Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Moritz Uhlig

A Type Class Based Approach for Modeling Transformations of Abstract Petri Nets

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Moritz Uhlig

A Type Class Based Approach for Modeling Transformations of Abstract Petri Nets

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Julia Padberg
Zweitgutachter: Prof. Dr. Friedrich Esser

Eingereicht am: 28. August 2012

Moritz Uhlig

Thema der Arbeit
A Type Class Based Approach for Modeling Transformations of Abstract Petri Nets

Stichworte
Petri-Netze, Abstrakte Petri-Netze, Funktionale Programmierung, Scala, Netztransformationen, Kategorientheorie, Typklassen, Netzklassen

Kurzzusammenfassung
Abstrakte Petri-Netze bieten einen einheitlichen Ansatz zur Beschreibung von Struktur und Semantik unterschiedlicher Arten von Petri-Netzen. Kategorientheorie stellt ein allgemeines Modell zur Beschreibung von Phänomenen bereit, die in unterschiedlichen Zweigen der Mathematik und Wissenschaft aufzufinden sind. Diese stellt die Basis für die Theorie der Abstrakten Petri-Netze dar, findet aber auch Anwendung in der funktionalen Programmierung. Diese Arbeit erläutert kategorielle Konzepte und deren Zusammenhang mit Abstrakten Petri-Netze sowie der funktionalen Programmierung. Sie stellt ein Softwaredesign vor, für das diese Konzepte die Grundlage bilden.
Moritz Uhlig

Title of the paper
A Type Class Based Approach for Modeling Transformations of Abstract Petri Nets

Keywords
Petri nets, Abstract Petri Nets, functional programming, Scala, net transformations, category theory, type classes, net classes

Abstract
Abstract Petri Nets offer a unified approach to describing the structure and semantics of different kinds of Petri nets. Category provides a common model for describing phenomena found in multiple branches of mathematics and science. It acts as the foundation for the theory of Abstract Petri Nets but also has applications in in functional programming. This thesis explains categorical concepts and their relation to both functional programming and the theory of Abstract Petri Nets. A software design for implementing transformations of Petri nets based on these concepts will be presented.

Contents

1. Introduction

1.1. Motivation and Goals

Petri nets are used to model and analyze concurrent processes with a wide range of applications in chemistry, business and computer science. Based on the original definition there exist numerous variations of Petri nets reaching from low level nets like elementary nets to various high level nets with typed, distinguishable tokens. Abstract Petri nets define a mathematical model for uniformly describing the structure of different classes of Petri nets and transformations between them. When viewed from an implementor's perspective this enables a generic design as it is possible to abstract over the elements describing structure and operations. When looking at the tools for modeling Petri nets it appears that most of them concentrate on graphical modeling and are often limited in the types of Petri nets they support. These types of Petri nets are also in many cases restricted to low level Petri nets, sometimes offering one type of high-level Petri net. There also exist more extensive tools like CPN Tools that are not restricted to graphical modeling but are also extensible via a programming interface. These are by far more expressive than simple graphical editing tools but are still restricted to one certain type of high-level nets. In the case of CPN Tools this is a special form of Colored Petri nets incorporating a custom dialect of the programming language ML for definition of behaviour.

Category theory is an abstract branch of mathematics that is used to examine abstract properties of mathematical concepts. It offers a toolset for describing the general abstract structures in mathematics. Category theory focusses on relations between elements - called objects - instead of the elements themselves. It provides abstractions that are useful in many branches of mathematics and science and defines a common language when working across boundaries of these disciplines. In recent years category theory has also gained importance in both functional programming and the theory of Petri nets. Having a common foundation for modeling the problem domain and structuring a program that implements the model can be a valuable asset. As the concept of

functional programming is based on treating computations as the evaluation of mathematical functions it is a good fit for implementing solutions to mathematical problems.

