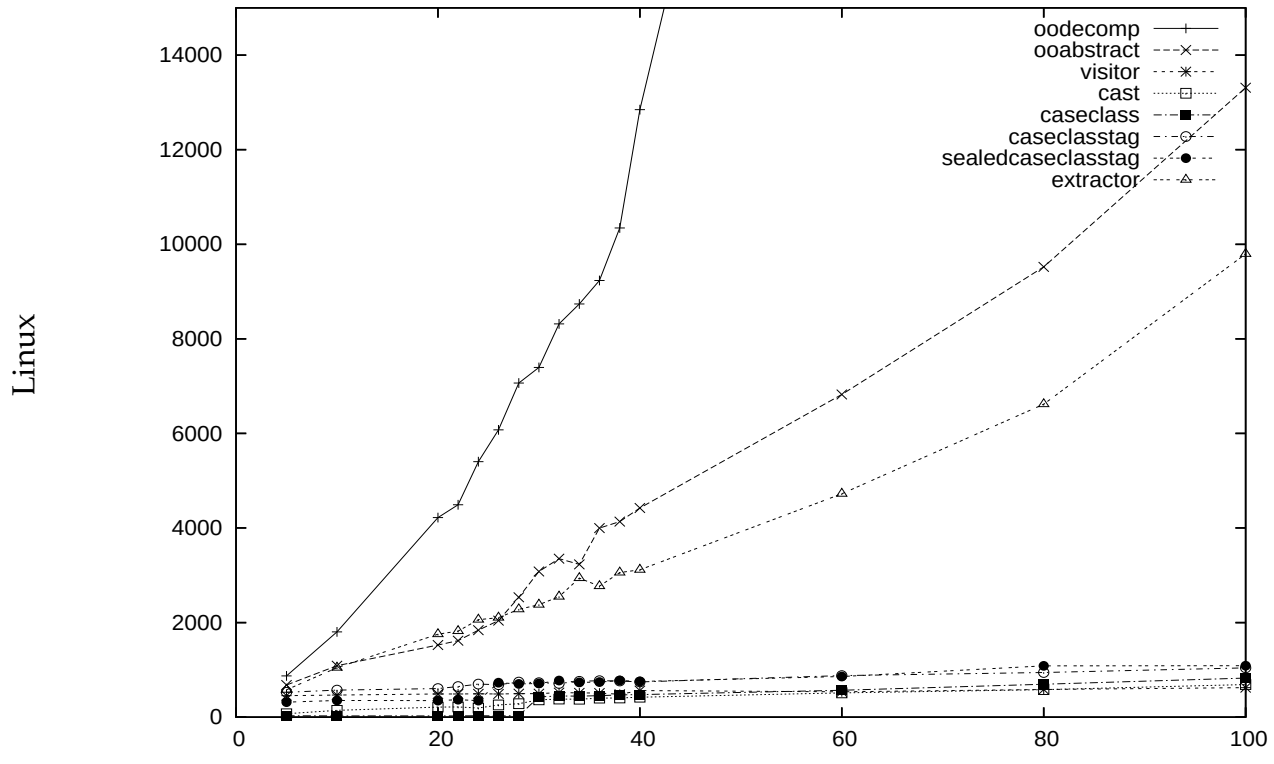


Figure 5.7: Diagrams for BREADTH benchmark, Linux

ncases	oodecomp	ooabstract	visitor	cast	caseclass	extractorTuple
5	727	488	428	62	29	886
10	1393	735	461	117	29	1215
20	2873	1135	470	225	30	1859
24	3657	1275	466	242	29	2113
28	4973	1434	473	296	29	2443
32	6126	1775	478	461	488	2528
36	7279	1926	480	506	529	2866
40	9098	1987	478	551	549	2980
60	15398	2794	492	763	704	4295
80	24098	3346	527	856	860	5791
100	35087	4430	559	1025	1116	7681

Table 5.3: Results of the BREADTH Mac-1.5

5.5 BREADTH Performance

The BREADTH benchmarks tests how the number of alternatives of a data type affect performance. To this end, for a given breadth n an algebraic signature was generated that has n constructors, whose arguments vary among the set of constructors. A set of instances is created that follows the encoding, with care taken that the set is the same for every technique.

More precisely, for a fixed number n, the BREADTH benchmark defines n generated subclass variants and a matching expression that covers all cases. Applying the match 25000 times on each term of a list of 500 randomly generated terms yields the result (the terms and the order are the same for all approaches). The raw numbers for the Mac-1.5 configuration are shown in Table 5.3. The graphs for this and the other configurations are shown in Figure 5.4, Figure 5.5 and Figure 5.7. The techniques are abbreviated oodecomp, ooabstract, visitor, cast, caseclass, extractor. Additionally, we include caseclasstag, sealedcaseclasstag which are discussed below. The range up to 1000ms shown in more detail in Figure 5.6, Figure 5.8 and Figure 5.9, since it shows some optimizations take in effect in more recent configuration.

Discussion: Chained if-statements with n conditions can be assumed to fail oodecomp in $n/2$ of the cases on average. The curve is quadratic, since the multiple interface inheritance feature of the JVM causes a search in the virtual method table during method dispatch, which itself grows larger when more alternatives are added due to the increase in `isX` methods.

In contrast, ooabstract grows linearly, because dispatching to a virtual method in an abstract class is constant. The visitor approach vis is predictably unaffected by the number of cases, because it uses double-dispatch. Surprisingly, cast and caseclass perform almost on a par with vis. Results for caseclass are close to cast, which is expected since they are only a

minor variation of type-test and type-cast. The graphs also show that in the Mac-1.4 configuration and for more than 20 branches, using integer tags to avoid the linear search pays off (although the close-up for "Mac-1.4" shows that there is still a linear search involved – which is easily identified as the switch statement). The result for extractor shows that extractors are costly, although they do better than object-oriented decomposition with interface methods. Whether extractor is better than ooabstract (Linux) or the other way round (Mac-1.5) depends on the platform.

5.5.1 Tags for Case Classes

Figure 5.8 and Figure 5.9 show a close-up of the graph for case class with and without the tag optimization. Note that the VISITOR and CAST curves are included to ease comparison with these techniques.

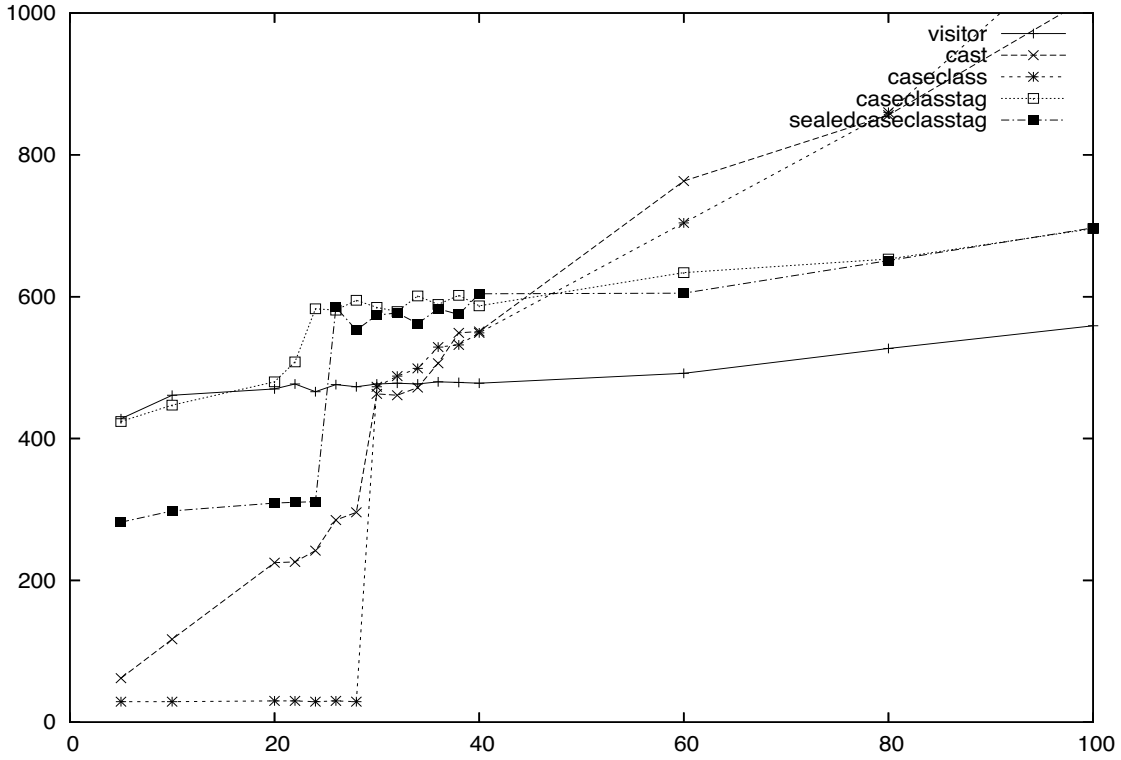
Whereas the Mac-1.4 configuration shows that using integer tags benefits scalability, the modern configurations Linux and Mac-1.5 exhibit more subtle effects, especially for less than 30 branches. This behavior, like the effect on the Mac-1.4 platform that leaves 18 classes integer tags off worse than cascading 'if's is probably due to aggressive inlining or differences in register allocation. Below a threshold of 30 branches (on the modern configurations), it seems better to generate code that the native compiler of the JVM can easily recognize to follow certain patterns, like cascading type-tests. However, one should recall that the difference is fairly small, that realistic programs may affect the inlining strategy in different ways, and that optimizing for a particular VM runs counter to the goal of platform independence.

The mixed result of these considerations is that the optimization seems to pay off in the Mac-1.4 and Mac-1.5 configurations, while due to internals of the JIT-compilation on Linux it actually introduces a slow-down. In the light of these results, it seems best to offer the optimization as a compiler option.

5.6 Application Performance

The application benchmark aims to show how a real application is affected by the overhead that extractors introduce with respect to case classes. To this end, the application at hand - the SCALA compiler - was run on 8 different sets of source files, which yielded the compilation times in Table 5.4 and Table 5.5. The graphs of the relative overhead for both platforms is found in Figure 5.10 and 5.11, with bars being normalized to 1.0 for the unmodified scalac (2.6.0) that uses case classes.

Figure 5.8: Case Class Tags in BREADTH, Mac-1.5



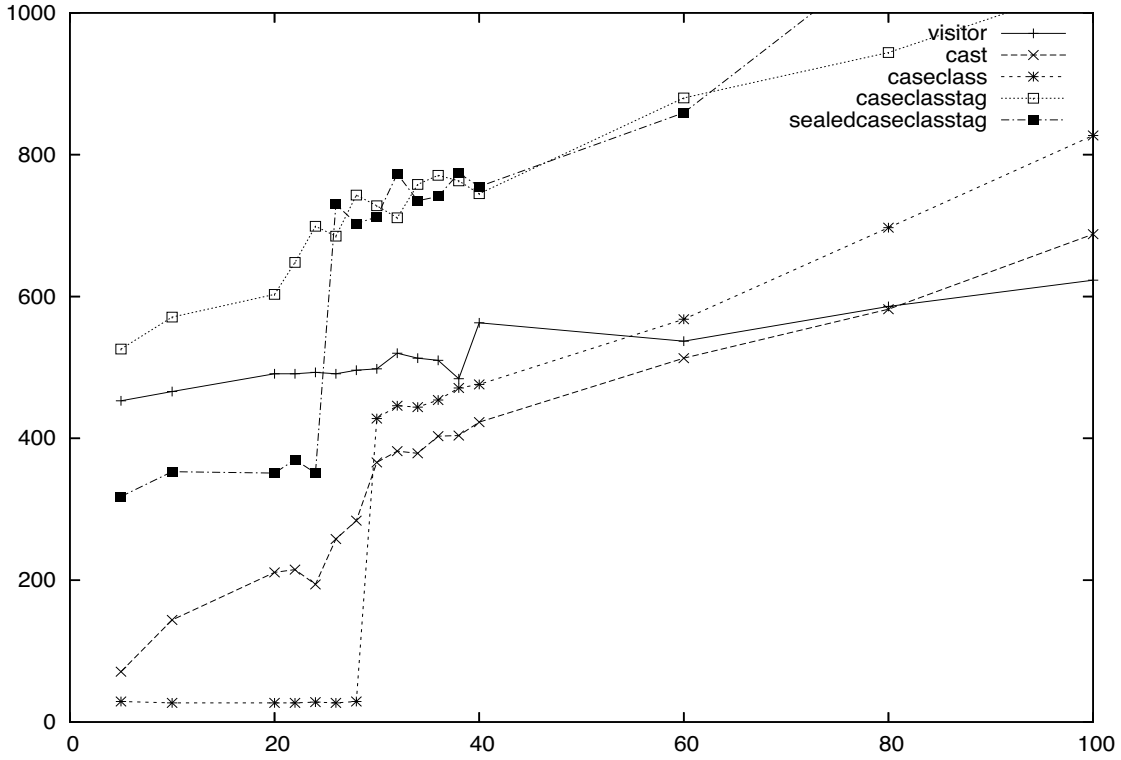


Figure 5.9: Case Class Tags in BREADTH, Linux

	extractor linux	2.6.0 linux	overhead
1. schema2src	9.9402	7.989	0.244235824
2. pex	6.3528	5.7284	0.109000768
3. xquery2src/a	22.2682	15.7758	0.411541728
4. xquery2src/b	10.8446	8.589	0.262614973
5. actors	14.0086	11.2162	0.248961324
6. dbc	14.9518	12.055	0.240298631
7. library	77.709	63.8204	0.217620071
8. compiler	193.8988	136.3185	0.422395346

Table 5.4: Measurements for Application Benchmark, Linux

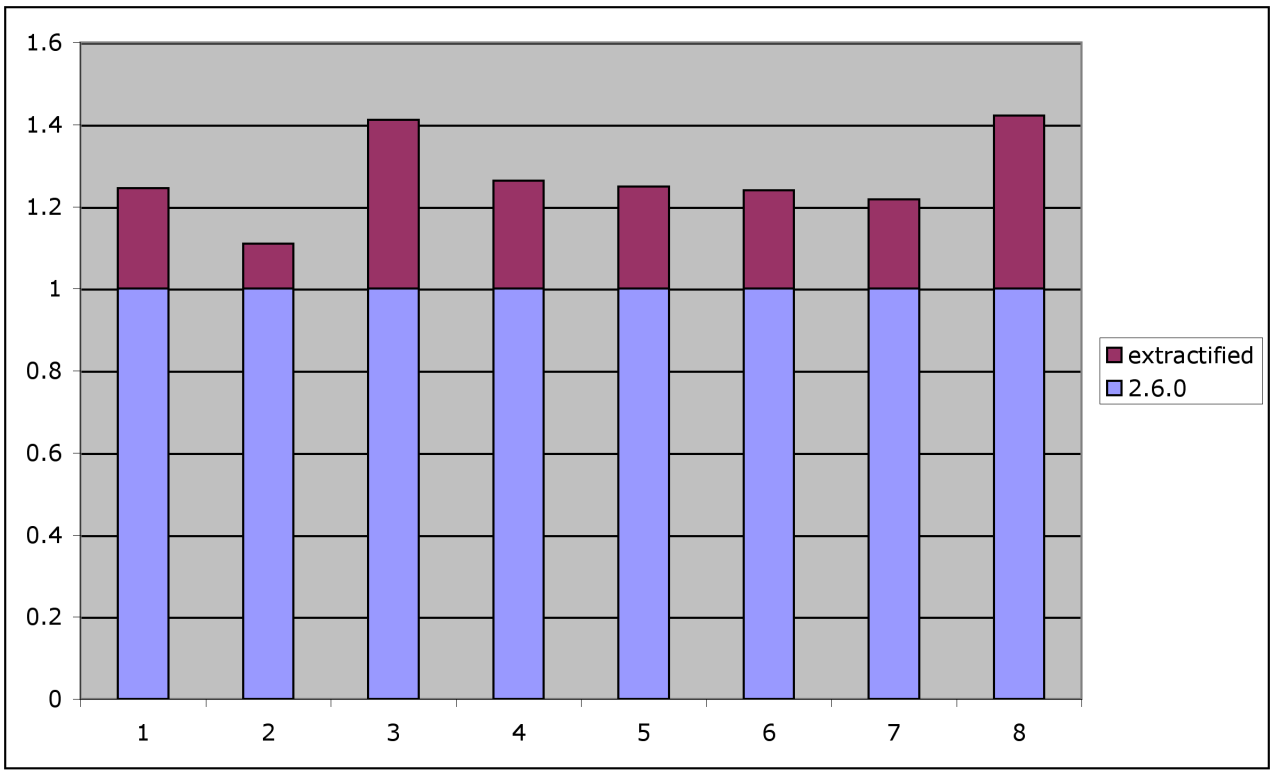
	extractor mac	2.6.0 mac	overhead
1. schema2src	8.1665	6.6625	0.225741088
2. pex	5.5374	4.9596	0.116501331
3. xquery2src/a	18.1113	12.5892	0.43863788
4. xquery2src/b	8.4615	6.8049	0.243442225
5. actors	10.5825	8.4633	0.25039878
6. dbc	11.3581	9.0104	0.26055447
7. library	57.1415	46.8958	0.218477987
8. compiler	154.877	106.6809	0.451778153

Table 5.5: Measurements for Application Benchmark, Mac

On average, the slowdown is by 26.9% on Linux and by 27.5% on the Mac platform. This result helps interpret the results of the micro-benchmark, although it is of course very specific to the application and the data.

It can be observed that the compilation times vary a lot according to the set of source files that is being compiled, which is of course due to the variation in size and complexity of the source files. More interestingly, the overhead varies as well. A possible explanation is that the JVM optimizes a sequence of type-tests for disjoint types, whereas it needs to process a sequence of conditionals calling unapply methods in a strictly sequential manner. This would mean that the ordering of case clauses together with the relative frequencies of tree nodes has a very noticeable effect on the compilation time.

Figure 5.10: Diagram for APPLICATION benchmark, Mac



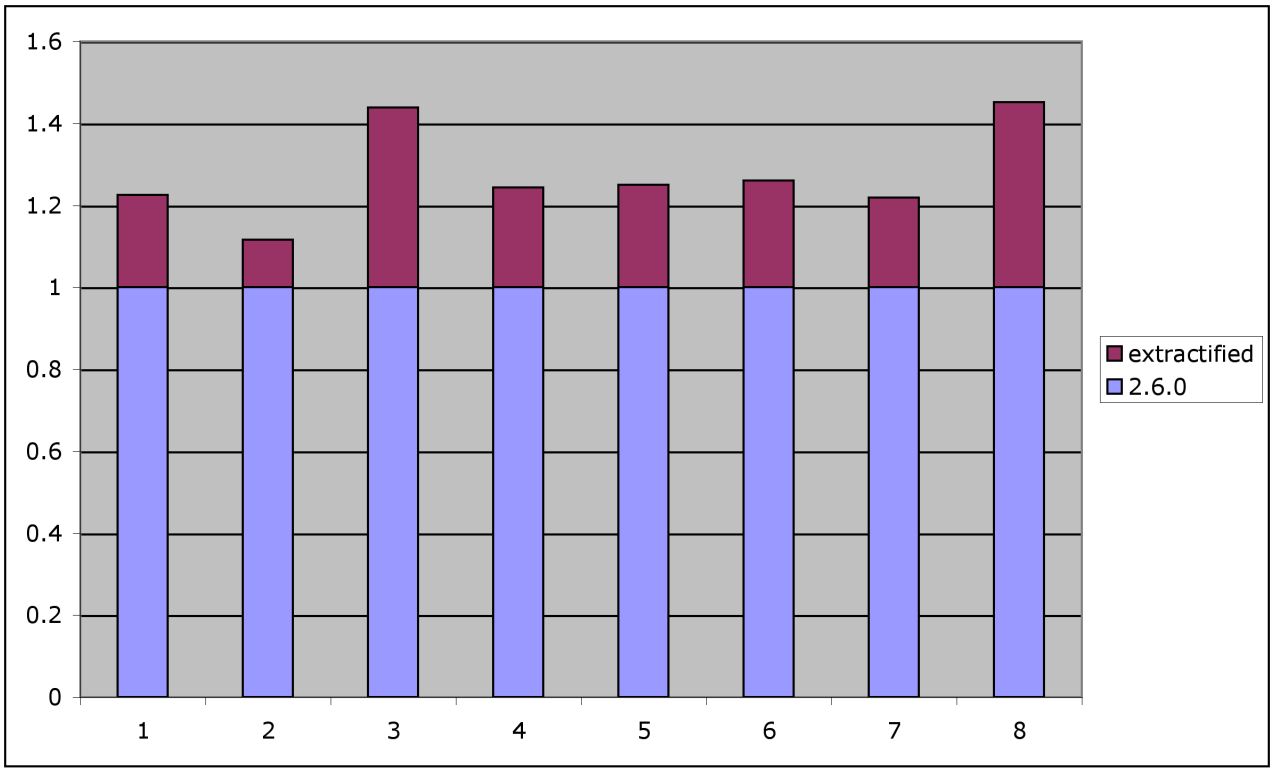


Figure 5.11: Diagram for APPLICATION benchmark, Linux

5.7 Summary

The results show that the HotSpot VM is good at optimizing the output resulting from translation of pattern matching. While providing for much more readable code, case classes and unapply methods in long-running computations have performance that is not inferior to the performance of standard techniques. Hotspot optimizes sequences of type tests, which shows that using integer tags as a replacement for cascading-ifs and type tests is not as good a strategy as could intuitively be expected.

While BASE performance of extractors is predictably worse than object-oriented decomposition, the DEPTH benchmark shows that this is amortized in deep traversal, leading to overall comparable performance. The BREADTH benchmark shows that extractors behave similarly as object-oriented decomposition if the latter is rooted in an abstract class. If methods are looked up from an interface, object-oriented decomposition is slower that extractors.

Case classes on the other hand can be used to obtain real performance gains over other techniques. It seems that modern architectures cache the result of runtime type tests and generally have more opportunites to optimize these than costly method calls used solely for classification. The overall picture seems to favor using case classes for performance, while not overly penalizing extractors for pattern matching when data abstraction is a concern.

The application benchmark shows that extractors carry a noticeable performance penalty in applications that make heavy use of pattern matching. Most of the overhead seems to be due to to object construction, with relative order of case clauses playing a role, too.

Chapter 6

Generic Pattern Matching

In this chapter, we explore interesting and useful interactions of pattern matching with parametric polymorphism. This exploration begins by reviewing second-order idioms from functional programming languages that involve pattern matching and algebraic data types. Following a standard taxonomy [50], we distinguish between parameterized algebraic data types and generalized algebraic data types. After this review, we turn to an adaptation to object-oriented pattern matching as described in the previous chapters. Given that case classes and extractor methods are based on dynamic type tests, this adaptation consists mainly of a study of the type-systematic consequences of providing *generic* type-tests. In the functional paradigm, a promising approach to model these is based on *constraints* [82]. We perform a similar treatment in the nominally typed context, with the core of our development being a theory of (nominal, object-oriented) subtyping in the presence of subtyping constraints. Unless otherwise mentioned, the theory is a simplified account of a collaboration of Emir, Kennedy, Russo and Yu [26] with the difference that class definitions discussed here are invariant, and that a language GPAT with pattern matching is introduced instead of C# minor. This calculus provides a polymorphic version of one introduced in Chapter 3.

6.1 Generic Pattern Matching, Functional Style

The polymorphic, or second-order lambda calculus discovered by Reynolds [76] and Girard [41] is the foundation of many statically typed functional programming languages. It is also the foundation of generic object-oriented programming [46, 86], albeit without the strong connection with mathematical logic. Parametric polymorphism, called "generics" by the object-oriented community, makes statically-typed programming practical since it enables generic data type definitions (like lists) and programs that can be written once but reused for a variety of types.

```
data PSrchT a = Node a (PSrchT a) (PSrchT a) | Leaf          {-Haskell-}

data Order = Less | Equal | Greater

pInsert i compare tree = case tree of
  Leaf           -> (Node i Leaf Leaf)
  (Node j le ri) -> case (compare i j) of
                      Less    -> (Node j (pInsert i le) ri)
                    | Greater -> (Node j le (pInsert i ri))
                    | Equal   -> (Node j le ri)
```

Figure 6.1: Binary Search Tree Insertion using Pattern Matching

Parametric polymorphism can be applied to algebraic data types (and hence to pattern matching) in several ways. We give examples in HASKELL following a taxonomy used by Kennedy and Russo [50] in distinguishing between *parameterized algebraic data types* and *generalized algebraic data types* (GADTs). We then expand on the significance of GADTs by showing how they allow to encode invariants of data structures within the data type. These invariants can be used to find more programmer mistakes through type-checking, without burdening the programmer with annotating his source code.

6.1.1 Parameterized Algebraic Data Types

Parametric polymorphism corresponds to *universal quantification* of a type that includes a variable, the formal type parameter. It amounts to unlimited reuse of a type definition for an unbounded number of types.

A first way of placing type parameters in algebraic data types definitions is to parameterize (only) the type itself and convene that all constructors of the type "see" the same type parameter. The type is then called a *parameterized algebraic data types* (PADTs), which yields one ADT definition per actual type parameter.

For instance, recall first-order binary search trees (Figure 2.1). Figure 6.1 defines the parameterized version, a PADT `PSrchT a` featuring a type parameter `a`. Note that we also need to introduce `Order`, since the PADT definition itself does not express that its type argument should be equipped with a comparison function. This comparison function is needed as an argument for `pInsert` function in order to compare the labels of the search tree nodes (there are ways to achieve this in HASKELL but these do not concern us here). Type inference ensures that type parameters are properly inserted and applied. Thus, after type-checking, the compiler considers each function `pInsert` and `compare` as taking an additional type parameter.

For the purpose of type-checking, the type of algebraic data type determines completely the

```
data Term :: * -> * where                                    {-Haskell-}
  Lit :: Int                          -> Term Int
  Plus:: Term Int -> Term Int         -> Term Int
  IsZ :: Term Int                     -> Term Bool
  If  :: Term Bool -> Term a -> Term a -> Term a
  Pair:: Term a -> Term b             -> Term (a,b)
  Fst :: Term (a,b)                   -> Term a
  Snd :: Term (a,b)                   -> Term b
```

Figure 6.2: Definition of a GADT for an Expression Language

types of the constructors. It is not possible to express a parametric signature that is specific to a constructor.

6.1.2 Generalized algebraic data types

Generalized algebraic data types (GADTs) go beyond quantifying the entire definition as a whole. They are obtained if one allows to abstract type information on a *per-constructor* basis. Another way to put this is to say that the constructors "do not see" the type parameters of the type they are defining. For instance, a simple expression language for a programmable calculator may define expressions as in Figure 6.2. As before, `Term` is a unary type constructor. However, its parameter is not needed in the constructor declarations, there it is specified via its *kind*. The syntax `* -> *`. indicates that `Term` is a unary type operator.

GADTs are more general in three aspects:

1. constructors do not need to return 'generic' instance of the data type. For instance, constructor `Lit` returns a `Term Int`.
2. data types can be used at different types within their own definition (e.g. `If` uses a `Term a` and a `Term Bool`).
3. a constructor may have additional type variables (e.g. `Fst` has two type parameters `a` and `b`).

The relationship between the constructors and the GADT of which they are a variant resembles the explicitly given subtyping relationship that is common in object-oriented style.

6.1.3 Existential Types

Interestingly universal quantification applied on a per-constructor basis can be considered as *existential quantification* through patterns [48]. Existential types can be used to encapsulate

```
data N = Z | S N                                        {-Haskell-}
data R
data B
data RBTree :: * -> * -> * -> * where
  Leaf ::                                          RBTree a B Z
  Red  :: a -> RBTree a B b -> RBTree a B n -> RBTree a B n
  Black:: a -> RBTree a c b -> RBTree a d n -> RBTree a B (S n)
```

Figure 6.3: Definition of a Red-Black Tree, using GADT invariants

abstract data types in functional programming [65]. Through GADTs, a limited but useful form of existential types can be simulated through pattern matching. Consider a declaration like `data T where F :: a -> T` . It can be used in expressions like `case t of F x -> e` where the exact type of x is *hidden*. In other words, it exists, but is not known. Using a trick that is loosely based on the equivalence $\big(\forall a.(Fa \rightarrow S)\big) \Leftrightarrow \big((\exists a.Fa) \rightarrow S\big)$ from second-order logic, we can type-check the body by treating the type a as a *Skolem-constant*. To this end, we introduce a fresh type variable $\&a$ whose scope spans the pattern and the body of a case. The return type S may not contain any reference to $\&a$ (alternatively, in a system with existential quantification, these have to capture-converted – we provide an example in Subsection 6.2.3).

This technique becomes even more useful when $\&a$ is known to satisfy certain *constraints*. Consider for instance that `Term Int` of the expression language in Figure 6.2 can be expressed as `Term a where [a=Int]`. Simonet and Pottier [82] give a complete formalization that shows that the full range of GADTs is covered by type equality constraints.

We shall see below that we can harness this convenient syntax for a limited form of existentials in object-oriented style.

6.1.4 An Application: Data Invariants

With GADTs, we can express data invariants (see e.g. [39]; the following example is from there). Figure 6.3 shows a GADT formulation of red-black trees. In red-black trees, approximate balance is enforced by the following conditions: every node is colored red or black, a red node may only have black subtrees, and every path from root to leaf has the same number of black nodes. We can enforce these conditions representing natural numbers and colors as types, and specifying constructors to be more picky about the arguments they accept.

Here, the type parameter `a` is used for the type of elements contained in the tree, while `c,d` are used for colors and `n` for natural numbers.

6.2 Generic Types and Subtype Constraints

6.2.1 Generic Classes and Objects in Scala

In SCALA, a class, object, trait or method declaration is made generic by following the type name with a formal type parameter section [69]. Analogously, parameterized types and methods are references with an actual type parameter section. These sections are delimited by brackets, as in `C[X,Y]`. Some examples of generic definitions are:

```
object List {     // a generic method
  def flatten[X](xs: List[List[X]]): List[X]) = { ... }
}
abstract class List[X] { // a generic class
  def :::(ys: List[X]): List[X]
}
```

Case class declarations are normal class declarations, and are made generic in the same way as classes and take part in the same subtype relationships. The subtyping relationship however is more involved than for first-order classes: there it was determined by the inheritance hierarchy, but with generic types, subtyping has to take into account type arguments. For instance, let us assume a simple generic class definition:

```
class Foo[X] extends Bar[T] {...}
```

This leads to an infinite number of subtype relationships `Foo[`R`]<:Bar[`S`]`, namely for each pair of types R, S, where S is equal to T with each occurrence of the formal type parameter X substituted with the actual type parameter R. We express this using type substitution $\{\!\{X \mapsto S\}\!\}$ and the equality $S = T\{\!\{X \mapsto R\}\!\}$.

6.2.2 From Bounds to Subtype Constraints

In practice, generic object-oriented languages also provide a means to *constrain* the type parameters. The goal here is to specify an upper bound, i.e. an interface that has to be implemented by types that are used as actual parameters. For instance in a SCALA definition **class** $C[X <: T]$. The upper bound restricts possible instantiations of X to only those types S for which $S <: T$ holds. This form of quantification is called *bounded quantification*. If the subtype relationship can mention the constrained parameter X itself, we get *F-bounded quantification* [15, 10], which is the basis for generics in GJ [12] and later SCALA, JAVA and all .NET languages supporting generics.

The typical application of F-bounded parameters is a parameter definition X `<: Comparable[`X`]`, where `Comparable[`Y`]` is a class that contains a `compareTo(that:`Y`)`

```
abstract class PSrchTree[X<:Comparable[X]] {
  def insert(i:X) = this match {
    case Leaf          => Node(i, Leaf(), Leaf())
    case Node(j,le,ri) => i.compareTo(j) match {
                case Less    => Node(j, le.insert(i), ri)
                case Greater => Node(j, le, ri.insert(i))
                case Equal   => this
  }}
}

case class Node[X<:Comparable[X]](x:X,le:Node[X],ri:Node[X])
  extends PSrchTree[X]
case class Leaf[X<:Comparable[X]]()
  extends PSrchTree[X]
```

Figure 6.4: Parameterized Search Tree in Scala

method. It means that parameter X can only be instantiated with a type S that has a `compareTo(that:`S`)` member. To put it another way, we demand that there be a total order among instances of S, which is precisely what we needed to for generic binary search trees above. SCALA additionally allows lower bounds on type parameters, which play a role when contravariant type parameters are available.

Figure 6.4 shows an object-oriented version of generic binary search tree insertion. For readability, we have used the same comparison constants as in the functional version. In contrast to the functional version, the compiler can now reject instantiations of `PSrchTree` with a type that does not support comparison (i.e. does not satisfy the constraint) by signaling a type error. Again, type inference plays a key role in keeping programs free from automatically derivable type annotations and type parameters.

6.2.3 Collecting Constraints from Pattern Structure

What happens when we match on a generic type, and we want to make use of the knowledge of the upper bounds of type parameters? We can introduce type variables in patterns, just as we use term variables. In fact, the pattern match in Figure 6.4 can be spelled out as follows.

```
def insert(i:X) = this match {
  case Leaf[&Z]          => ...
  case Node[&Y](j,le,ri) => ... // here &Y <: Comparable[&Y]
}
```

In this code, we have given names (and therefore made explicit) the type arguments of the patterns. These Skolem-constants can be inserted automatically by the type-checker during type inference. They are different from type variables introduced by method or class definitions, hence we will sometimes use the notation $\&Y$ to stress this fact (although we do

not keep this distinction in the formal treatment). The main difference is that they may not escape scope, e.g. a method with a match expression like the following would be ill-typed.

```
def nakedSkolem(x:Object): &Y = this match { // what is &Y ?
   case Node[&Y](j,le,ri) => j
 }
```

If $\&Y$ were not a Skolem-constant, but a type parameter reference, this problem could never arise. In order to solve the problem of Skolem-constants escaping their scope, one can either replace the type by an upper bound so that the return type does not mention Skolems anymore (e.g. `Object` in this case). Or, if we have existential quantification in the sub-language of type expressions, then we can capture all free Skolems and quantify them (which here would lead to $\exists Y.Y$). Given these constants, we can now elucidate the reasoning that has to be performed in the type-checking algorithm in order to type-check the generic match expressions.

In the body of the second clause of `insert`, we know that term variable `j` is of type $\&Y$. Moreover, from the definition of `Node`, we know that the actual type that will be substituted for $\&Y$ will also satisfy the subtype constraint $\&Y$`<:Comparable[`$\&Y$`]`. So we know that a `compareTo` method can be invoked on the variable `j`. This information is however not enough to type-check the comparison expression `i.compareTo(j)`, because `i` has type X and the comparison method of `i` expects an argument of type X.

What is missing is that we need to take into account the static type of the input value **`this`**. Since **`this`** is of type `PSrchTree[`X`]`, and the pattern tests for a type `Node` which extends `PSrchTree`, we can deduce that the subtype relationship `Node[`$\&Y$`] <: PSrchTree[`X`]` must hold, whatever the actual value of $\&Y$ is. A look at the inheritance hierarchy then reveals that this constraint can only be satisfied, if also the equality $\&Y = X$ holds. With this information, the method call `i.compareTo(j)` can be type-checked.

This kind of reasoning with constraints is the key that allows us to type-check the comparison operation and the more general idioms that involve GADTs. It is based on combining the evidence of an accepting pattern (more precisely, its dynamic type test) with information from the class hierarchy.

6.3 Declarative Subtyping modulo Constraints

In the following, we will introduce a meta-theory of subtyping that will be used in GPat. This second-order language resembles a SCALA fragment, however it offers a slightly different syntax where F-bounds are replaced with a set of arbitrary constraints on the type parameter. This allows us to treat the essential ingredients of generic type/pattern interactions without having to discuss the entire SCALA type system.

Declarative Subtyping $\Delta \vdash S <: T$

$$\frac{}{\Delta \vdash \mathsf{Exc} <: T}\ (\mathsf{Sthr}) \qquad \frac{}{\Delta \vdash S <: \mathsf{Obj}}\ (\mathsf{Sobj})$$

$$\frac{}{\Delta \vdash X <: X}\ (\mathsf{Svar})$$

$$\frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U}\ (\mathsf{Stran})$$

$$\frac{T <: U \in \Delta}{\Delta \vdash T <: U}\ (\mathsf{Shyp})$$

$$\frac{\Delta \vdash S_\star <: T_\star^{1..n} \quad \Delta \vdash T_\star <: S_\star^{1..n}}{\Delta \vdash C[S_\star^{1..n}] <: C[T_\star^{1..n}]}\ (\mathsf{Scon})$$

$$\frac{\Delta \vdash C[S_\star^{1..n}] <: C[T_\star^{1..n}]}{\Delta \vdash S_i <: T_i}\ (\mathsf{Sdecon+}) \qquad \frac{\Delta \vdash C[S_\star^{1..n}] <: C[T_\star^{1..n}]}{\Delta \vdash T_i <: S_i}\ (\mathsf{Sdecon-})$$

$$\frac{\mathbf{class}\ C[X_\star^{1..m}\ \Vdash \Delta](\mathsf{f}_\star : C_\star^{1..n})\ \triangleleft\ D[T_\star^{1..n}]\ \{md_\star^{1..l}\}}{\Delta \vdash C[S_\star^{1..m}] <: D[T_\star^{1..n}]\{\!\{X_\star \mapsto S_\star^{1..m}\}\!\}}\ (\mathsf{Sext})$$

$$\frac{C[T_\star^{1..k}] \triangleleft^\star D[U_\star^{1..l}] \quad \Delta \vdash C[T_\star^{1..k}] <: D[V_\star^{1..l}]}{\Delta \vdash D[U_\star^{1..l}] <: D[V_\star^{1..l}]}\ (\mathsf{Sdeext})$$

Figure 6.5: GPat Declarative Subtyping

Instead of attaching upper and lower bounds to the type parameter, we now use type parameter sections $[X_\star^{1..k}\ \Vdash\ \Delta]$ where Δ is a list of constraints $T_1 <: T_2$. We will allow the abbreviation $S = T$ for the two inequalities $S <: T$ and $T <: S$. Two important consequences of this choice are that

- subtyping relates open type expressions and depends on a typing context. In addition to type variables, the context holds all constraints that are assumed to hold, and
- constraints have to be decomposed in order to properly derive all subtyping judgments entailed by a constraint.

This leads to an expressive theory, which we shall now introduce before we turn to examples. Figure 6.5 contains declarative rules for subtyping.

The rules (Sthr), (Sobj) describe the most general and most specific type, as before. Rule (Stran) asserts that subtyping is transitive. Instantiations of the same class means that the type arguments have to be equal (inequalities holding in both directions),which is expressed by rule (Scon). To account for type variables, we add reflexivity (Svar) and hypothesis

(Shyp). These rules alone are not enough to derive all consequences of subtype constraints. For instance, assume a class declaration $\mathbf{class}\ \mathsf{C}[Z]\ \mathbf{extends}\ D[T]$. If the context contains a subtype constraint $C[X] <: C[Y]$, the we should be able to also make use of the equality $X = Y$ because the former inequality cannot hold unless also the equality holds. This is due to *invariance* of generic types in our system and justifies rules (Sdecon+) and (Sdecon-). We should also be able to use the class hierarchy to derive all constraints $C[S] <: D[T\{\!\{Z \mapsto S\}\!\}]$ that we mentioned earlier. This is done using rule (Sext).