Scala is a multi-paradigm programming language for the Java Virtual Machine that incorporates many features from functional as well as object oriented languages. It has a very expressive type system that allows encoding much more information in types than is possible in other languages. This has lead to efforts to port libraries that are based on categorical concepts from the functional programming language Haskell to Scala. Libraries like Scalaz[1] allow expressing axioms in the type system and implement programs that act as proofs for these axioms. Influences from other programming languages also lead to patterns to emulate a construct called *type classes* in the Scala language. Type classes are a type level construct for implementing polymorphism that separates data types and operations. The use of type classes often results in more modular and more extensible code. It also limits code duplication as in combination with other Scala features a higher level of abstraction can be reached. Especially when modeling mathematical concepts in type classes this also leads to a higher degree of reusability.

This thesis takes a code centric approach to modeling Petri nets opposed to the more traditional graphical approach to modeling. In particular the goals of this thesis are the following:

1. We show how Scala's programming language features and its type system can be used to model categorical concepts.

2. We analyze how the common foundation of both functional programming and Abstract Petri Nets given by category theory can be leveraged to provide meaningful abstractions usable for the implementation.

3. We inspect how transformations both between two net classes and inside one class of Petri nets can be implemented.

To answer these questions, we perform an in-depth analysis of the Scala programming language. We design and implement a softwares model for the given problem domain and evaluate it in terms of usability and general applicability.

In summary, the contributions of this thesis can be summarized as follows:

- We analyze category theory, functional programming and Abstract Petri Nets as well as the relationships between these disciplines.

[1] Documentation Available at `http://code.google.com/p/scalaz/`

- We try to leverage the common underlying ideas of category theory to deliver an abstraction applicable to multiple instantiations to multiple types of Abstract Petri Nets.
- We design a framework for modeling and transforming Abstract Petri Nets and evaluate the model in terms of applicability and extensibility.

1.2. Structure of the Thesis

The thesis starts off by introducing the categorical concepts used in both the theory of Abstract Petri Nets and functional programming. It continues by explaining fundamental concepts of functional programming along with the Scala programming language. Advanced type system constructs of Scala that are used to implement features not possible in many other languages will be discussed in more detail. Petri nets will be introduced by defining the properties of two distinct classes of Petri nets before the notion of Abstract Petri Nets will be presented. Two different notions of transformation on Petri nets will also be introduced in this section. The implementation part starts with showing the relationship between the concepts involved and continues by implementing data structures and operations. The thesis is closed by a discussion an evaluating the applicability of the model concerning high-level Petri nets and gives some ideas how extension to the model could be implemented for improved support of high-level nets.

Due to the strong coherence of the fundamental theories involved and due to the abstract nature of the problem it is inevitable to also include forward references.

2. Categorical Foundations

Category theory is a branch of mathematics that exists since the 1940s. It is an effort to use a convenient symbolism and common language to describe precisely many similar phenomena. Additionally it provides the means to simultaneously investigate constructions with similar properties that occur in different fields of mathematics and related fields such as computer science and physics.

2.1. Categories

A category **A** consists of

- a class of objects (A, B, ...),
- morphisms (also called arrows) between objects (f, g, ...),
- for each object A an identity morphism 1_A called the identity and
- a composition law for morphisms.

For each pair (A, B) of objects of a category **A** there is a set $\hom(A, B)$ whose members are called **A**-morphisms from A to B. The sets $\hom(A, B)$ are pairwise disjoint (Adamék et al., 2009, 3.1). The source of each morphism $f : A \longrightarrow B$ is called *domain*, denoted $dom(f) = A$, the target is called *codomain* denoted $cod(f) = B$.

Morphisms f, g are can be composed whenever $cod(f) = dom(g)$ yielding a new morphism $h = g \circ f$ with domain $dom(h) = dom(f)$ and codomain $cod(h) = cod(g)$. Composition is associative, i.e. given any morphisms $f : A \longrightarrow B$, $g : B \longrightarrow C$ and $h : C \longrightarrow D$ the equation

$$h \circ (g \circ f) = (h \circ g) \circ f$$

holds.

The identities are **A**-morphisms from an object A to itself and have to also act as identities with respect to composition:

$$1_B \circ f = f = f \circ 1_A.$$

Example 2.1.1. Set is the category whose objects are sets and the morphisms are functions between the sets. The identity is the identity function and composition is given by function composition.