Finally, suppose a declaration $\mathbf{class}\ \mathsf{D}[Z]\ \mathbf{extends}\ \mathsf{E}[Z]$ and the context contains $\mathsf{D}[X] <: \mathsf{E}[Y]$ and. Then, for any ground instantiations $\{\!\{X \mapsto T, Y \mapsto U\}\!\}$, we can only have $\bullet \vdash \mathsf{D}[T] <: \mathsf{E}[U]$ if also $\bullet \vdash \mathsf{E}[T] <: \mathsf{E}[U]$ and hence, $T = U$. This justifies inverting the rule (Sext) as rule (Sdeext), using the symbol $\lhd^\star$ for the transitive closure of the direct inheritance relation.

Lemma 14 (Type Substitution preserves Subtyping) *If $\Delta \vdash T <: U$, then $\Delta\Theta \vdash T\Theta <: U\Theta$ for any type substitution Θ.*

Proof By induction on the derivation. □

Lemma 15 (Weakening) *If $\Delta \vdash \Delta'$ and $\Delta' \vdash T <: U$, then $\Delta \vdash T <: U$.*

Proof By induction on the derivation. □

6.3.1 Syntax-Directed Subtyping

The declarative subtyping rules make it easy to prove properties about the system. However, cannot be used to implement an algorithm that would decide the subtyping relation, because it is always possible to introduce new subgoals using (Stran). Therefore we give different rules which are syntax-directed, but equivalent to the declarative system. Figure 6.6 has the rules, with each rule applying to exactly one form of type expression.

The judgment $\Psi \succ T <: U$ asserts that "under context Ψ we can deduce that T is a subtype of U". The transitivity rule (Stran)is eliminated and the rules (Sext), (Sup), and (Sdn)changed accordingly. An important change from the implementation perspective is that the context Ψ will no longer contain an arbitrary set of subtype assertions, the context Ψ provides upper or lower bounds for type variables. Rules (Sup) and (Sdn) make a hypothesis rule unnecessary. To show admissibility of transitivity, we need to impose some restrictions on the context Ψ. For example, in context $\Psi = \{C[X] <: Z, Z <: C[Y]\}$ we might have $\Psi \succ C[X] <: Z$ and $\Psi \succ Z <: C[Y]$, but we would not obtain $\Psi \succ C[X] <: C[Y]$. To obtain this assertion, we would need to add $X <: Y$ to Ψ. We define a notion of consistency for contexts (see Pottier [74] and Trifonov and Smith [88] for similar ideas).

Algorithmic Subtyping $\Psi \succ C <: D$

$$\frac{}{\Psi \succ \mathsf{Exc} <: T}\ (\mathsf{Sthr}) \qquad \frac{}{\Psi \succ S <: \mathsf{Obj}}\ (\mathsf{Sobj})$$

$$\frac{}{\Psi \succ X <: X}\ (\mathsf{Svar})$$

$$\frac{\Psi \succ S_\star <: T_\star^{1..n} \quad \Psi \succ T_\star <: S_\star^{1..n}}{\Psi \succ C[S_\star^{1..n}] <: C[T_\star^{1..n}]}\ (\mathsf{Scon})$$

$$\frac{S <: X \in \Psi \quad \Psi \succ R <: S}{\Psi \succ R <: X}\ (\mathsf{Sdn}) \qquad \frac{X <: S \in \Psi \quad \Psi \succ S <: T}{\Psi \succ X <: T}\ (\mathsf{Sup})$$

$$\frac{\mathbf{class}\ C[X_\star^{1..m} \Vdash \Delta](\mathsf{f}_\star : C_\star^{1..n}) \lhd D[T_\star^{1..n}]\ \{md_\star^{1..l}\}}{\Psi \succ C[S_\star^{1..m}] <: D[T_\star^{1..n}]\{\!\{X_\star \mapsto S_\star^{1..m}\}\!\}}\ (\mathsf{Sext})$$

Figure 6.6: GPat Algorithmic Subtyping

Def 7 (Consistency) *A context Ψ is consistent if for any pair of assertions $T <: X \in \Psi$ and $X <: U \in \Psi$ it is the case that $\Psi \succ T <: U$.*

With this notion, the syntax-directed and declarative rules can be related: given a consistent context Ψ that is is equivalent to a set of constraints Δ (in the sense that $\Psi \succ \Delta$ and $\Delta \vdash \Psi$), the relation $\Psi \succ \cdot <: \cdot$ coincides with $\Delta \succ \cdot <: \cdot$. We refer the reader to Emir *et al*[26] for details.

Thm 4 (Equivalence of Syntax-Directed and Declarative Rules)
Provided Ψ is consistent, $\Psi \succ \Delta$ and $\Delta \vdash \Psi$, then $\Psi \succ T <: U$ iff $\Delta \vdash T <: U$.

6.4 Subtyping Algorithm

We introduce the notion of *closure of set of types* under decomposition and inheritance, in order to clarify when our algorithm can be applied to solve the subtyping problem in presence of constraints.

Def 8 (Closure of Types) *A set of types $\mathcal{S}$ is closed if whenever $C[S_\star^{1..n}] \in \mathcal{S}$ then $S_\star^{1..n} \in \mathcal{S}$ (decomposition) and whenever $C[S_\star^{1..n}] \lhd T$ then $T \in \mathcal{S}$ (inheritance). The closure of an arbitrary set of types $\mathcal{S}$ is the least closed superset of $\mathcal{S}$.*

Our language is defined in a way that significantly restricts the class hierarchy. If these restrictions where not in place, it would be easy to arrive at an undecidable problem [49].

$$
\begin{array}{l}
\mathrm{Sub}(\Xi, \Psi, T, U) = \textbf{if } \langle T, U\rangle \in \Xi \textit{ false } \textbf{else let } \Xi' = \Xi \cup \langle T, U\rangle \textbf{ in case } T, U \textbf{ of} \\
\quad \begin{array}{ll}
X, X & \Rightarrow \textit{true} \\
T, X & \Rightarrow \bigvee_{T_\star <: X^{1..n} \in \Psi} \mathrm{Sub}(\Xi', \Psi, T, T_i) \\
X, D[S_\star^{1..n}] & \Rightarrow \bigvee_{X <: T_\star^{1..n} \in \Psi} \mathrm{Sub}(\Xi', \Psi, T_i, T) \\
C[S_\star^{1..m}], D[T_\star^{1..n}] & \Rightarrow \mathrm{Sub}(\Xi', \Psi, T\Theta, D[T_\star^{1..n}]) \text{ where} \\
 & \qquad \Theta = \{\!\{X_\star \mapsto S_\star^{1..m}\}\!\}, C \neq D, C[X_\star^{1..m}] \triangleleft T \\
C[S_\star^{1..m}], C[T_\star^{1..n}] & \Rightarrow \bigwedge^{1..n} \mathrm{Sub}(\Xi', \Psi, S_\star, T_\star) \wedge \bigwedge^{1..n} \mathrm{Sub}(\Xi', \Psi, T_\star, S_\star)
\end{array}
\end{array}
$$

Figure 6.7: Subtyping Algorithm

Def 9 (Finitary Definitions) *A set of class definitions is finitary if for any set of types $\mathcal{S}$ making use of those classes, its closure is finite.*

Class definitions in GPAT are finitary by definition: Inheritance cycles are not permitted, a parent type must have the form of a constructed type, and there is no variance.

In order to show completeness of the algorithm, we need to define small derivations, i.e. derivations where no trivial uses of (Sref) and (Stran) occur.

Def 10 (Small Derivations) *A derivation of $\Psi \succ T <: U$ is small if each proper subderivation has a conclusion other than $\Psi \succ T <: U$ and is itself small. Likewise, a derivation $\Delta \vdash T <: U$ is small if each proper subderivation has a conclusion other than $\Delta \vdash T <: U$.*

It is easy to see that an arbitrary derivation can be transformed into a small derivation. We make use of this fact in the proof of completeness. Figure 6.7 presents our subtyping algorithm. The additional parameter Ξ is a set of pairs of types representing subtype assertions already visited. The algorithm assumes that class definitions are finitary.

The following results are from [26].

Thm 5 (Soundness and Completeness of Subtyping Algorithm) $\mathrm{Sub}(\emptyset, \Psi, T, U) = \textit{true}$ *if* $\Psi \succ T <: U$.

Proof For soundness, use induction on the call tree. For completeness, Let $\mathcal{P} = \{\langle T, U\rangle \mid \Psi \succ T <: U \text{ is a subderivation of } \mathcal{D}\}$. We show by induction on $\mathcal{D}$, that if $\mathcal{D}$ is a small derivation of $\Psi \succ T <: U$ and $\Xi \cap \mathcal{P} = \emptyset$ then $\mathrm{Sub}(\Xi, \Psi, T, U) = \textit{true}$. □

Thm 6 (Termination) *For any consistent, finite Ψ and types T, U, the procedure $\mathrm{Sub}(\emptyset, \Psi, T, U)$ terminates.*

Proof Consider the set $\{T, U\} \cup \{T \mid T <: X \in \Psi\} \cup \{U \mid X <: U \in \Psi\}$. Call its closure $\mathcal{T}$, which is finite if we assume finitary class definitions. Then it is easy to see that at each recursive call to Sub, the cardinality of $(\mathcal{T} \times \mathcal{T}) \backslash \Xi$ decreases by one. Hence the algorithm terminates. □

Discussion. The algorithm is a simplified version of Emir *et al*'s subtyping algorithm [26], which deals with variant parametric types. The algorithm described there additionally deals with multiple parents. A more recent result by Kennedy and Pierce [49] assures us that since we have neither contravariance, nor infinite sets of supertypes, nor class hierarchies in which a type may have multiple supertypes with the same head constructors, our class definitions are finitary. One might be tempted to drop the list of already visited subgoals, however, these are necessary even for finitary class definitions. To see why, consider what happens for $\Psi = \{T <: X, X <: T\}$ and $\mathrm{Sub}(\emptyset, \Psi, T, U)$ for T, U not related in the class hierarchy.

6.5 Constraint Closure

The development so far assumed that we have a way of turning a set of arbitrary-shaped subtype constraints Δ into an equivalent, consistent context Ψ which only keeps type variables with their associated upper and lower bounds. It is helpful to think of such a context Ψ as a "normalized" representation of Δ: we are going to present a way to preprocess the constraint set Δ into a form that is more convenient for direct implementation. and allows for direct checking of *entailment* between constraint sets. Being able to check entailment is essential for an object-oriented language, since we will want to check that constraints introduced by a subclass entail the constraints demanded by the superclass, or that constraints available in a given typing context are sufficiently strong for a generic type instantiation to be well-formed.

Apart from enabling algorithmic checking of subtyping and entailment, another virtue of this normalization is the possibility to detect unsatisfiable constraints. Due to a programmer mistake, an *unsatisfiable* constraint set may be encountered, like set $\{C <: D\}$ where C, D are unrelated classes. Worse, such situations may arise as consequence of the subtyping rules when constraints in the source entail unsatisfiable constrains, as in $\{C <: X, X <: D\}$.

Def 11 (Constraints Closure) *A constraint set Δ is closed if it is closed under transitivity, inheritance and decomposition:*

- *If $T <: U \in \Delta$ and $U <: V \in \Delta$ then $T <: V \in \Delta$*
- *If $C[T_\star^{1..n}] <: D[U_\star^{1..m}] \in \Delta$ and $C[T_\star^{1..n}] <: D[V_\star^{1..m}] \in \Delta$ then $U_\star <: V_\star \in \Delta^{1..m}$ and $V_\star <: U_\star \in \Delta^{1..m}$*

The closure of Δ, written $\mathrm{Cl}(\Delta)$, is the least closed superset of Δ.

We can now make precise what we mean by consistent constraint sets: they must only contain constraints which express a subtype relationship that is supported by the class hierarchy.

$$
\begin{array}{rcl}
\mathrm{Dec}(X <: T) & = & X <: T \\
\mathrm{Dec}(T <: X) & = & T <: X \\
\mathrm{Dec}(C[V_\star^{1..n}] <: D[U_\star^{1..m}]) & = & \begin{cases} \bigcup^{1..n} \mathrm{Dec}(T_\star <: V_\star) \cup \bigcup^{1..n} \mathrm{Dec}(V_\star <: T_\star) \\ \quad \text{if } C[V_\star^{1..n}] \lhd^\star D[T_\star^{1..m}] \text{ for some } T_\star^{1..m} \\ \text{undefined otherwise} \end{cases}
\end{array}
$$

Figure 6.8: Constraint Decomposition

Def 12 (Consistency of constraint sets) *A constraint set Δ is consistent if for any constraint $C[T_\star^{1..n}] <: D[U_\star^{1..m}] \in \mathrm{Cl}(\Delta)$ there exists some $V_\star^{1..m}$ such that $C[T_\star^{1..n}] \lhd^\star D[V_\star^{1..m}]$.*

To construct a context Ψ from a constraint set Δ we make use of a partial function Dec defined in Figure 6.8 which takes an arbitrary constraint $T <: U$ and produces a set of constraints on type variables through a combination of inheritance and decomposition (Pottier [74] defines a similar notion).

Lemma 16 (Context Construction) *Let Δ be a set of constraints. Define*

$$
\begin{array}{rcl}
\Psi_0 & = & \bigcup_{T <: U \in \Delta} \mathrm{Dec}(T <: U) \\
\Psi_{n+1} & = & \Psi_n \cup \Sigma_{T <: X <: U \in \Psi_n} \mathrm{Dec}(T <: U)
\end{array}
$$

If the class definitions are finitary, and Δ is consistent, then Ψ_n is defined for each n and has a fix-point $\Psi = \Psi_\infty$ which is consistent and satisfies $\Delta \vdash \Psi$ and $\Psi \succ \Delta$.

This provides a means of computing a consistent Ψ that models a set of constraints Δ, or rejecting the constraints as unsatisfiable if they are found to be inconsistent. In practice, one might want to simplify constraints further, using tree automata techniques such as those described by Pottier [74], though constraint sets are unlikely to be large (generic definitions tend to have few parameters and inheritance graphs typically are not very deep).

6.6 Formal Definition of GPat

We can now generalize the object-oriented calculus with pattern matching to include generic definitions and type constraints. To this end, we give generalized definitions for a second-order language with pattern matching. We thereby establish a declarative system that permits use of GADTs using pattern matching in an object-oriented language.

S, T, U	::=	$C[T_\star^{1..k}] \mid X$
Δ	::=	$\bullet \mid T <: T, \Delta$
cd	::=	$\textbf{class}\ C[X_\star^{1..k} \Vdash \Delta](\mathsf{f}_\star : T_\star^{1..n}) \lhd D[S_\star^{1..j}]\ \{md_\star^{1..k}\}$
md	::=	$an\ \textbf{def}\ \mathsf{m}[X_\star^{1..k} \Vdash \Delta](\mathsf{x}_\star : T_\star^{1..n}) : C = \{e\}$
an	::=	$@\textbf{safe} \quad \mid \quad (\mathsf{empty})$
a, b, d, e	::=	$\textbf{null}$
	$\mid$	x
	$\mid$	$e.\mathsf{f}$
	$\mid$	$e.\mathsf{m}[T_\star^{1..k}](e_\star^{1..n})$
	$\mid$	$C[T_\star^{1..k}](e_\star^{1..n})$
	$\mid$	$\textbf{throw}$
	$\mid$	$e?\{\mathsf{x}\colon C[Y_\star^{1..k}] \Rightarrow e\}/\{e\}$
	$\mid$	$e_\star^{1..n}\ \textbf{match}\ \{c_\star^{1..m}\}$
		(convention: $c_m \equiv \textbf{case}\ \mathsf{x}_\star^{1..n} \Rightarrow e$)
c	::=	$\textbf{case}\ p_\star^{1..n} \Rightarrow e$
p, π	::=	$\mathsf{x} \quad \mid \quad \hat{v}.\mathsf{m}[X_\star^{1..k}](p_\star^{1..n})$
q	::=	$\dot{v} \quad \mid \quad \textbf{throw}$
$\dot{v}$	::=	$v \quad \mid \quad \textbf{null}$
u, v, w	::=	$C[T_\star^{1..k}](v_\star^{1..k})$
$\hat{v}$	::=	$\mathsf{x} \quad \mid \quad C[T_\star^{1..n}](\hat{v}_\star^{1..k})$

Figure 6.9: Grammar for the Generic Language GPat

6.6.1 Syntax

The grammar of the second-order calculus GPAT is depicted in Figure 6.9. We will only highlight the differences to the first-order version.

The most obvious difference is that all class and method definitions now have a type parameter section. This section has the form $[X_\star^{1..k} \Vdash \Delta]$, with the intended meaning that type parameters $X_\star^{1..k}$ satisfy the constraint set Δ. Both the number of type parameters and the constraint set may be empty, and a non-empty constraint set contains only subtype constraints of the general form $S <: T$ (we still use an equational constraint $S = T$ for the pair of inequalities $S <: T$, $T <: S$).

Type references are now either a reference to a generic class $C[T_\star^{1..k}]$ or to a type parameter X, and method invocations now have a type argument section. Type references are used in object construction. If a class C or method m does not have any type parameters, we will omit the empty type parameter section. Likewise, if there are no constraints, we omit the constraint part in the type parameter section.

Type test expressions $e?\{\mathsf{x}\colon C[Y_\star^{1..k}] \Rightarrow e\}/\{e\}$ now test a value against a type reference that has new type variables as parameters. This means that it is not possible to test against full types (e.g. List[String]), but only against the class – just as in the first-order calculus. This restriction allows us to easily implement type tests on platforms like the JVM, which we will describe below when we discuss the semantics. Unlike the first-order calculus, the type variables $Y_\star^{1..k}$ transmit some information from the definition of C, since every instance of C is known to satisfy the constraints that are part of C's class declaration. This type information is recovered and transmitted via the type variables Y which are really renamings of of type parameters of C.

Since extractor patterns $\hat{v}.\mathsf{m}[Y_\star^{1..k}](p_\star^{1..n})$ are method calls, they also offer a type argument section, however just like for test expressions, the type arguments all have to be variables. Similar to test expressions, these will transmit information from the definition of the extractor method $\hat{v}.\mathsf{m}$.

6.6.2 Semantics

The computation rules for GPat are contained in Figure 6.10 and Figure 6.11. Recall that type substitution is written as $\{\!\{X_\star \mapsto S_\star^{1..l}\}\!\}$.

The semantics is a straightforward generalization of the first-order calculus. The type arguments of methods are substituted in the method body, so the mbody and mtype judgments are modified to deal with formal type parameters. In type tests, type arguments of the tested

Computation $e \Downarrow q$

$$\frac{e \Downarrow C[T_\star^{1..m}](\dot{v}_\star^{1..n}) \quad \mathsf{fields}(C[T_\star^{1..m}]) = \mathsf{f}_\star\colon S_\star^{1..n}}{e.\mathsf{f}_i \Downarrow \dot{v}_i}\ (\mathsf{Rfld})$$

$$\frac{\begin{array}{c} e \Downarrow v \quad v \equiv C[T_\star^{1..m}](\dot{v}_\star^{1..k}) \quad e_\star \Downarrow \dot{w}_\star^{1..n} \\ \mathsf{mbody}(\mathsf{m}, C[T_\star^{1..m}]) = [X_\star^{1..l}](\mathsf{x}_\star^{1..n})d \\ \theta = \{\!\{X_\star \mapsto S_\star^{1..l}\}\!\} \quad \sigma = \{\mathsf{this} \mapsto v, \mathsf{x}_\star \mapsto \dot{w}_\star^{1..n}\} \\ d\,\theta\,\sigma \Downarrow q \end{array}}{e.\mathsf{m}[S_\star^{1..l}](e_\star^{1..n}) \Downarrow q}\ (\mathsf{Rinvk})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..n}}{C[T_\star^{1..m}](e_\star^{1..n}) \Downarrow C[T_\star^{1..m}](v_\star^{1..n})}\ (\mathsf{Rnew})$$

$$\frac{\begin{array}{c} e \Downarrow v \quad v = C[T_\star^{1..n}](v_\star^{1..n}) \quad C[T_\star^{1..n}] \lhd^\star D[S_\star^{1..m}] \\ \theta = \{\!\{X_\star \mapsto S_\star^{1..l}\}\!\} \quad \sigma = \{\mathsf{x} \mapsto v\} \quad e_1\,\theta\,\sigma \Downarrow q \end{array}}{e?\{\mathsf{x}\colon D[X_\star^{1..m}] \Rightarrow e_1\}/\{e_2\} \Downarrow q}\ (\mathsf{Rcst})$$

$$\frac{e \Downarrow \mathbf{null}\ or\ \left(e \Downarrow C[T_\star^{1..n}](\dot{v}_\star^{1..n}) \quad C[T_\star^{1..n}] \not\lhd^\star D[S_\star^{1..m}]\right) \quad e_2 \Downarrow q}{e?\{\mathsf{x}\colon D[X_\star^{1..m}] \Rightarrow e_1\}/\{e_2\} \Downarrow q}\ (\mathsf{Rskp})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..n} \quad \forall j < i\,.\,\dot{v}_\star^{1..n}; c_j \Downarrow \mathbf{reject} \quad \dot{v}_\star^{1..n}; c_i \Downarrow q}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star\} \Downarrow q}\ (\mathsf{Rmch})$$

Figure 6.10: GPat Computation Rules

$$\frac{}{\mathbf{throw} \Downarrow \mathbf{throw}}\ (\mathsf{Cthr})$$

$$\frac{e \Downarrow \mathbf{throw}}{e.\mathsf{f} \Downarrow \mathbf{throw}}\ (\mathsf{Cfld}) \qquad \frac{e \Downarrow \mathbf{throw}}{e.\mathsf{m}[T_\star^{1..m}](e_\star^{1..n}) \Downarrow \mathbf{throw}}\ (\mathsf{Crcv})$$

$$\frac{e_\star \Downarrow v_\star^{1..i-1} \quad e_i \Downarrow \mathbf{throw}}{e.\mathsf{m}[T_\star^{1..m}](e_\star^{1..n}) \Downarrow \mathbf{throw}}\ (\mathsf{Carg})$$

$$\frac{e_\star \Downarrow v_\star^{1..i-1} \quad e_i \Downarrow \mathbf{throw}}{C[T_\star^{1..m}](e_\star^{1..n}) \Downarrow \mathbf{throw}}\ (\mathsf{Cnew})$$

$$\frac{e_\star \Downarrow v_\star^{1..i-1} \quad e_i \Downarrow \mathbf{throw}}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \Downarrow \mathbf{throw}}\ (\mathsf{Cmch})$$

$$\frac{e \Downarrow \mathbf{throw}}{e?\{\mathsf{x}\colon C[X_\star^{1..n}] \Rightarrow e_1\}/\{e_2\} \Downarrow \mathbf{throw}}\ (\mathsf{Ctst})$$

Figure 6.11: GPat Computation Rules (ctd.)

value are bound to the type parameters of the type reference which we test against. Since values contain type references, the concrete type arguments are available for every instance.

We will assume that generic tuple types $\mathsf{Tuple}_n[X_\star^{1..n}]$ are available in a library. This way, we do not need to treat classes as case classes and specify a casefld judgment in order to extract their components. This choice brings the formalization closer to the SCALA implementation, which also uses tuple types to group the results returned by an extractor method.

The algorithmic subtyping relation which is needed to define test expression and matching semantics is described in Figure 6.6. The context Δ denotes a set of constraints, without any processing or simplification – the relation between sets of subtype constraints Δ and simplified contexts Ψ has been elaborated above.

Note that in rules (Rcst) and (Rskp) we are not using the subtype relation, but the reflexive and transitive closure of the inheritance relation $\lhd^\star$. This is because subtyping can only happen with respect to some context, but the dynamic type tests do not need the machinery of subtype checking with respect to constraints. The type tests are limited by their syntax to only test head types (like class C of a type $C[T_\star^{1..k}]$), which means that the calculus can be implemented in execution environments like the JVM which employ *erasure* to compile generic classes to first-order classes.

Acceptance $\dot{v}_\star^{1..n}; c \Downarrow q \quad \dot{v} \curvearrowright p \dashv \Theta; \sigma$

$$\frac{c \equiv \mathbf{case}\ p_\star^{1..n} \Rightarrow b \qquad \dot{v}_\star \curvearrowright p_\star \dashv \Theta_\star; {\sigma_\star}^{1..n} \quad b\,{\Theta_\star}^{1..n}\,{\sigma_\star}^{1..n} \Downarrow q}{\dot{v}_\star^{1..n}; c \Downarrow q}\ (\text{mcase})$$

$$\frac{}{\dot{v} \curvearrowright \mathsf{x} \dashv \bullet; \{\mathsf{x} \mapsto \dot{v}\}}\ (\text{mvar})$$

$$\frac{\begin{array}{c} u \equiv C[S_\star^{1..k}](\dot{u}_\star^{1..j}) \quad C[S_\star^{1..k}] \lhd^\star D[T_\star^{1..l}] \\ \mathsf{xtype}(\bullet, v, \mathsf{m}[Y_\star^{1..l}]) = \Delta(D[Y_\star^{1..l}])S \\ v.\mathsf{m}[T_\star^{1..l}](u) \Downarrow \mathsf{Tuple}_n(\dot{w}_\star^{1..n}) \quad \dot{w}_\star \curvearrowright p_\star \dashv \Theta_\star; {\sigma_\star}^{1..k} \end{array}}{u \curvearrowright v.\mathsf{m}[Y_\star^{1..l}](p_\star^{1..k}) \dashv \{\!\{Y_\star^{1..l} \mapsto T_\star^{1..l}\}\!\}\Theta_\star^{1..k}; {\sigma_\star}^{1..k}}\ (\text{mextr})$$

Rejection $\dot{v}_\star^{1..n}; c \Downarrow \mathbf{reject} \quad \dot{v} \curvearrowright p \dashv \mathbf{reject}$

$$\frac{v.\mathsf{m}(u) \Downarrow \mathbf{null}}{u \curvearrowright v.\mathsf{m}(p_\star^{1..k}) \dashv \mathbf{reject}}\ (\text{rnull})$$

$$\frac{\begin{array}{c} \dot{u} \equiv \mathbf{null}\ or\ \left(\dot{u} \equiv C[S_\star^{1..k}](\dot{u}_\star^{1..j}) \quad C[S_\star^{1..k}] \not\lhd^\star D[T_\star^{1..l}]\right) \\ \mathsf{xtype}(\bullet, v, \mathsf{m}[Y_\star^{1..l}]) = \Delta(D[Y_\star^{1..l}])S \end{array}}{\dot{u} \curvearrowright v.\mathsf{m}[Y_\star^{1..l}](p_\star^{1..k}) \dashv \mathbf{reject}}\ (\text{rtype})$$

$$\frac{\begin{array}{c} u \equiv C[S_\star^{1..k}](\dot{u}_\star^{1..j}) \quad C[S_\star^{1..k}] \lhd^\star D[T_\star^{1..l}] \\ \mathsf{xtype}(\bullet, v, \mathsf{m}[Y_\star^{1..l}]) = \Delta(D[Y_\star^{1..l}])S \\ v.\mathsf{m}[T_\star^{1..l}](u) \Downarrow \mathsf{Tuple}_n(\dot{w}_\star^{1..n}) \\ \dot{w}_\star \curvearrowright p_\star \dashv \Theta_\star; {\sigma_\star}^{1..i-1} \quad \dot{w}_i \curvearrowright p_i \dashv \mathbf{reject} \end{array}}{u \curvearrowright v.\mathsf{m}[Y_\star^{1..l}](p_\star^{1..k}) \dashv \mathbf{reject}}\ (\text{rchild})$$

$$\frac{c \equiv \mathbf{case}\ p_\star^{1..n} \Rightarrow b \qquad \dot{v}_\star \curvearrowright p_\star \dashv \Theta_\star {\sigma_\star}^{1..i-1} \quad \dot{v}_i \curvearrowright p_i \dashv \mathbf{reject}}{\dot{v}_\star^{1..n}; c \Downarrow \mathbf{reject}}\ (\mathsf{rcase})$$

Figure 6.12: GPat Acceptance and Rejection

6.6.3 Matching

The acceptance and rejection relationships of values, patterns and substitutions are described in Figure 6.12. Acceptance judgments produce not only term substitutions, but type substitutions as well. We discuss the rules one by one.

The (mcase) rule specifies that a case clause accepts if all its pattern accept. The resulting type and term substitutions are applied to the body, which is afterwards evaluated to a result. The (mvar) rule variables match values and **null** results, and do not produce a type substitution.

The (mextr) rule does more than in the first-order calculus. To be accepted by an extractor pattern, a value first has to conform to the argument type of the extractor (there exists some type substitution such that the formal argument type is a direct or indirect superclass instance of the runtime type). In the first-order case, this was trivial since the argument type was Obj – in the generic case however, this additional type test allows to communicate type constraints between extractors and class definitions. If the input value has a conforming type, then type parameters $Y_\star^{1..n}$ can be replaced with actual types $T_\star^{1..n}$. Using these type parameters, evaluating the extractor call then has to yield a tuple $\mathsf{Tuple}_n(w_\star^{1..n})$. Finally, all sub-patterns of the extractor pattern have to match. The resulting type and term substitutions are combined and returned.

Rejection. As before, an extractor pattern rejects its input if the extractor method returns **null** (rnull). Additionally, rejection is extended to deal with failing argument type conformance. Thus an extractor patterns rejects also if its argument is **null** of if it does not conform to its argument type (rtype). Finally, it can also reject if one of the sub-patterns rejects (rchild).

6.6.4 Typing

The typing rules for GPAT are given in Figure 6.13, Figure 6.14, Figure 6.15 and Figure 6.16. Auxiliary judgments are given in Figure 6.17. The typing rules $\Gamma \vdash e \in T$ are formulated according to a convention that Γ contains not only variable-type bindings $\mathsf{x}\colon T$, but also subtyping constraints $T <: T$ and type variables X. We will use the notation $\Gamma \Vdash \Delta$ to express that in context Γ, every constraint in Δ holds. Given that the context Γ contains type constraints as well, this amounts to checking *entailment* of constraint sets.

Well-Formed Types. Well-formedness of types is specified in Figure 6.13. In order to be well-formed, a type reference has to be either one of the magic types, a type variable that is in scope or a type reference that has the right number of type parameters which moreover satisfy the constraints in the given context.

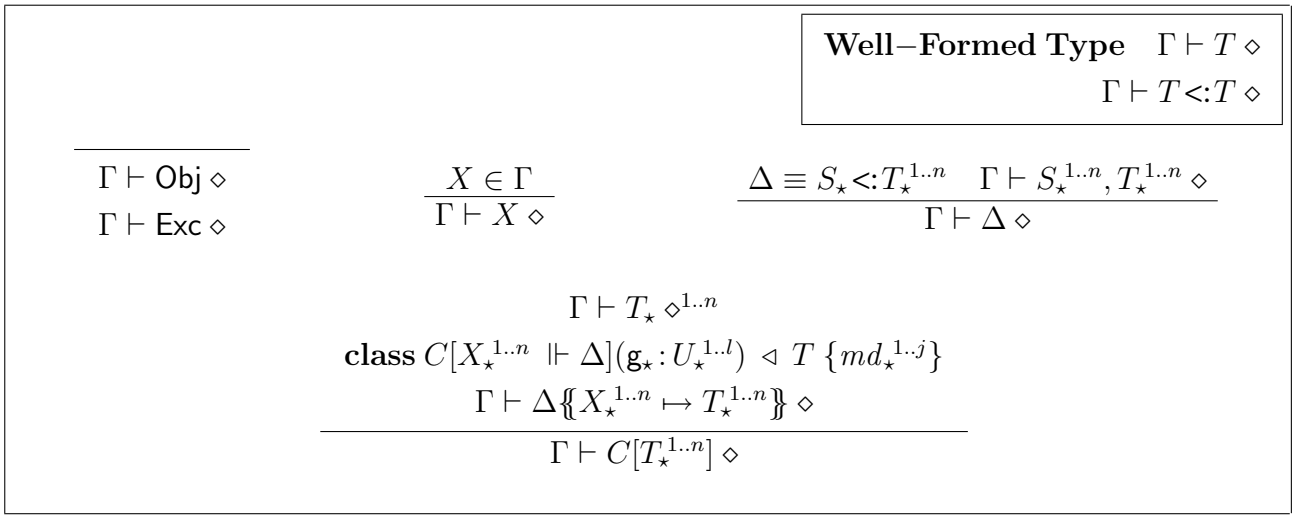


Figure 6.13: Well-formed Types

Expression Typing. Let us have a look at each typing rule. Rule (Tvar) and (Tthr) are straightforward and do not need explanation. Rule (Tfld) describes well-typed field access. The auxiliary judgment $\mathsf{fields}(T)$ performs type substitution in order to calculate field types in presence of parametric polymorphism.

Rule (Tinvk) describes well-typed method invocation. The auxiliary judgment $\mathsf{mtype}(\mathsf{m}, T)$ now substitutes actual type arguments of the receiver type T_0, producing a specialized signature of m. The type parameters of the method have to satisfy the type constraints, which is checked by entailment in $\Gamma \Vdash \Delta\Theta$. We do not need to check that the argument types are subtypes in the given context, as in $\Gamma \vdash S_\star <: T_\star\Theta^{1..n}$, because we are defining a declarative system with the (Tsub) rule. An implementation would check this condition at this precise point.

Rule (Tnew) specifies well-typed object construction. As for method invocation, the type parameters of the class have to satisfy the type constraints, as in $\Gamma \Vdash \Delta\Theta$. Again, an implementation would check here that argument types are subtypes in the given contexts, as in $\Gamma \vdash S_\star <: T_\star\Theta^{1..n}$.

Rule (Ttst) is for type-checking type-tests. The type variables $X_\star^{1..n}$ are fresh names for the formal type parameters $Y_\star^{1..n}$ of class C, which are required to satisfy constraints Δ. Consequently, the succeeding branch of the test is type-checked in a new type environment Γ', which extends Γ with type variables $X_\star^{1..n}$, constraints $\Delta\Theta$ and binding $x : C[X_\star^{1..n}]$. Having typed the succeeding and failing branch of the type-test with R and S respectively, the result is assigned the least upper bound $R \sqcup S$.

Rule (Tmch) achieves something similar for case clauses in match expressions – a match expression is well-typed if all its case clauses are well-typed. The type of the match expression is the least upper bound of the result types of all case clauses.

Expression Typing $\Gamma \vdash e \in C$

$$\frac{}{\Gamma \vdash \mathrm{x} \in \Gamma(\mathrm{x})}\ \textsf{(Tvar)} \qquad \frac{\Gamma \vdash T <: U \quad \Gamma \vdash U\ \diamond \quad \Gamma \vdash e \in T}{\Gamma \vdash e \in U}\ \textsf{(Tsub)}$$

$$\frac{}{\Gamma \vdash \mathbf{null} \in \mathbf{Exc}}\ \textsf{(Tnul)} \qquad \frac{}{\Gamma \vdash \mathbf{throw} \in \mathbf{Exc}}\ \textsf{(Tthr)}$$

$$\frac{\Gamma \vdash e_0 \in T_0 \quad \mathsf{fields}(T_0) = \mathsf{f}_\star : T_\star^{1..n}}{\Gamma \vdash e_0.\mathsf{f}_i \in T_i}\ \textsf{(Tfld)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 \in T \quad \Gamma \vdash e_\star \in T_\star\,\Theta^{1..n} \\ \mathsf{mtype}(\mathsf{m}, T_0) = an[X_\star^{1..m} \Vdash \Delta](T_\star^{1..n})T \\ \Theta = \{\!\{X_\star \mapsto R_\star^{1..m}\}\!\} \quad \Gamma \Vdash \Delta\,\Theta\end{array}}{\Gamma \vdash e_0.\mathsf{m}[R_\star^{1..m}](e_\star^{1..n}) \in T\,\Theta}\ \textsf{(Tinvk)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_\star \in T_\star\,\Theta^{1..n} \\ \mathsf{fields}(C[R_\star^{1..m}]) = \mathsf{f}_\star : T_\star^{1..n} \\ \mathbf{class}\ C[X_\star^{1..m} \Vdash \Delta](\mathsf{g}_\star : U_\star^{1..l}) \lhd T\ \{md_\star^{1..j}\} \\ \Theta = \{\!\{X_\star \mapsto R_\star^{1..m}\}\!\} \quad \Gamma \Vdash \mathcal{E}\Theta\end{array}}{\Gamma \vdash C[R_\star^{1..m}](e_\star^{1..n}) \in C[R_\star^{1..m}]}\ \textsf{(Tnew)}$$

$$\frac{\begin{array}{c}\mathbf{class}\ C[Y_\star^{1..n} \Vdash \Delta](\mathsf{g}_\star : U_\star^{1..m}) \lhd T\ \{md_\star^{1..l}\} \\ \Gamma' = \Gamma, X_\star^{1..n}, \Delta\{\!\{Y_\star \mapsto X_\star^{1..n}\}\!\}, \mathsf{x}\colon C[X_\star^{1..n}] \\ \Gamma \vdash e \in U \quad \Gamma' \vdash a \in R \quad \Gamma \vdash b \in S\end{array}}{\Gamma \vdash e?\{\mathsf{x}\colon C[X_\star^{1..n}] \Rightarrow a\}/\{b\} \in R \sqcup S}\ \textsf{(Ttst)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_\star \in R_\star^{1..n} \\ \Gamma; R_\star^{1..n} \vdash c_\star \in S_\star^{1..m}\end{array}}{\Gamma \vdash e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in \bigsqcup S_\star^{1..m}}\ \textsf{(Tmch)}$$

Figure 6.14: GPat Expression Typing

Pattern and Case Typing $\Gamma; R_\star^{1..n} \vdash c \in S$ / $\Gamma; R \ni p \dashv \Gamma'$

$$\frac{}{\Gamma; R \ni \mathsf{x} \dashv \mathsf{x}\colon R}\ (\mathsf{TPvar})$$

$$\frac{\begin{array}{c}\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}[Y_\star^{1..n}]) = \Delta(C[Y_\star^{1..n}])\mathsf{Tuple}_l[S_\star^{1..l}] \\ X fresh \\ \Gamma' = \Gamma, Y_\star^{1..n}, \Delta, X <: R, X <: C[Y_\star^{1..n}] \quad \Gamma'; S_\star \ni p_\star \dashv \Gamma_\star^{1..m}\end{array}}{\Gamma; R \ni \hat{v}.\mathsf{m}[Y_\star^{1..n}](p_\star^{1..m}) \dashv \Gamma', \Gamma_\star^{1..m}}\ (\mathsf{TPext})$$

$$\frac{\begin{array}{c}\Gamma; R_\star \ni p_\star \dashv \Gamma_\star^{1..n} \\ \Gamma, \Gamma_\star^{1..n} \vdash b \in T\end{array}}{\Gamma; R_\star \vdash \mathbf{case}\ p_\star^{1..n} \Rightarrow b \in T}\ (\mathsf{Tcase})$$