Example 2.1.2. Every monoid $(M, \cdot, e)$, i.e. a semigroup with unit (see Section A.2) can be seen as category with the underlying set M as only object and

$$\hom(M, M) = M, \quad 1_M = e, \quad y \circ x = y \cdot x.$$

Example 2.1.3. The category of a functional programming language L consists of the types of the programming language as objects and the computable functions as morphisms. As in Example 2.1.1 the identity is given the identity function, composition is given by the composition of functions.

A common way to represent objects and morphisms graphically are *commutative diagrams*. These are directed graphs with the objects as nodes and morphism as edges. Commutative diagrams are not only used for visualization but also for proofs. The technique used is often called *diagram chasing* as it is possible to follow the arrows in the diagram to examine certain properties of the morphisms and their compositions. Figure 2.1 shows a diagram with three objects A, B, and C and morphisms f, g and h between them. Saying that this diagram *commutes* is equivalent to saying $h = g \circ f$.

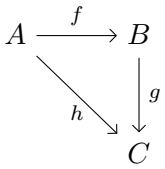


Figure 2.1.: Commutative diagram

Another commonly used notation for categories is the tuple notation. A category $\mathbf{A}$ can be written as the quadruple $\mathbf{A} = (\mathcal{O}_\mathbf{A}, \hom_A, id, \circ)$ where $\mathcal{O}_\mathbf{A}$ denotes the objects of the category, hom the hom-set, id the identity morphisms and $\circ$ the composition.

Category theory concentrates on inpecting the morphisms between objects instead of focussing on the objects of a category.

One very important kind of morphism is called an *isomorphism* which plays a central role in category theory. A morphism $f : A \longrightarrow B$ is called an *isomorphism* if there exists a morphism $g : B \longrightarrow A$ such that $f \circ g = 1_B$ and $g \circ f = 1_A$.
g is then called the *inverse* of f and can also be written as f^{-1}. If an isomorphism exists between objects A and B we say A *is isomorphic to* B written $A \cong B$.

Example 2.1.4. In **Set** isomorphisms are bijective functions between sets. All singleton sets are isomorphic as there exists a unique, trivial morphism between any two of them.

Example 2.1.5. Every category with a set of morphisms is isomorphic to one in which the objects are sets and the morphisms are functions (Awodey, 2010, Theorem 1.6). This is in fact a very important theorem in the context of this thesis. Together with example 2.1.3 this can be used to formally verify that the approach we take later is correct.

Due to the abstract nature of category theory the notion of equality is not of much use in proofs or ideas in general. When examining properties of categories it is often sufficient to check if these properties hold *up to isomorphism*. This means just that the property does not neccesseraly hold for exactly the objects examined but for objects isomorphic to them. A commonly used concept is *uniqueness up to isomorphism* which says that a certain object or morphism may not be uniquely determined but all other possible choices are isomorphic to it.

2.2. Duality

Duality is one important concept in category theory. Given any category $\mathbf{A} = (\mathcal{O}, \hom_A, id, \circ)$ there exists a *dual* category $\mathbf{A}^{op} = (\mathcal{O}, hom_{A^{op}}, id, \circ^{op})$. This dual (or oppsoite) category has the same objects and same identity morphisms. It also has the same morphisms, except for their direction. Informally the dual category can be constructed by reversing the arrows of the original category.

Example 2.2.1. Preordered classes can be considered as a category. The category $\mathbf{A} = (X, \leq)$ has as objects the elements of the underlying class X and a morphism between any two objects for that the ordering relation holds. The dual category of this is $\mathbf{A}^{op} = (X, \geq)$.

For every property of an $\mathbf{A}$-object X $\mathcal{P}_{\mathbf{A}}(X)$ we have an associated property $\mathcal{P}^{op}_{\mathbf{A}}(X)$ that holds in $\mathbf{A}^{op}$.