Extractor Type $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}[Y_\star^{1..n}])$

$$\frac{\begin{array}{c}\Gamma \vdash \hat{v} \in R \quad \Theta_X = \{\!\{X_\star^{1..n} \mapsto Y_\star^{1..n}\}\!\} \quad \Theta_Z = \{\!\{Z_\star^{1..n} \mapsto Y_\star^{1..n}\}\!\} \\ \mathbf{class}\ C[X_\star^{1..l}\ \Vdash \Delta_C](\mathsf{f}_\star\colon S_\star^{1..m})\ \lhd\ T\ \{an_\star md_\star^{1..k}\} \\ \mathsf{mtype}(\mathsf{m}, R) = @\mathbf{safe}[Z_\star^{1..n}\ \Vdash \Delta_\mathsf{m}](C[Z_\star^{1..n}])\mathsf{Tuple}_l[R_\star^{1..l}] \\ Y_\star^{1..n}, \Delta_\mathsf{m}\,\Theta_Z\ \Vdash \Delta_C\,\Theta_X \quad Y_\star^{1..n}, \Delta_C\,\Theta_X\ \Vdash \Delta_\mathsf{m}\,\Theta_Z\end{array}}{\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}[Y_\star^{1..n}]) = \Delta_C\,\Theta_X(C[Y_\star^{1..n}])\mathsf{Tuple}_l[R_\star\,\Theta^{1..l}]}$$

Figure 6.15: GPat Pattern and Case Typing

Method Typing $md\ \diamond\ \text{in}\ C$

$$\frac{\begin{array}{c}\Gamma = X_\star^{1..k}, Y_\star^{1..m}, \Delta_C, \Delta_\mathsf{m} \\ \Gamma, \mathsf{this}\colon C[X_\star^{1..k}], \mathsf{x}_\star\colon T_\star^{1..n} \vdash e \in T \quad \Gamma \vdash T <: T_0 \quad \Gamma \vdash T_0, T_\star^{1..n}, \Delta_\mathsf{m}\ \diamond \\ \mathbf{class}\ C[X_\star^{1..k}\ \Vdash \Delta_C](\mathsf{f}_\star\colon D_\star^{1..l})\ \lhd\ S\ \{md_\star^{1..k}\} \\ \mathsf{override}(an[Y_\star^{1..m}\ \Vdash \Delta_\mathsf{m}](T_\star^{1..n})T_0, \mathsf{m}, S)\end{array}}{an\ \mathbf{def}\ \mathsf{m}[Y_\star^{1..m}\ \Vdash \Delta_\mathsf{m}](\mathsf{x}_\star\colon T_\star^{1..n})\colon T_0 = \{e\}\ \diamond\ \text{in}\ C}$$

Class Typing $cd\ \diamond$

$$\frac{\begin{array}{c}an_\star md_\star\ \diamond\ \text{in}\ C^{1..k} \\ X_\star^{1..l}, \Delta \vdash T, T_\star^{1..n}, \Delta\ \diamond\end{array}}{\mathbf{class}\ C[X_\star^{1..l}\ \Vdash \Delta](\mathsf{f}_\star\colon T_\star^{1..n})\ \lhd\ T\ \{an_\star md_\star^{1..k}\}\ \diamond}\ (\mathsf{Tc})$$

Figure 6.16: GPat Method and Class Typing

Pattern and Case Typing. Pattern typing $\Gamma; R \ni \mathsf{x} \dashv \Gamma'$ is done with respect to an "expected type" so we can assign a type to pattern variables, and produces a context Σ. This happens in rule (TPvar).

Rule (TPext) obtains the "extractor type signature" $\Delta(R)S$ which consists of a set of constraints Δ, an argument type R and a result type S that must be a tuple $\mathsf{Tuple}_l[S_\star^{1..l}]$. Tuple types play the role of the case classes and the $\mathsf{casefld}$ judgment played in the first-order version. The new context $\Gamma' = \Gamma, Y_\star^{1..n}, \Delta$ is used to type-check the sub-patterns as in $\Gamma'; S_\star\Theta \ni p_\star \dashv \Sigma_\star{}^{1..m}$. We merge the resulting environments and additionally add the subtype constraints that come from the extractor call. This results in a context $\Gamma', \Sigma_\star^{1..n}$. The fresh type variable X has the sole purpose of placing both of its upper bounds in the context. By rule (Sdeext), any relation between the expected type and the pattern type that can be derived from the class hierarchy can then be used to obtain precise bounds for type variables in either.

Using these pattern typing rules, case clauses are checked using (Tcase). Each pattern of the clause is type-checked, as in $\Gamma; R_\star \ni p_\star \dashv \Sigma_\star^{1..n}$, and the resulting contexts (including constraints) are merged to type-check the body b, as in $\Gamma, \Sigma_\star^{1..n} \vdash b \in T$.

The extractor type signature judgment $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}[Y_\star^{1..n}])$ looks up the extractor method m from the static type R of the receiver $\hat{v}$. The argument type of the extractor is some type reference $C[Z_\star^{1..n}]$, whose actual type parameters are exactly the formal type parameters of the extractor. Whereas in the first-order case, we used type Obj, here extractors have a more specific type which is going to be tested prior to the extractor call. It is then necessary to ensure that the constraints of the extractor call are equivalent of the constraints of class C, since the type test that is going to be generated in pattern matching translation can only test for the head type C. The rule could have been simplified if we demanded syntactic equality between constraints Δ_m and Δ_C, but this would require using the same type parameter names in the extractor as in the definitions of its argument type. So instead, we use two substitutions Θ_X and Θ_Z that unify type constraints from extractor and class definition by renaming their parameters to type variables from the extractor call. Equivalence is then checked using by checking entailment in both directions, as in $Y_\star^{1..n}, \Delta_\mathsf{m}\Theta_Z \Vdash \Delta_C\Theta_X$ and $Y_\star^{1..n}, \Delta_C\Theta_X \Vdash \Delta_\mathsf{m}\Theta_Z$. The resulting extractor type signature is then returned, with its components substituted to work in the context that is constructed in the pattern typing rule.

Method and Class Typing. The typing of methods and classes is straightforward. In addition to checking each method definitions, the well-formedness of types has to be checked as well. The $\mathsf{override}(an[Y_\star^{1..m} \Vdash \Delta_\mathsf{m}](T_\star^{1..n})T_0, \mathsf{m}, S)$ ensures that methods are overridden with the same annotations, type parameters, constraints, argument types and return type.

Overriding $\mathsf{override}(an[X_\star^{1..k} \Vdash \Delta](B_\star^{1..n})B, \mathsf{m}, T)$

$$\frac{\mathsf{mtype}(\mathsf{m}, T) = an[X_\star^{1..k} \Vdash \Delta](B_\star^{1..n})B \text{ or undefined}}{\mathsf{override}(an[X_\star^{1..k} \Vdash \Delta](B_\star^{1..n})B, \mathsf{m}, T)}$$

Method Lookup $\mathsf{mtype}(\mathsf{m}, T)$ $\mathsf{mbody}(\mathsf{m}, T)$

$$\frac{\begin{array}{c}\mathbf{class}\ C[X_\star^{1..l} \Vdash \Delta_C](\mathsf{f}_\star : C_\star^{1..m}) \lhd T\ \{md_\star^{1..k}\} \\ an_i \equiv an \\ md_i \equiv \mathbf{def}\ \mathsf{m}[Y_\star^{1..j} \Vdash \Delta_\mathsf{m}](\mathsf{x}_\star : B_\star^{1..n}) : B = \{e\} \\ \Theta = \{\!\{X_\star^{1..l} \mapsto R_\star^{1..l}\}\!\}\end{array}}{\begin{array}{rl}\mathsf{mtype}(\mathsf{m}, C[R_\star^{1..l}]) & = an[Y_\star^{1..j} \Vdash \Delta_\mathsf{m}\Theta](x_\star : B\Theta_\star^{1..n})B\Theta \\ \mathsf{mbody}(\mathsf{m}_i, C[R_\star^{1..l}]) & = [Y_\star^{1..j}](x_\star^{1..n})e\end{array}}$$

$$\frac{\begin{array}{c}\mathbf{class}\ C[X_\star^{1..l} \Vdash \Delta](\mathsf{f}_\star : C_\star^{1..m}) \lhd T\ \{md_\star^{1..k}\} \\ \mathsf{m} \notin md_\star^{1..k} \quad \Theta = \{\!\{X_\star^{1..l} \mapsto R_\star^{1..l}\}\!\}\end{array}}{\begin{array}{rl}\mathsf{mtype}(\mathsf{m}, C[R_\star^{1..l}]) & = \mathsf{mtype}(\mathsf{m}, T\Theta) \\ \mathsf{mbody}(\mathsf{m}, C[R_\star^{1..l}]) & = \mathsf{mbody}(\mathsf{m}, T\Theta)\end{array}}$$

Field Lookup $\mathsf{fields}(T)$

$$\frac{}{\mathsf{fields}(\mathsf{Obj}) = \bullet}$$

$$\frac{\begin{array}{c}\mathsf{fields}(T\Theta) = \mathsf{f}_\star : S_\star^{1..m} \\ \mathbf{class}\ C[X_\star^{1..l} \Vdash \Delta](\mathsf{g}_\star : T_\star^{1..n}) \lhd T\ \{an_\star md_\star^{1..k}\} \\ \Theta = \{\!\{X_\star^{1..l} \mapsto R_\star^{1..l}\}\!\}\end{array}}{\mathsf{fields}(C[R_\star^{1..l}]) = \mathsf{f}_\star : S_\star^{1..m}; \mathsf{g} : T_\star\Theta^{1..n}}$$

Figure 6.17: GPat Auxiliary Judgments for Overriding, Method and Field Lookup

Auxiliary Judgments. Auxiliary judgments are given to check valid overriding, method and field lookup. The overriding check is demanding syntactic equality between the signature of the overridden and the overriding method. This restriction is not necessary but seems reasonable in the interest of readability. Method lookup and field lookup are straightforward, as mentioned above the only difference to the first-order case is that a type substitution is performed on the results.

6.6.5 Divergent Programs

Figure 6.18 contains the coinductive rules that characterize divergent programs. The rules are all straightforward adaptations of the first-order versions.

6.7 Type Soundness

We can now prove type soundness, in the same way we proved it for the first-order calculus.

Lemma 17 (Uniqueness) *For all a, if $a \Downarrow q$ then for all q', if $a \Downarrow q'$ then $q = q'$.*

Proof By induction on $a \Downarrow q$ and case analysis on q'. □

Lemma 18 (Closed Types) *The following statements on closed types hold:*

- *If* $\bullet \vdash C[S_\star^{1..m}] <: C[T_\star^{1..m}]$ *then* $S_\star \equiv T_\star$.
- *If* $\bullet \vdash S <: T$ *then* $S \equiv C[S_\star^{1..m}]$, $T \equiv D[T_\star^{1..n}]$ *and* $S \lhd^* T$.

Proof Since the context is empty, S, T are closed. Since they are well-formed, they do not contain type variables and thus must have the desired shape (this includes special cases Obj and Exc). A small derivation of $\bullet \vdash S <: T$ can only be derived using (Scon),(Sext),(Sthr) and (Sobj). □

Lemma 19 (Values) *For all v, if $\bullet \vdash v \in T$ then $v \equiv C[T_\star^{1..m}](\dot{v}_\star^{1..n})$ with $C[T_\star^{1..m}] \lhd^* T$*

Proof By rule (Tnew) and **(Closed Types)**. □

Lemma 20 (Termination) *For all a and all q, it holds that if $a \Downarrow q$ then $a \not\Uparrow$.*

Divergent Computation $e \Uparrow$

$$\frac{e \Uparrow}{e.\mathsf{f} \Uparrow}\ (\mathsf{Dfld}) \qquad \frac{e \Uparrow}{e.\mathsf{m}[T_\star^{1..n}](e_\star^{1..n}) \Uparrow}\ (\mathsf{Drcv}) \qquad \frac{e \Downarrow v \quad e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Uparrow}{e.\mathsf{m}[T_\star^{1..n}](e_\star^{1..n}) \Uparrow}\ (\mathsf{Darg})$$

$$\frac{\begin{array}{c} e \Downarrow C[S_\star^{1..m}](\dot{v}_\star^{1..m}) \quad e_\star \Downarrow \dot{w}_\star{}^{1..n} \\ \mathsf{mbody}(\mathsf{m}, C[S_\star^{1..m}]) = [Y_\star^{1..n}](\mathsf{x}_\star^{1..n})b \\ b\,\{\!\{Y_\star^{1..n} \mapsto T_\star^{1..n}\}\!\}\,\{\mathsf{this} \mapsto C(\dot{v}_\star^{1..m}), \mathsf{x}_\star \mapsto \dot{w}_\star{}^{1..n}\} \Uparrow \end{array}}{e.\mathsf{m}[T_\star^{1..n}](e_\star^{1..n}) \Uparrow}\ (\mathsf{Dinvk})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Uparrow}{C[S_\star^{1..m}](e_\star^{1..n}) \Uparrow}\ (\mathsf{Dnew}) \qquad \frac{e \Uparrow}{e?\{\mathsf{x}\colon D[Y_\star^{1..n}] \Rightarrow a\}/\{b\} \Uparrow}\ (\mathsf{Dtst})$$

$$\frac{\begin{array}{c} e \Downarrow C[S_\star^{1..m}](\dot{v}_\star^{1..n}) \quad C[S_\star^{1..m}] \lhd^\star D[T_\star^{1..n}] \\ a\,\{\!\{Y_\star^{1..n} \mapsto T_\star^{1..n}\}\!\}\,\{\mathsf{x} \mapsto C[S_\star^{1..m}](\dot{v}_\star^{1..n})\} \Uparrow \end{array}}{e?\{\mathsf{x}\colon D[Y_\star^{1..n}] \Rightarrow a\}/\{b\} \Uparrow}\ (\mathsf{Dcst})$$

$$\frac{e \Downarrow \mathbf{null}\ \mathit{or}\ \left(e \Downarrow C[S_\star^{1..m}](\dot{v}_\star^{1..n}) \quad C[S_\star^{1..m}] \not\lhd^\star D[T_\star^{1..n}]\right) \quad b \Uparrow}{e?\{\mathsf{x}\colon D[Y_\star^{1..n}] \Rightarrow a\}/\{b\} \Uparrow}\ (\mathsf{Dskp})$$

$$\frac{e_\star \Downarrow \dot{v}_\star^{1..i-1} \quad e_i \Uparrow}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \Uparrow}\ (\mathsf{Dmch})$$

$$\frac{\begin{array}{c} e_\star \Downarrow \dot{v}_\star^{1..n} \quad \forall j < i\ .\ \dot{v}_\star^{1..n} \curvearrowright c_j \dashv \mathbf{reject} \\ \dot{v}_\star^{1..n}; c_i \Uparrow e \end{array}}{e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \Uparrow}\ (\mathsf{Dcase})$$

Divergent Cases and Patterns $\dot{v}_\star^{1..n}; c \Uparrow e$

$$\frac{c = \mathbf{case}\ p_\star^{1..n} \Rightarrow b \quad \dot{v}_\star \curvearrowright p_\star \dashv \Theta_\star, \sigma_\star^{1..n} \quad b\,\Theta_\star^{1..n}\,\sigma_\star^{1..n} \Uparrow}{\dot{v}_\star^{1..n}; c \Uparrow b\,\sigma_\star^{1..n}}\ (\mathsf{Dbdy})$$

Figure 6.18: GPat Divergence Rules

Proof By induction on $a \Downarrow q$ and inversion of $a \Uparrow$. □

Lemma 21 (Substitution Property for Lookup) *Let* $\Theta = \{\!\{Y_\star^{1..k} \mapsto U_\star^{1..k}\}\!\}$ *and* $S = C[T_\star^{1..m}]$. *The following holds:*

- *If* $\mathsf{fields}(T) = \mathsf{f}_\star : S_\star^{1..n}$ *then* $\mathsf{fields}(T\Theta) = \mathsf{f}_\star : S_\star^{1..n}\Theta$
- *If* $\mathsf{mtype}(\mathsf{m}, S) = an[X_\star^{1..m} \Vdash \Delta](T_\star^{1..n})T$ *then*
  $\mathsf{mtype}(\mathsf{m}, S\Theta) = an[X_\star^{1..m} \Vdash \Delta\Theta](T_\star\Theta^{1..n})T\Theta$.
- *If* $\mathsf{mtype}(\mathsf{m}, S)$ *is undefined then* $\mathsf{mtype}(\mathsf{m}, S\Theta)$ *is undefined as well.*

Proof The first two statements are proven by induction on derivation of fields and mtype. The last statement is clear from the observation that Θ does not alter which classes are traversed when going from a class to a superclass.

Lemma 22 (Type Substitution Property for Types) *Let* $\mathcal{J}$ *range over judgment forms of subtyping* $(S{<:}T)$, *well-formedness* $T\diamond$ *and typing* $e \in T$. *If* $X_\star^{1..m}, Y_\star^{1..n}, \mathsf{x}_\star : T_\star^{1..j}, \Delta \vdash \mathcal{J}$ *and* $\Theta = \{\!\{Y_\star^{1..n} \mapsto U_\star^{1..n}\}\!\}$ *then* $X_\star^{1..m}, \mathsf{x}_\star : T\Theta_\star^{1..j}, \Delta\Theta \vdash \mathcal{J}\Theta$.

Proof Straightforward induction on the derivation of $\mathcal{J}$, using **(Type Substitution Property for Lookup)**. □

The following lemma lets us discharge constraint set that were already proven. It is used whenever we have more precise bounds and the induction hypothesis holds for weaker bounds in the context.

Lemma 23 (Constraint Elimination) *Let* $\mathcal{J}$ *range over judgment forms of subtyping* $(S{<:}T)$, *well-formedness* $T\diamond$ *and typing* $e \in T$. *If* $\Gamma, \Delta \vdash \mathcal{J}$ *and* $\Gamma \Vdash \Delta$, *then* $\Gamma \Vdash \mathcal{J}$.

Proof Induction on the derivation of $\mathcal{J}$. □

Lemma 24 (Subtypes have all Fields) *If* $\bullet \vdash C[S_\star^{1..k}]{<:}D[U_\star^{1..m}]$, $C \neq \mathsf{Exc}$ *then* $\mathsf{fields}(C[S_\star^{1..k}]) = \mathsf{fields}(D[U_\star^{1..m}]); \mathsf{g}_\star : E_\star^{1..m}$.

Proof We will prefer arguing with the algorithmic jugdment $\Psi \succ C[S_\star^{1..k}]{<:}D[U_\star^{1..m}]$ for suitable Ψ, which (apart from the special cases for magic classes) leaves us with a trivial (Scon) or non-trivial (Sext) inheritance relationship. By case analysis on the derivation of $\Psi \succ C[S_\star^{1..k}]{<:}D[U_\star^{1..m}]$.

Case (Sobj) Then $\mathsf{fields}(\mathsf{Obj}) = \bullet$

Case (Scon) Trivial

Case (Sext) Then the definition of fields is applied

The other cases cannot happen. □

Lemma 25 (Subtypes have all Methods) *If* $C[S_\star^{1..k}] <: D[U_\star^{1..m}$, $C \neq \mathsf{Exc}$ *and* $\mathsf{mtype}(\mathsf{m}, D[U_\star^{1..m}]) = an[Y_\star^{1..k} \Vdash \Delta](T_\star^{1..n})T$, *then* $\mathsf{mtype}(\mathsf{m}, D[U_\star^{1..m}]) = an[Y_\star^{1..k} \Vdash \Delta](T_\star^{1..n})T$.

Proof We again use the algorithmic jugdment $\Psi \succ C[S_\star^{1..k}] <: D[U_\star^{1..m}]$ for suitable Ψ and the trivial (Scon) or non-trivial (Sext) inheritance relationship. By induction on the derivation of $\mathsf{mtype}(\mathsf{m}, D[U_\star^{1..m}]) = an[Y_\star^{1..k} \Vdash \Delta](T_\star^{1..n})T$ and case analysis over $\Psi \succ C[S_\star^{1..k}] <: D[U_\star^{1..m}]$.

Case (Sobj) Cannot happen, since Obj has no methods.

Case (Scon) Trivial

Case (Sext) If C does not contain a definition for m, then the definition of mtype is applied. Otherwise, class definition of C is well-typed, thus $\mathsf{override}(an(C_\star^{1..n})B, \mathsf{m}, D)$ asserts that $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{mtype}(\mathsf{m}, D)$.

The other cases cannot happen. □

We now turn to generalizations of lemmata for the pattern matching construct. Since the typing rules involve collecting a type context, we want to show that a suitable context is returned when type substitution takes place. Fortunately, the subsumption rule allows us to simplify the lemma a bit with respect to the first-order version. Also, we express that the resulting context is stronger, using the notion of entailment, while the notation $\Gamma <: \Gamma'$ still expresses that term variables have "better" types.

Lemma 26 (Subtypes yield Refined Environment)
If $\Gamma \vdash S <: T$ *and* $\Gamma; T \ni p \dashv \Gamma'$ *then* $\Gamma; S \ni p \dashv \Gamma''$ *for some* $\Gamma'' <: \Gamma'$ *and* $\Gamma, \Gamma'' \Vdash \Gamma, \Gamma'$.

Proof By induction on $\Gamma; T \ni p \dashv \Gamma'$

Case (TPvar) Then $\Gamma; T \ni \mathsf{x} \dashv \{\mathsf{x}\colon T\}$ and we can also derive $\Gamma; S \ni \mathsf{x} \dashv \{\mathsf{x}\colon S\}$. From $\Gamma \vdash S <: T$ follows $\{\mathsf{x}\colon S\} <: \{\mathsf{x}\colon T\}$

Case (TPextr) Then $\Gamma; T \ni \hat{v}.\mathsf{m}[Y_\star^{1..k}](p_\star^{1..n}) \dashv {\Gamma'}_\star^{1..n}$ and subpatterns have derivations $\Gamma; R_\star \ni p_\star \dashv {\Gamma'}_\star^{1..n}$ for some types $R_\star^{1..n}$ and there is a type substitution $\Theta = \{\!\{Y_\star \mapsto U_\star^{1..k}\}\!\}$ for the extractor call. The expected type is only used as an upper bound for the fresh type

variable X for typing the subpatterns, and by **(Constraint Elimination)** the subderivations can be reused, yielding $\Gamma; S \ni \hat{v}.\mathsf{m}(p_\star^{1..n}) \dashv \Gamma'_\star{}^{1..n}$.

The only difference between the new context Γ'' and Γ' is that the constraint $X <: T$ has been replaced with $X <: S$. Since $\Gamma \vdash S <: T$ was assumed, clearly $\Gamma'' <: \Gamma'$ and $\Gamma, \Gamma'' \Vdash \Gamma, \Gamma'$ □

Lemma 27 (Refined Environment preserves Typing) *If* $S_\star <: T_\star^{1..n}$ *and* $\Gamma, \mathsf{x}_\star : T_\star^{1..n} \vdash e \in U$ *then* $\Gamma, \mathsf{x}_\star : S_\star^{1..n} \vdash e \in U$.

Proof By straightforward induction on $\Gamma, \mathsf{x}_\star : T_\star^{1..n} \vdash e \in U$, using (Tsub) where necessary. □

Lemma 28 (Weakening) *If* $\Gamma \vdash d \in T$ *and* $\mathsf{x} \notin fv(d)$, *then* $\Gamma, \mathsf{x}: S \vdash d \in T$ *for any* S.

Proof Straightforward induction on $\Gamma \vdash d \in T$. □

Lemma 29 (Term Substitution Lemma) *If* $\Gamma, \mathsf{x}_\star : T_\star^{1..n} \vdash b \in U$ *and* $\bullet \vdash \dot{u}_\star \in T_\star^{1..n}$, $\dot{u}_\star \in \mathsf{Values} \cup \{\mathbf{null}\}$ *then* $\Gamma \vdash b\{\mathsf{x}_\star \mapsto \dot{u}_\star{}^{1..n}\} \in U$.

Proof By induction on the derivation of $\Gamma, \mathsf{x}_\star : T_\star^{1..m} \vdash b \in U$. Let $\sigma = \{\mathsf{x}_\star \mapsto \dot{u}_\star{}^{1..n}\}$ and $\Gamma' = \Gamma, \mathsf{x}_\star : T_\star^{1..m}$.

Case (Tvar) $b \equiv \mathsf{x}$

i) If $\mathsf{x} = \mathsf{x}_i$ for some i, then $\mathsf{x}\sigma = \dot{u}_i$ with $\Gamma \vdash a_i \in T_i$ by assumption, **(Weakening)**.

ii) Otherwise, $\mathsf{x}\sigma = \mathsf{x}$ and rule (Tvar).

Case (Tthr),(Tnul) trivial because $b\sigma \equiv b$

Case (Tfld) $b \equiv e.\mathsf{f}$ We have $\Gamma' \vdash e \in R$ and i.h. yields $\Gamma \vdash e\sigma \in R$. (Tfld) finishes the case.

Case (Tinvk) $b \equiv e.\mathsf{m}(e_\star^{1..n})$ We have $\Gamma' \vdash e \in R$ and $\mathsf{mtype}(\mathsf{m}, R) = an(U_\star^{1..n})U$. The i.h. yields $\Gamma \vdash e\sigma \in R$. We also have $\Gamma' \vdash e_\star \in U_\star^{1..n}$ and i.h. yields $\Gamma \vdash e_\star\sigma \in U_\star^{1..n}$. Rule (Tinvk) finishes the case.

Case (Tnew) $b \equiv C[U_\star^{1..m}](e_\star^{1..n})$ We have $\mathsf{fields}(C[U_\star^{1..m}]) = R_\star^{1..n}$ and $\Gamma' \vdash e_\star \in R_\star^{1..n}$. The i.h. yields $\Gamma \vdash e_\star\sigma \in R_\star^{1..n}$. Rule (Tnew) finish the case.

Case (Ttst) $b \equiv a?\{\mathsf{x}: C[U_\star^{1..m}] \Rightarrow d\}/\{e\}$ We have $\Gamma' \vdash a \in S$, $\Gamma' \vdash d \in R_1$, and $\Gamma' \vdash e \in R_0$ and $R_\star <: U^{0,1}$. The i.h. yields $\Gamma \vdash a\sigma \in S$, $\Gamma \vdash d\sigma \in R_1$, and $\Gamma \vdash e\sigma \in R_0$. Transitivity of $<:$ and (Ttst) finishes the case.

Case (Tmch) $b \equiv e_\star^{1..n}$ **match** $\{c_\star^{1..m}\}$ We have $\Gamma' \vdash e_\star \in R_\star^{1..n}$ and i.h. yields $\Gamma \vdash e_\star\sigma \in R_\star^{1..n}$.

For each $j \in 1..m$, let $c_j \equiv$ **case** $p_\star^{1..n} \Rightarrow b_j$ (we omit the extra j index for patterns). We have a case typing $\Gamma'; R_\star^{1..m} \vdash c_j \in U_j$ via $\Gamma'; R_\star \ni p_\star \dashv \Gamma''^{1..n}_\star$ and $\Gamma', \Gamma''^{1..n}_\star \vdash b \in U_j$.

By **(Subtypes yield Refined Environment)**, we get $\Gamma'; R'_\star \ni p_\star \dashv \Gamma'''^{1..n}_\star$. for $\Gamma'''_\star <: \Gamma''^{1..n}_\star$.

By **(Refined Environment preserves Typing)** we get $\Gamma, \Gamma''^{1..n}_\star \vdash b_j \in U_j$.

Applying the i.h. yields $\Gamma, \Gamma'''^{1..n}_\star \vdash b_j\sigma \in U_j$. Rule (Tmch) finishes the case. □

Lemma 30 (Preservation)
If $a \Downarrow q$ and $\bullet \vdash a \in T$, then $\bullet \vdash q \in T$.

Proof For $q \equiv$ **throw** and $q \equiv$ **null**, rules (Tthr) and (Tnul) and (Tsub) yield the proof. Otherwise, induction on $a \Downarrow v$.

Case (Rfld) $a \equiv e.\mathsf{f}_i$

The premises of (Tfld) are $\bullet \vdash e \in T_0$ and $\mathsf{fields}(T_0) = \mathsf{f}_\star : T_\star^{1..m}$ with $T = T_i$.

We have $e \Downarrow D[S_\star^{1..m}](\dot{w}_\star^{1..n})$ and $v = \dot{w}_i$.

By i.h. $\bullet \vdash D[S_\star^{1..m}](\dot{w}_\star^{1..n}) \in D[S_\star^{1..m}]$ for $D[S_\star^{1..m}] <: T_0$.

By **(Subtypes have all Fields)**, we obtain $m \leq n$ and $f_i \in \mathsf{fields}(D[S_\star^{1..m}])$.

Finally, from $\bullet \vdash D[S_\star^{1..m}](\dot{w}_\star^{1..n}) \in D[S_\star^{1..m}]$ and rule (Tnew) we arrive at $\bullet \vdash \dot{w}_i \in T_i$ and thus $\bullet \vdash v \in T$.

Case (Rinvk) $a \equiv e.\mathsf{m}[R_\star^{1..k}](e_\star^{1..n})$

The premises of (Tinvk) are $\Gamma \vdash e \in U$, $\mathsf{mtype}(\mathsf{m}, U) = an[Y_\star^{1..k} \Vdash \Delta](S_\star^{1..n})S$, $\bullet \vdash e_\star \in S_\star^{1..n}$. The type parameters satisfy the constraints, i.e. for type substitution $\Theta = \{\!\{Y_\star^{1..k} \mapsto R_\star^{1..k}\}\!\}$, we have $\bullet \Vdash \Delta\Theta$.

Then $e \Downarrow D[U_\star^{1..j}](\dot{w}_\star^{1..m})$, $e_\star \Downarrow \dot{v}_\star^{1..n}$, $\mathsf{mbody}(\mathsf{m}, D[U_\star^{1..j}]) = [Y_\star^{1..k}](\mathsf{x}_\star^{1..n})e_0$ with $\bullet \vdash e_0 \in T$. Using substitution $\sigma = \{\mathsf{this} \mapsto D[U_\star^{1..j}](\dot{w}_\star^{1..m}), \mathsf{x}_\star \mapsto \dot{v}_\star^{1..n}\}$, we evaluate the body as $e_0\Theta\sigma \Downarrow v$.

Applying the i.h. for the receiver and **(Values)** yields $D[U_\star^{1..j}] <: U$.

By **(Subtypes have all Methods)** we get $\mathsf{mtype}(\mathsf{m}, D[U_\star^{1..j}]) = \mathsf{mtype}(\mathsf{m}, U)$.

Applying the i.h. for the arguments yields $\bullet \vdash \dot{v}_\star \in S_\star^{1..n}$.

By **(Type Substitution Property for Types)** and **(Substitution Lemma)** we get $\bullet \vdash e_0 \Theta \sigma \in T$.

Applying the i.h. for the body then yields $\bullet \vdash v \in T$.

Case (Rnew) $a \equiv D[{U_\star}^{1..j}](\dot{w}_\star^{1..n})$ then $a \Downarrow a$, and $D[{U_\star}^{1..j}] <: D[{U_\star}^{1..j}]$ by **(Closed Types)**.

Case (Rcst) $a \equiv e?\{\mathsf{x}\colon D[{Y_\star}^{1..k}] \Rightarrow b\}/\{d\}$

We have $e \Downarrow C[{R_\star}^{1..m}](\dot{w}_\star^{1..n})$ as well as $C[{R_\star}^{1..m}] \lhd^\star D[{S_\star}^{1..k}]$ and $b\,\{\!\{Y_\star \mapsto {S_\star}^{1..k}\}\!\}\,\{\mathsf{x} \mapsto C[{R_\star}^{1..m}](\dot{w}_\star^{1..n})\} \Downarrow v$.

Using typing premises from (Ttst), we apply the **(Type Substitution Property for Types)** and **(Substitution Lemma)** followed by the i.h.

Case (Rskp) $a \equiv e?\{\mathsf{x}\colon D[{Y_\star}^{1..k}] \Rightarrow b\}/\{d\}$

We have $e \Downarrow C[{R_\star}^{1..m}](w_\star^{1..n})$, $C[{R_\star}^{1..m}] \not\lhd^\star D[{S_\star}^{1..k}]$ for any ${S_\star}^{1..k}$ and $d \Downarrow v$.

Using typing premises from (Ttst), we apply the i.h. to d, yielding $\bullet \vdash d \in T$.

Case (Rmch) $a \equiv {e_\star}^{1..m}\ \mathbf{match}\ \{{c_\star}^{1..l}\}$. Let i be the index of the matching case.

The premises of (Tmch) include $\bullet \vdash e_\star \in {S_\star}^{1..m}$ and case typing $\bullet; {C_\star}^{1..m} \vdash c_i \in T_i$ for $T_i <: T$.

The case typing has premises $\bullet; S_\star \ni p_\star \dashv {\Gamma'_\star}^{1..n}$ and $\bullet; {\Gamma'_\star}^{1..m} \vdash b \in T_i$ where $c_i \equiv \mathbf{case}\ p_\star \Rightarrow b$.

We have $e_\star \Downarrow {\dot{v}_\star}^{1..m}$ as well as $\dot{v}_\star \curvearrowright p_\star \dashv \Theta_\star; {\sigma_\star}^{1..m}$ and $b\,\sigma_\star \Downarrow v$.

Applying the induction hypothesis to $e_\star$ yields $\bullet \vdash v_\star \in {S_\star}^{1..m}$.

By **(Type Substitution Property for Types)** and **(Substitution Lemma)**, $\bullet \vdash b{\Theta_\star}^{1..m}{\sigma_\star}^{1..m} \in T_i$.

Applying the i.h. yields $\bullet \vdash v \in T_i$ and by (Tsub) $\bullet \vdash v \in T$.

□

Lemma 31 (Progress)
If $\bullet \vdash a \in T$ and $a \not\Downarrow q$ for all q, then $a \Uparrow$.

Proof By coinduction and case analysis over the last rule used in a small derivation of $\bullet \vdash a \in T$.

Case $a \in \{\mathbf{throw}, \mathbf{null}\}$ and $a \equiv \mathsf{x}$ are not interesting, since $a \notin R$

Case $a \equiv e_0.\mathsf{f}$ and (Rfld), (Cfld) are blocked. By $\bullet \vdash a \in T$ and (Tfld), we also have $\bullet \vdash e_0 \in T_0$. Thus, either

i) $e_0 \not\Downarrow q_0$ for any q_0. This amounts to $e_0 \in R$ and shows that (Dfld) preserves R.

ii) $e_0 \Downarrow \mathbf{throw}$ or $e_0 \Downarrow \mathbf{null}$, but this contradicts (Cfld) blocked.

iii) $e_0 \Downarrow D(\dot{w}_\star^{1..n})$ but by **(Preservation)**and **(Subtypes have all Fields)**, this contradicts (Rfld) blocked.

Case $a \equiv e_0.\mathsf{m}[R_\star^{1..k}](e_\star^{1..n})$ and (Rinvk),(Crcv) and (Carg) are blocked. By $\bullet \vdash a \in T$ and (Tinvk), we also have $\bullet \vdash e_0 \in T_0, \mathsf{mtype}(\mathsf{m}, T_0) = an[Y_\star^{1..k} \Vdash \Delta](T_\star^{1..n})T$ and $\bullet \vdash e_\star \in T_\star\Theta^{1..n}$ for $\Theta = \{\!\{Y_\star \mapsto R_\star^{1..k}\}\!\}$

Thus, either

i) $e_0 \not\Downarrow q_0$ for any q_0. This amounts to $e_0 \in R$ and shows that (Drcv) preserves R.

ii) $e_0 \Downarrow \mathbf{throw}$ or $e_0 \Downarrow \mathbf{null}$ but this contradicts (Crcv) blocked.

iii) $e_0 \Downarrow D[U_\star^{1..j}](\dot{w}_\star^{1..n})$. By **(Preservation)** and **(Closed Types)**, $D[U_\star^{1..j}] <: T_0$ and by **(Subtypes have all Methods)**, $\mathsf{mbody}(\mathsf{m}, D[U_\star^{1..j}]) = [Y_\star^{1..k}](\mathsf{x}_\star^{1..n})b$. We can distinguish further

   a) There exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \not\Downarrow q_0$ for any q_0. Then $e_i \in R$ and (Darg) preserves R.

   b) There exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$, but this contradicts (Carg) blocked

   c) $e_\star \Downarrow \dot{v}_\star^{1..n}$ and **(Preservation)** yields $\bullet \vdash \dot{v}_\star \in T_\star^{1..n}$. Then let $\sigma = \{\mathsf{this} \mapsto D(\dot{w}_\star^{1..m})\ \mathsf{x}_\star \mapsto \dot{v}_\star^{1..n}\}$ and consider $b\,\Theta\,\sigma$. Either

      1. $b\,\Theta\,\sigma \not\Downarrow q$ for any q. This amounts to $b\,\Theta\,\sigma \in R$ and shows that (Dinvk) preserves R.
      2. $b\,\Theta\,\sigma \Downarrow q$, but this contradicts (Rinvk) blocked.

Case $a \equiv D[U_\star^{1..j}](e_\star^{1..n})$ and (Rnew), (Cnew) are blocked. By $\bullet \vdash a \in T$ and (Tnew), we also have $\mathsf{fields}(D[U_\star^{1..j}]) = T_\star^{1..n}$ and $\bullet \vdash e_\star \in T_\star^{1..n}$. Thus, either

i) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \not\Downarrow q$ for any q. Then $e_i \in R$ and (Dnew) preserves R

ii) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$, but this contradicts (Cnew) blocked.

iii) $e_\star \Downarrow \dot{v}_\star^{1..n}$, but this contradicts (Rnew) blocked.

Case $a \equiv e?\{\mathsf{x}\colon D[Y_\star^{1..j}] \Rightarrow b\}/\{d\}$ and (Rcst),(Rskp),(Ctst) are blocked. By $\bullet \vdash a \in T$ and (Ttst), we have all premises of the rule (Ttst). Thus either

i) $e \not\Downarrow q$ for any q, then $e \in R$ and (Dtst) preserves R.

ii) $e \Downarrow \mathbf{throw}$, but this contradicts (Ctst) blocked.

iii) $e \Downarrow C[R_\star^{1..k}](v_\star^{1..n})$. There are several subcases to consider:

a) $C[R_\star^{1..k}] \lhd^\star D[U_\star^{1..j}]$, and for $\Theta = \{\!\{Y_\star \mapsto U_\star^{1..j}\}\!\}$, $\sigma = \{\mathsf{x} \mapsto C[R_\star^{1..k}](\dot{v}_\star^{1..n})\}$, $b\Theta\sigma \not\Downarrow q$ for any q. Then $b\Theta\sigma \in R$ and (Dcst) preserves R.

b) $C[R_\star^{1..k}] \lhd^\star D[U_\star^{1..j}]$, and for $\Theta = \{\!\{Y_\star \mapsto U_\star^{1..j}\}\!\}$, $\sigma = \{\mathsf{x} \mapsto C[R_\star^{1..k}](\dot{v}_\star^{1..n})\}$, $b\sigma \not\Downarrow \mathbf{throw}$ but this contradicts (Rcst) blocked.

c) $C[R_\star^{1..k}] \not\lhd^\star D[U_\star^{1..j}]$, and $d \not\Downarrow q$ for any q. Then $d \in R$ and (Dskp) preserves R.

Case $a \equiv e_\star^{1..n}$ **match** $\{c_\star^{1..m}\}$ and (Rmch),(Cmch) are blocked.

By $\bullet \vdash a \in T$ and (Tmch), we have $\bullet \vdash e_\star \in S_\star^{1..n}$, for all j a case typing $\bullet; S_\star^{1..n} \vdash c_j \in T_j$, and for each body b_j a typing $\Gamma'_j \vdash b_j \in T_j$.

Thus, either

i) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \not\Downarrow q$ for any q. Then $e_i \in R$ and (Dmch) preserves R.

ii) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$ or $e_i \Downarrow \mathbf{null}$, but this contradicts (Cmch) blocked.

iii) $e_\star \Downarrow \dot{v}_\star^{1..n}$. Then we distinguish these cases:

a) if all cases reject, this contradicts that the last case always accepts.

b) There exists an i such that $\forall j < i \;.\; \dot{v}_\star^{1..n}; c_j \Downarrow \mathbf{reject}$, $c_i = \mathbf{case}\ p_\star^{1..n} \Rightarrow b$ and $\dot{v}_\star \curvearrowright p_\star \dashv \Theta_\star; \sigma_\star^{1..n}$. Then, either

1) $b\,\Theta_\star^{1..n}\,\sigma_\star^{1..n} \not\Downarrow q$ for any q, then $b\,\Theta_\star^{1..n}\,\sigma_\star^{1..n} \in R$ and (Dbdy), (Dcase) preserve R

2) $b\,\Theta_\star^{1..n}\,\sigma_\star^{1..n} \Downarrow q$, which contradicts (Rmch) blocked.

□

Thm 7 (Type Soundness)
If $\bullet \vdash a \in C$ then either $a \Uparrow$ or $a \Downarrow q$ for some q with $\bullet \vdash q \in C'$, $C' <: C$.

Proof Consequence of **(Progress)** and **(Termination)**.

6.8 Example

We now show how the GADT of expressions in HASKELL can be rendered in GPAT, which illustrates just how the extractor typing rule achieves a form of existential quantification. Consider the following definitions:

```
class Term[X]

class Num(value:Int) ◁ Term[Int] {}
class PairTerm[Y,Z](a:Term[Y], b:Term[Z]) ◁ Term[Tuple2[Y,Z]] {}

class Match() {

 @safe def pairTerm[Y,Z](p:PairTerm[Y,Z]): Tuple2[Term[Y],Term[Z]] =
    Tuple2[Term[Y],Term[Z]](p.a, p.b)

 @safe def num(p:Num): Tuple1[Int] =
    Tuple1(n.value)
}

d match {
  case Match().pairTerm[Y,Z](fst, Match().num(i)) => e
  case x                                          => null
}
```

Figure 6.19 contains an example for a typing derivation that uses the system presented above to derive a typing for this match expression. It demonstrates how pattern matching expressions can provide an anchor for type information to be extracted from pattern shapes and types.

6.9 Summary and Discussion

The development presented in this chapter points out that pattern matching can enable useful interactions with the type system. However, the calculus is far from being a usable system and leaves out many aspects of the SCALA programming language, which has a richer meta-theory of subtyping.

Class definitions in SCALA can be marked with *definition-site variance annotations*. There, subtyping relationships like `List[`S`] <: List[`T`]` can be derived from subtype relationship $S <: T$ and the *covariance* of the type `List` in its type parameter, which is indicated through an annotation +. Dually, *contravariance* is indicated by annotating a type parameter with –. In contrast, the generic type system presented here is *invariant*. As mentioned earlier, the meta-theory of subtyping presented here has originally been developed with definition-site variance in mind [26].

$$
\begin{array}{lll}
p_1 & = \mathsf{Match}().\mathsf{pairTerm}[Y,Z](\mathsf{fst}, p_2) & \\
p_2 & = \mathsf{Match}().\mathsf{num}(\mathsf{i}) & \\
\mathcal{D}_0 & = \Gamma \vdash d \in \mathsf{Term}[A] & \\
\mathcal{D}_1 & = \dfrac{\cdots \quad \mathsf{xtype}(\Gamma, \Gamma 1, \mathsf{Match}, \mathsf{num}) = (\mathsf{Num})\mathsf{Tuple}_1[\mathsf{Int}]}{\Gamma, \Gamma_1; \mathsf{Term}[Z] \ni p_2 \dashv \Gamma_2, \mathsf{i}\colon \mathsf{Int}}\ (\mathsf{TPext}) & \\
\mathcal{D}_2 & = \dfrac{\Gamma; \mathsf{Term}[A] \ni \mathsf{x} \dashv \mathsf{x}\colon \mathsf{Term}[A] \qquad \Gamma, \mathsf{x}\colon \mathsf{Term}[A] \vdash \mathbf{null} \in \mathsf{Exc}}{\Gamma; \mathsf{Term}[A] \vdash \mathbf{case}\ \mathsf{x} \Rightarrow \mathbf{null} \in \mathsf{Exc}}\ (\mathsf{Tcase}) &
\end{array}
$$

$$
\dfrac{\cdots \quad \mathcal{D}_1 \quad \mathsf{xtype}(\Gamma, \mathsf{Match}(), \mathsf{pairTerm}[Y,Z]) = [Y,Z](\mathsf{PairTerm}[Y,Z])\mathsf{Tuple}_2[\mathsf{Term}[Y], \mathsf{Term}[Z]]}{\dfrac{\dfrac{\Gamma; \mathsf{Term}[A] \ni p_1 \dashv \Gamma, \Gamma_1, \Gamma_2, \Gamma'}{\Gamma; \mathsf{Term}[A] \vdash \mathbf{case}\ p_1 \Rightarrow e \in T}\ (\mathsf{TPext}) \qquad \Gamma, \Gamma_1, \Gamma_2, \Gamma' \vdash e \in T}{\mathcal{D}_0 \qquad \Gamma \vdash d\ \mathsf{match}\{\mathbf{case}\ p_1 \Rightarrow e\ \mathbf{case}\ \mathsf{z} \Rightarrow \mathbf{null}\} \in T}\ (\mathsf{Tcase}) \qquad \mathcal{D}_2}\ (\mathsf{Tmch})
$$

$$
\begin{array}{lll}
\Gamma' & = \{\mathsf{fst}\colon \mathsf{Term}[Y], \mathsf{i}\colon \mathsf{Int}\} & \Delta_0 = \Delta_1, \Delta_2 \\
\Gamma_1 & = X_1, Y, Z, \Delta_1 & \Delta_1 = \{X_1 <: \mathsf{Term}[A], X_1 <: \mathsf{PairTerm}[Y,Z]\} \\
\Gamma_2 & = X_2, \Delta_2, \mathsf{i}\colon \mathsf{Int} & \Delta_2 = \{X_2 <: \mathsf{Term}[Z], X_2 <: \mathsf{Num}\}
\end{array}
$$

Figure 6.19: Example of a Typing Derivation

However, SCALA also adds type members, multiple inheritance and a fundamental notion of nested types. Again, since pattern matching introduces new variables and offers flexible syntax, interesting and useful experimental type-system interactions have been implemented.

A particular point that deserves mentioning is that adding existential quantification the type system opens up possibilities that go beyong the system we described here. Existentially quantified types allow for instance to give precise types to extractor methods without having to add a type-test, like so:

```
@safe def pairTerm(t:Term[A] forSome {type A}):
   Tuple2[Term[Y],Term[Z]] forSome {type Y; type Z} =...
}
```

They thus add to the modularity of extractors for generic pattern matching, since the extractor can be written, compiled, altered independently from the matched type `PairTerm[`Y`,`Z`]`. A formalization of an object-oriented calculus with pattern matching, subtyping constraints and existential quantification is planned for future work.

Chapter 7

Related Work

This chapter enumerates literature relevant to translation of pattern matching and interactions of pattern matching constructs and type systems.

7.1 Case Classes, Extractors and Views

Case classes were first proposed by Odersky and Wadler [70], where they denote closed, non-extensible classes that were introduced with the purpose of providing algebraic data types in an object-oriented context. A slightly more extensible version (in the sense that it encourages adding new variants to existing types) is the basis for Zenger's diploma thesis on an extensible compiler framework [96]. The ideas were subsequently refined, leading to extensible algebraic data types with defaults [97]. Case classes as presented in this thesis subsume the developments with regards to language design. In contrast, the compilation techniques of the mentioned literature are different. While the original PIZZA implementation used an algorithm described by Wadler [93] (discussed in Section 7.2 below) which did not apply to case classes inheriting from case classes, Zenger used an incremental algorithm which would in principle work for this general form of case classes but which did not support incompleteness checks.

Despite the mainstream object-oriented setting being based on nominal subtyping and thus significantly different from algebraic data types, no formalization seems to have been carried out with the goal of describing the optimizing translations. This line of work has subsequently led to the concept of case classes in SCALA. In this thesis, we contribute to it by presenting case classes as a derived concept, while considering extractors as the fundamental building block to object-oriented pattern matching.

Views in functional programming languages [92, 71] are conversions from one data type to another that are implicitly applied in pattern matching. They play a role similar to extractors

in Scala, in that they permit to abstract from the concrete data-type of the matched objects. However, unlike extractors, views are anonymous and are tied to a particular target data type.

Tullsen [89] proposes to use functions returning optional values as first class patterns, which is an important predecessor of extractor-based pattern matching as described here. Since functions are first-class values and the option type returned by such a pattern is associated with a monad, it is straightforward to define pattern combinators, making it possible to define new patterns from old ones. Furthermore, he arrives at a more flexible form of HASKELL lazy pattern matching by giving different pattern combinators which vary in the evaluation order they enforce. He proposes syntactic sugar called pattern binders to handle variable binding conveniently. The approach is tailored to algebraic data types and moreover assumes, in the interest of minimality, that all pattern matches are complete. Focusing on expressivity with a minimal set of primitives, Tullsen does not deal with the issue of optimized translation - instead the verbose nature of hand-coded match expressions involving pattern binders and functions returning options suggests that programmers perform the optimization by hand as well.

Erwig's active patterns [30] provide views for non-linear patterns with more refined computation rules. Palao Gostanza, Pena and Nunez's active destructors [43] are closest to extractors; an active destructor corresponds almost exactly to an `unapply` method in an extractor. However, they do not provide data type injection, which is handled by the corresponding `apply` method in our design. Also, being tied to traditional algebraic data types, active destructors cannot express inheritance with varying type parameters in the way it is found in GADT's. Regarding compilation, they describe a scheme that inlines active destructors before performing further optimizations. This is an interesting idea, however it means giving up separate compilation. It can be very helpful in whole-program compilers and optimizers, or when users can live with the fact that changing an extractor requires them to recompile all sources that make use of the extractor.

Erwig and Peyton Jones propose an extension of HASKELL pattern matching with pattern guards and transformational patterns [31]. Their transformational patterns can play the role of views, with the benefit that they do not interfere with equational reasoning.

Kirchner, Moreau and Reilles introduce user-defined operators (anchors) in TOM, which are very briefly described in their papers [52, 11]. This concept similar to extractors in that it allows the user to define what it means to match an algebraic signature. Their pattern matching compiler works for JAVA, C and C++. Unfortunately, the TOM framework has not been described comprehensively using a complete formal semantics for a given language, which is probably due to their targetting various host languages.

Syme, Neverov and Margetson [87] independently introduced active patterns which are a concept very similar to extractors. Their active patterns can be defined to partially or com-

pletely discriminate amongst values of a given type, which happens by structured names. For complete discriminators, incompleteness checking is available. They give an operational semantics in the form of an F# interpreter for pattern matching expressions. Their development is very close to ours, however they allow heuristics (these are described below) which allow traversals of input values that differ from the standard left-to-right one. The paper lists many applications and a promising future direction to monadic and transactional pattern matching: both are based on the chaining of extractor calls.

In their paper on abstract value constructors [4], Aitken and Reppy present symbolic constants and symbolic patterns ("templates"). However, a template can only be defined in terms of another pattern. In particular, one cannot perform arbitrary computation, which only slightly extends the limitations imposed by fixed algebraic data types. Fähndrich and Boyland [32] expand on this idea, by allowing a template to choose from several patterns and to be recursive, yielding a way to recognize full regular tree languages. This form of "computation", while still not comprising the source language, allows the authors to describe semi-structured data by means of pattern types. The authors devise a full type system that can check the language recognized by patterns.

Garrigue's polymorphic variants [38] for OCaml provide another way to reconcile algebraic data types and extensibility. Here, the programmer is allowed to form arbitrary subtype relationships between variant types. Since a variant type is merely a tag and a normal data type, type checking these variants amounts to checking relationships between sets of tags. Every instance of a variant is then at the same time instance of many other variant types, which is case of subtype polymorphism. The implementation uses an integer tag that is computed from the tag names (with the drawback that collisions might in principle make a perfectly valid program fail to compile). A problem with polymorphic variants is that they do not blend seamlessly with the rest of the type system. Since there are too many subtyping relationships, explicit type annotations are required to express that a value is of a particular variant type. This is immediately clear from the following example definitions:

```
atype = `A of int
btype = `B of int
aorb = [atype|btype]
```

Now, a value `'A 3` is an instance of `atype`, but it is not an instance of `aorb`. It can, however, be turned into one using an explicit typecast. These explicit typecasts are cumbersome and limit the applicability of polymorphic variants to extensible algebraic data types.

7.2 Correctness

Wadler [93] argues for correctness of pattern matching by making extensive use of the exception monad (expressed as a "fatbar" operator that combines decision trees and a FAIL

constant). We describe this reference in more detail below.

The correctness proof in this thesis follows the general structure of a similar proof from Qin Ma's thesis [59]. In the first part of her work, the author proves that a translation from patterns in the join-calculus into ML pattern matching is correct.

7.3 Optimizing Pattern Matching

Augustsson [8, as cited by [93]] seems to be one of the earliest description of pattern match compilation. Wadler [93] presents the same algorithm in detail. He uses a function **match** that translates match expressions to lambda-expressions. This function takes three arguments, a list of variables, a list of case clauses, and a default expression that is used for match failure. Four rewrite rules are then given to handle translation, namely the empty rule, the variable rule, the constructor rule and the mixture rule. The algorithm is not optimal in the sense that it generates codes that in some cases performs a test twice.

Ophel [72] describes an improvement. He elides the third argument to the **match** function and rewrites a match expression into a case distinction and "smaller" match expressions. The "equations", as the case clause matrix is called, is analyzing complete sets of constructors. Informally, he describes that the incompleteness of a match expression is signaled when the FAIL constant is evaluated, and redundancy is signaled by checking each body for reachability. The generated code performs the tests on the input from left to right.

Cardelli [16] also describes briefly how algebraic pattern matching can be translated to decision trees in an optimizing manner. He also mentions heuristics other than left-to-right.

Pettersson [73] describes a variant of the above algorithm for match expression translation to decision-trees. Its behavior resembles deterministic, finite automata in the sense that multiple translations of the same expression are avoided by *hash-consing* (the memoization of expressions that have been translated before). This leads to a notion of state, like in automata. The generated code can be more compact by using jumps instead of regenerating equivalent decision-trees. Another advantage of the matrix-based algorithm is that incompleteness and redundancy checking become cheap operations.

Scott and Ramsay [79] studied the effects of using different orderings in the algorithm described by Pettersson. It is of course not possible to freely choose a completely arbitrary ordering of pattern positions, since no part of a sub-pattern can be tested before the input value has been successfully tested against the parent type. The choice of ordering, called heuristics, thus focus on ordering the sub-patterns on one level. Since the case clause matrix offers to see, at each translation step, the entire set of sub-patterns, a lot of information is available that can be used to devise heuristics: for instance, a possible strategy might be to

first test the column that leads to the earliest distinction. Unfortunately, their study does not seem to take the merging of equivalent states into account.

With similar aims, Sekar, Ramesha and Ramakrishnan coin the term adaptive pattern matching for the dynamic selection of the traversal order based on the entire match expressions [80]. They use a formalism similar to tree automata and compare several strategies for synthesizing a traversal order from a set of patterns. After showing through an intricate example that all of these sometimes decrease the code size or execution time, they introduce a strategy based on *index* positions which is never worse than any of the strategies in terms of space or time, while sometimes being better. They give an algorithm that works on untyped systems, which computes *representative sets* for untyped systems in quadratic time and show that in typed systems, the computation of these sets is NP complete.

Sestoft [81] takes a rather different approach to translation of pattern matching: he gives an ML function that *interprets* ML pattern match expressions and uses partial evaluation to derive from it a function performing the behavior of decision trees directly. His translation makes explicit the use of the set of variants which is available for closed set of types (which he calls the *span* of a type): if a type has n branches, and $n - 1$ branches have been tested already, the last test can be omitted.

Maranget [60] deals with functional pattern matching warnings due to incompleteness or redundancy, including a closer look at the special case of lazy functional languages. The definition of pattern matching has to be adapted to laziness, since values may possess infinite structure and evaluation triggered by pattern matching may not terminate. He introduces a formalism based on case clause matrices similar to our development here. He demonstrates also that when incompleteness checking can be done without unfolding alternative branches, thereby avoiding exponential blowup.

In contrast to the algorithms discussed above, Field and Harrison [34] describe a notion of best-fit pattern matching where it is the most specific pattern rather than the first case clause from the top that will be chosen in a match expression. This yields to a different intermediate representation. Otherwise, translation is similarly avoiding redundant tests in algebraic data types.

Zenger [96] introduces algebraic data types in the form of case classes to an extensible compiler framework. He describes an incremental algorithm that updates a mutable internal representation for each case clause and each sub-pattern. The internal representation is very compact and the translation of match expressions very efficient. In the course of this thesis, we tried to extend this algorithm with other kinds of patterns. However, incompleteness checking proved difficult since reachability analysis on the intermediate representation amounts to interpreting generated code.

Moreau, Ringeissen and Vittek [66] translate pattern matching code into existing languages, without requiring extensions. Their translation is a source-to-source translation from match

expression to decision trees. Algebraic data types are declared through a macro-language, and match expressions embedded in sources are compiled to expressions in the host language. Unfortunately, they do not give details of their translation algorithm.

7.4 Generic Pattern Matching

7.4.1 Dynamic Typing

Abadi *et al* [2] study dynamic typing in a statically typed language, more precisely the polymorphic lambda calculus. Their paper introduces a typecase construct and a type `Dynamic`, with the idea its instances are value bundled type information. Only those values for which dynamic typing is needed have the type `Dynamic`, and the typecase construct serves to recover the type information when it is needed.

7.4.2 Subtyping Constraints

Trifonov and Smith [88] describe subtyping in the presence of constraints. They study a semantic characterization of subtyping using regular trees, in the aim of generalizing Hindley-Milner type inference to polymorphic type schemes with subtype constraints.

Pottier [74] presents a theory for simplifying subtyping constraints, based on unification. Like Trifonov and Smith, he carries over the syntactic problem of type and type constraint simplification to the domain of tree automata. He presents a minimization algorithm that reduces a constraint problem, often yielding optimal equivalent problems.

Litvinov's thesis [57] deals with subtyping constraints in the context of the multiple dispatch, as provided by the CECIL programming language. He proposes constraint-bounded polymorphism, which generalizes F-bounds to arbitrary constraints, similar to our GPAT type system described in Chapter 6. His presentation is tailored to multi-methods, which makes it difficult to compare it with our system where constraint are derived from deep patterns in pattern matching expressions. The motivation for his system are type-checking idioms like partial subtyping that are encountered when bootstrapping and type-checking the CECIL compiler (some of which are quite specific to CECIL).

7.4.3 Generalized Algebraic Data Types

Generalized algebraic datatypes (GADTs) made their first appearance as guarded recursive data type constructors, proposed in an ML context by Xi, Chen and Chen [95]. They are also

mentioned as first-class phantom types by Cheney [19]. Crary, Weirich and Morrissett introduced type descriptors (type reps) as a particular early example of GADT's when studying intensional type analysis [23].

Peyton Jones, Vytiniotis, Weirich and Washburn [48] show how type annotations can be used to make the type inference task easy in the functional setting. They use "wobbly types", which express in a declarative way the uncertainty caused by the incremental nature of typical type-inference algorithms. Vytiniotis, Weirich and Peyton Jones also perform a generalization of the technique to "boxy types" in order to solve the problem for higher-rank types [91], yielding a conservative extension of Hindley-Milner type inference.

Kennedy and Russo [50] describe ways to harness GADTs in object-oriented programming. They show that while definitions that model GADTs come almost for free using generic classes, programs that operate on GADTs need to resort to casts. They also show that equational type constraints in definitions are remedy to this problem and allow writing visitors that properly operate on GADTs. They furthermore propose a matching construct and argue that this makes programs operating on GADT more readable and also permits to match tuples of expressions. Their matching construct corresponds to the typecase instruction described in Chapter 2.

7.5 Other Aspects to Pattern Matching

Queinnec describes applications in and for LISP [75]. His text shows the origins of pattern matching in LISP meta-programming constructs, and some applications in natural language processing. Steele and Gabriel [83] is another text that contains remarks on the history of pattern matching/template filling as an important building block to meta-programming.

The π-calculus provides foundations for concurrent and distributed computation and interaction. The "applied" extends pi-calculus with algebraic pattern matching [35], specifying matching as an equational theory. Algebraic pattern matching is subsumed by matching on case classes as described in this thesis.

Ma [59] compiles patterns in the (nondeterministic) join calculus into ML style algebraic patterns, using the partial order "matches more instances" that is defined on patterns.

Harrison, Sheard and Hook define precisely how functional pattern matching in Haskell alternates between lazy and strict evaluation in order to handle nested patterns [44].

While matching is the computation of a substitution that allows a pattern to be syntactically equal to value, syntactic unification is the generalization that seeks to make two patterns syntactically equal. Syntactic unification is the basis of logic programming. Baader [9] describes the concept and provides pointers to the vast literature on unification algorithms.

In cotrast to pattern matching in a statically typed context, Erlang [7] is a functional programming language that use pattern matching but is dynamically typed. Term pattern matching is defined in terms of ground terms and does not differ fundamentally from algebraic data types. Pattern matching in Erlang is thus subject to the same restrictions as algebraic data types and subsumed under the case class mechanism proposed here.

Applications in Object-Oriented Programming. Pattern matching in the context of object-oriented programming has been applied to message exchange in distributed systems by Lee, LaMarca and Chambers [55]. They argue that through pattern matching on semistructured data, programs can be made more robust against changes to data formats. Gapeyev also applies pattern matching to semistructured data [37], following similar approach by Hosoya and Pierce put forward in the functional paradigm [45].

Chin and Millstein [20] use pattern matching in the form of a special language construct called a responder. Responders support event-handling in user interfaces. The paper contains many code examples that show how events and responders (which correspond to case classes and pattern matching) can facilitate programming interactive applications.

Sequent Calculus Although not mentioning pattern matching explicitly, Abramsky's computational interpretation of linear logic [3] contains pattern matching primitives. He motivates a calculus based on sequent calculus rather than natural deduction and derives from it a parallel, lazy language that can be linearly typed. Matching is necessary because in the subderivation of sequent calculus, a variable is replaced with a whole term substituted for it. The connection to sequent calculus is also the theme of Kesner, Puel and Breazu-Tannen pattern calculus [51].

Multiple Dispatch Multi-methods [17, 18, 63, 21] are an alternative technique which unifies pattern matching with method dispatch. Multi-methods are particularly suitable for matching on several arguments at the same time. An extension of multi-methods to predicate-dispatch [29, 62] can also access embedded fields of arguments; however it cannot bind such fields to variables, so support for deep patterns is limited.

A design that is close to extractors is Richard and Lhoták describe how multi-methods can be used for pattern matching akin to algebraic data type deconstruction [77].

Reversible Computation Liu and Myers [58] propose to interpret matching as a backwards mode of computation. No translation algorithm is given, instead the authors focus on language design.

Historical References Rod Burstall's paper [13] seems the earliest description of pattern matching as a separate programming language construct. It seems to first have been implemented in an extension of LISP by Fred McBride [61].

Chapter 8

Conclusion

In this chapter, we summarize the results of our study of object-oriented pattern matching.

8.1 First Order Pattern Matching

We defined the object-oriented pattern matching problem and compared several standard solutions that are used in object-oriented programs today. We then defined pattern matching construct compatible with object-oriented programming, based on type tests and two underlying mechanisms, case classes and extractor methods. We showed that while case classes follow functional programming in simplicity and elegance, they also lack extensibility and complete representation independence. Extractors give us full representation independence but are associated with a small performance penalty.

The formal translation we presented is an advance over previous presentations of translation algorithms, since it has a modular correctness proof *and* provides the basis for a reasonable, modular implementation. This means that if new kinds of patterns are to be added to the language, they can easily be integrated in the algorithm, its correctness proof as well as a rewriting-based implementation.

Our formal development brought to light that optimizations for this object-oriented form of pattern matching can have important effects on termination. It seems thus desirable to allow only restricted forms of computation to take place in extractor methods, and additionally provide compilers with static analysis facilities that can ensure termination and exception-freeness on this restricted fragment. Pragmatically, it is useful to know that the only condition for optimized translation to preserve semantics is termination and exception-freeness, since it means that programmers that are certain that their extractors are safe may choose to perform unrestricted computation. In other words, our definition of safety is not

exotic or specific to matching, and programmers frequently use code that is unsafe in this sense.

We discussed some implementation choices, which fall into two categories: more kinds of patterns in order to enhance usability and programmer convenience, and changes to the underlying mechanisms in order to enhance runtime performance. An interesting choice that defies this classification is the integrated type test in extractor definitions, as it increases programmer convenience and at the same time enables further optimization (albeit at the price of increasing the coupling between the matched type and the extractor behavior).

The performance evaluation showed that object-oriented pattern matching has roughly the same performance characteristics as other techniques. A vast improvement in readability is not paid with a horrible performance penalty, on the contrary it even leads to efficient programs in some cases where encodings introduce inefficiencies.

8.2 Generic Pattern Matching

Parametric polymorphism has a deep impact on the way in which object-oriented programs are written and type-checked. Our study of pattern matching revealed new aspects in which the enhanced expressivity of type systems can be harnessed. This was predictable, since matching is on the one hand based on type tests and, on the other hand, introduces new identifiers to a scope (the case clause body), similar to the "let" construct in functional languages, which provides for ample possibilities of interacting with the type-system in meaningful ways.

We studied how advanced techniques from functional programming languages, namely generalized algebraic data types, can be carried over to object-oriented style. Our development rested on a rich notion of subtyping that involved subtyping constraints. Subtyping in the presence of generalized constraints (as opposed to only upper bounds) seems to be a good match to object-oriented pattern matching, particularly because expressive object-oriented languages already have to deal with some form of subtyping constraints for the upper bounds on type parameters. We introduced subtyping in presence of constraints and gave declarative and syntax-directed versions of the subtyping rules needed in order to handle introduction of type parameters in case clauses. We furthermore described how type-checking collects the various constraints and lets us check the body of each case clause in a type environment contains information from the pattern structure.

8.3 Future Work

Besides further case studies to assess whether and to which extent object-oriented pattern matching can impact software engineering practice, future work is indicated in three directions.

First, a formal model of local type inference that takes into account existentially quantified type expressions and subtype constraints. The formalization presented in this document did not deal with existential types, although these can add to the extensibility of programs that make use of extractor-based pattern matching.

Second, we experimented with a notion of matching on so-called outer types in SCALA programs. It may turn out that pattern matching is the only operation that can cleanly query dynamically the level from which an object is instantiated – since in SCALA, types can be arbitrarily nested, this would open new possibilities of structuring programs while maintaining the a safe dynamic type test construct.

Third, various enhancements to the usability have been effected and suggested for the SCALA implementation of pattern matching, among these parameterized extractors, and-patterns (conjunctions) and support for more precise exhaustivity checking that includes information from guards. It seems useful to find a general mechanism that would permit the user to implement these enhancements himself by expressing them in terms of smaller pattern primitives. Chaining extractor calls (combined with evaluation at compile time) may provide a useful angle at this problem.

An orthogonal aspect that requires more work is the optimization of extractor-based pattern matching. Each of the above-mentioned directions and the original mechanism described in this thesis needs to be efficiently implementable, in order to preserve the balance of expressiveness and efficient compilation that is characteristic to pattern matching in functional programming languages.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, 1991.

[3] Samson Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1992.

[4] William E. Aitken and John H. Reppy. Abstract value constructors. Technical Report TR 92-1290, Cornell University, Ithaca, NY 14853, 1992.

[5] Philippe Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings*. PhD thesis, EPF Lausanne, 2006.

[6] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd Ed.* Cambridge University Press, 2002.

[7] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology Stockholm, 2003.

[8] Lennart Augustsson. Compiling pattern matching. In *Jean-Pierre Jouannaud, ed., Functional programming languages and computer architecture, LNCS 201*, pages 368 – 381, 1985.

[9] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Springer Verlag, 1999.

[10] Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaetà. Basic theory of F-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

[11] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal islands. In *In Proceedings of 11th International Conference on Algebraic Methodology and Software Technology - AMAST '06*, 2006.

[12] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 1998.

[13] Rod M. Burstall. Proving properties of programs by inductive definitions. *Computer*, pages 41–48, 1969.

[14] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. Hope: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, aug 1980.

[15] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 273 – 280, 1989.

[16] Luca Cardelli. Compiling a functional language. In *Proc. ACM Symposium on Lisp and Functional Programming*, 1984.

[17] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Lisp and Functional Programming*, pages 182–192, June 1992.

[18] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Springer LNCS*, pages 33–56, 1992.

[19] James Cheney and Ralf Hinze. First class phantom types. Technical Report 1901, Cornell University, 2003.

[20] Brian Chin and Todd Millstein. Responders: Language Support for Interactive Applications. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[21] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design Rationale, Compiler Implementation, and Applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, May 2006.

[22] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[23] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, nov 2002.

[24] D. A. Spuler. Compiler code generation for multiway branch statements as a static search problem. Tech. Rep. 94/3, James Cook University, Department of Computer Science, 1994.

[25] Mirco Dotta and Philippe Suter. Verifying pattern matching with guards., 2007. Class Project Report for course "Software Analysis and Verification" (V. Kuncak). available at `http://lara.epfl.ch/dokuwiki/doku.php?id=verifying_pattern_matching_with_guards`.

[26] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and Generalized Constraints for C# Generics. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[27] Burak Emir, Sebastian Maneth, and Martin Odersky. Scalable Programming for XML Services. In *J.Kohlas, B.Meyer, A.Schiper (Eds.): Dependable Systems, LNCS 2048, pp. 103–126*. Springer Verlag, 2006.

[28] Burak Emir, John Williams, and Martin Odersky. Matching objects with patterns. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[29] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: unified theory of dispatch. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Springer LNCS*, pages 186–211, 1998.

[30] Martin Erwig. Active patterns. In *8th Int. Workshop on Implementation of Functional Languages*, volume 1268 of LNCS, pages 21–40, 1996.

[31] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. haskell workshop, 2000. available at `http://research.microsoft.com/~simonpj/Papers/pat.htm`.

[32] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[33] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proc. of International Conference on Functional Programming(ICFP)*, pages 26–37, 2001.

[34] A.J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[35] Cédric Fournet and Martín Abadi. Mobile values, new names, and secure communication. In *Proc. of Principles of Programming Languages (POPL)*, 2001.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[37] Vladimir Gapeyev and Benjamin C. Pierce. Regular Object Types. In *Proceedings of European Conference on Object-Oriented Programming*, 2003.

[38] Jaques Garrigue. Programming with polymorphic variants. In *ML Workshop. Baltimore*, 1998.

[39] Jeremy Gibbons, Meng Wang, and Bruno C. d. S. Oliveira. Generic and indexed programming. In Marco Morazan, editor, *Trends in Functional Programming*, 2007.

[40] Jürgen Giesl and Thomas Arts. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[41] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

[42] Andrew D. Gordon. A Tutorial on Co-induction and Functional Programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Springer Workshops in Computing, pages 78 – 95, 1995.

[43] Pedro Palao Gostanza, Ricardo Pena, and Manuel Manuel Nunez. A new look at pattern matching in abstract data types. In *Proceedings of International Conference on Functional Programming (ICFP)*, 1996.

[44] William Harrison, Tim Sheard, and James Hook. Fine control of Demand in Haskell. In *Sixth International Conference on Mathematics of Program Construction*, Dagstuhl, Germany, 2002.

[45] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *Jounral of Functional Programming*, 13(6):961–1004, 2003.

[46] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 1999.

[47] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[48] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple Unification-based Type Inference for GADTs. In *Proc. of International Conference on Functional Programming*, 2006.

[49] Andrew Kennedy and Benjamin C. Pierce. On Decidability of Nominal Subtyping with Variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, 2007.

[50] Andrew Kennedy and Claudio Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In *Proc. of Object-Oriented Programming Systems and Languages (OOPSLA)*, 2005.

[51] Delia Kesner, Laurence Puel, and Val Breazu-Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.

[52] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal verification of pattern matching code. In *Proc. of the International Conference on Principles and Practice of Declarative Programming*, 2005.

[53] Anton Korobeynikov. Improving Switch Lowering for the LLVM compiler system. In *Proc. of the 2007 Spring Young Researchers Colloquium on Software Engineering (SYRCoSE'2007), Moscow, Russia*, 2007.

[54] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proc. of European Conference on Object-Oriented Programming (ECOOP), Springer LNCS 1445*, 1998.

[55] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. Hydroj: Object-oriented Pattern Matching for Evolvable Distributed Systems. In *Proc. of Object-Oriented Programming Systems and Languages (OOPSLA)*, 2003.

[56] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Theoretical Computer Science*, (submitted).

[57] Vasily Litvinov. *Constraint-Bounded Polymorphism: and Expressive and Practical Type System for Object-Oriented Languages*. PhD thesis, University of Washington, 2003.

[58] Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *Proceedings of 5th Internnational Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 110–127, 2003.

[59] Qin Ma. *Concurrent Classes and Pattern Matching in the Join Calculus*. PhD thesis, INRIA-Rocquencourt and University Paris 7, 2005.

[60] Luc Maranget. Warnings in pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

[61] Fred McBride. *Computer Aided Maniuplation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.

[62] Todd Millstein. Practical Predicate Dispatch. In *Proc. of Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 245–364, 2004.

[63] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):836–889, September 2004.

[64] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[65] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470 – 502, 1988.

[66] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In *G. Hedin, ed., 12th Conference on Compiler Construction, volume 2622 of LNCS*, pages 61–76, 2003.

[67] Martin Odersky. In defense of pattern matching, 2006. http://www.artima.com/weblogs/viewpost.jsp?thread=166742.

[68] Martin Odersky. The Scala Experience. Invited Talk at ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2007.

[69] Martin Odersky. Scala Language Reference, 2007. available at http://www.scala-lang.org.

[70] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. of Principles of Programming Languages (POPL)*, 1997.

[71] Chris Okasaki. Views for Standard ML. In *Proceedings of SIGPLAN Workshop on ML*, pages 14–23, 1998.

[72] John Ophel. An Improved Mixture Rule For Pattern Matching. *ACM SIGPLAN Notices*, 24(6), 1989.

[73] Mikael Pettersson. A term pattern-match compiler inspired by finite-automata theory. In *Proc. of International Workshop on Compiler Construction (CC).*, volume Volume 641 of LNCS, 1992.

[74] Francois Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, Nov 2001.

[75] Christian Queinnec. *Le Filtrage - une application de (et pour) Lisp*. InterEditions, Paris, 1990.

[76] John Reynolds. Towards a theory of type structure. In *In Programming Symposium, Proceedings Colloque sur la Programmation, LNCS 19*. Springer Verlag, 1974.

[77] Adam Richard and Ondřej Lhoták. OOMatch: Pattern Matching as Dispatch in Java. Technical Report CS-2007-05, University of Waterloo, 2007.

[78] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, EPF Lausanne, 2005.

[79] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, University of Virginia, 2000.

[80] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal of Computing*, 24(6):1207–1234, 1995.

[81] Peter Sestoft. ML Pattern Match Compilation and Partial Evaluation. In *Danvy, Glück, and Thiemann (eds.): Dagstuhl Seminar on Partial Evaluation. LNCS 1110*. Springer Verlag, 1996.

[82] Vincent Simonet and Francois Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1), 2007.

[83] Guy L. Steele, Jr. and Richard P. Gabriel. The Evolution of Lisp. `http://www.dreamsongs.com/Files/HOPL2-Uncut.pdf`.

[84] Don Syme. Coinductive semantics. Personal communication.

[85] Don Syme. F♯ Manual. `http://research.microsoft.com/fsharp/manual/default.aspx`.

[86] Don Syme and Andrew Kennedy. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, 2001.

[87] Don Syme, Greg Neverov, and James Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2007.

[88] Valery Trifonov and Scott F. Smith. Subtyping constraint types. In *Static Analysis Symposium (SAS), LNCS 1145*, pages 349–365. Springer Verlag, Sept 1996.

[89] Mark Tullsen. First class patterns. In *In 2nd Int. Workshop on Practical Aspects of Declarative Languages, LNCS 1753*. Springer Verlag, 2000.

[90] David Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Languages and Computer Architecture (FPCA), LNCS 201*. Springer, 1985.

[91] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *Proc. of International Conference on Functional Programming*, 2006.

[92] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. of Principles of Programming Languages (POPL)*, 1987.

[93] Philip Wadler. Pattern Matching, Chapter 4 of Peyton Jones, Wadler "Implementation of Functional Programming Languages", Prentice Hall., 1987.

[94] Wolfgang Wechler. *Universal Algebra*. Number 25 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.

[95] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proc. of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

[96] Matthias Zenger. Erweiterbare Übersetzer. Master's thesis, Universität Karlsruhe, 1998.

[97] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2001.

[98] Matthias Zenger and Martin Odersky. Scalable component abstraction. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2005.

Index

Curriculum Vitae

Personal Data

Name	Burak Emir
Gender	male
Born	11.11.1977 in Ankum, Germany

Education

1990	1996	Artland Gymnasium Quakenbrück
1997	2003	Studies of Computer Science at RWTH Aachen
2003	2007	Doctoral Programme at EPF Lausanne

Professional Experience, Internships, Exchanges

1996	1997	Civil Service, Christl. Krankenhaus Quakenbrück, Germany
1999	2000	Exchange Year at EPF Lausanne, Switzerland
	2000	Internship CERN, Geneva, Switzerland
	2005	Internship Microsoft Research, Cambridge, UK