2.3. Functors

Functors can be viewed as structure preserving morphisms between categories. A functor $F : \mathbf{C} \longrightarrow \mathbf{D}$ is a mapping from category $\mathbf{C}$ into category $\mathbf{D}$ that assigns to each $\mathbf{C}$-object C a $\mathbf{D}$-object $F(C)$ and to each $\mathbf{C}$-morphism $f : C \longrightarrow C'$ a $\mathbf{D}$-morphism $g : F(C) \longrightarrow F(C')$ in such a way that

- F preserves identities:

$$\forall X \in \mathbf{C} : F(1_X) = 1_{F(X)}$$

- F preserves composition:

$$\forall f : X \longrightarrow Y,\ g : Y \longrightarrow Z \in \mathbf{C} : F(g \circ f) = F(g) \circ F(f).$$

A functor that maps a category to itself is called an *endofunctor*.

Example 2.3.1. The (covariant) power-set functor $\mathcal{P} : \mathbf{Set} \longrightarrow \mathbf{Set}$ sends each set A to its powerset, i.e. the set of all subsets $\mathcal{P}A$ of A. For each subset $X \subset A$, $\mathcal{P}f(X)$ is the image $f[X]$ of X under f.

Example 2.3.2. The Grothendieck group construction (see A.3) is a functor from the category of commutative semigroups to the category **Ab** of abelian groups. A morphism $f : S \longrightarrow T$ induces a morphism $K(f) : K(S) \longrightarrow K(T)$ which sends an element (s^+, s^-) to an element $(f(s^+), f(s^-))$.

2.4. Free Functor

The free functor is a generalization of the concept of free constructs known from algebra. These constructs include amongst others free monoids, free groups and free lattices. In terms of category theory these can be expressed as constructs over concrete categories.

As an example we will inspect how the free monoid can be constructed in **Set**. For any set A we can create the *Kleene Closure* of a given by

$$A^* = \{\text{words over } A\}$$

Together with the concatenation operation $*$, defined by $w * w' = ww'$ and the empty word ε as unit this set forms the free monoid. All elements of A can be viewed as words of length one, giving rise to a function $i : A \longrightarrow A^*$ defined as $i(a) = a$.

As every monoid N has an underlying set $|N|$ and every monoid morphism $f : N \to M$ has an underlying function $|f| : |N| \to |M|$ it can easily be seen that this is a functor. The name of this functor is *forgetful functor* as it sends each object with a structure to another object forgetting parts of this structure.

With this we can define the universal mapping property of the free monoid $M(A)$ over a set A.

Having a function $i : A \to |M(A)|$ and given any monoid N together with any function $f : A \to |N|$ there exists a *unique* monoid homomorphism $\bar{f} : M(A) \to N$ such that $\left|\bar{f}\right| \circ i = f$ as in the following diagrams:

in **Mon**:

$$M(A) \overset{\bar{f}}{\dashrightarrow} N$$

in **Set**:

$$\begin{array}{ccc} |M(A)| & \xrightarrow{|\bar{f}|} & |N| \\ {\scriptstyle i}\uparrow & \nearrow_{f} & \\ A & & \end{array}$$

2.5. Natural Transformations

For categories **C**, **D** and functors $F, G : \mathbf{C} \longrightarrow \mathbf{D}$ a natural transformation $\nu : F \longrightarrow G$ is a family of morphisms in **D**

$$(\nu_C : FC \longrightarrow GC)_{C \in \mathbf{C}}$$

such that for every $f : C \longrightarrow C' \in \mathbf{C}$ the *naturality condition*

$$\nu_{C'} \circ F(f) = G(f) \circ \nu_C$$

holds, i.e. the following diagram commutes:

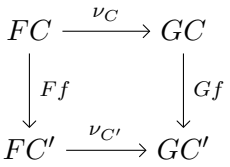
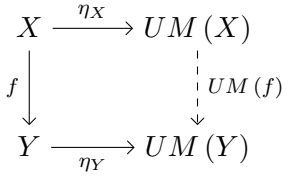


Figure 2.2.: The naturality square

A natural transformation whose components are isomorphisms is called a *natural isomorphism* (Awodey, 2010, Lemma 7.11).

Example 2.5.1. Considering the free monoid $M(X)$ on a set X we define a natural transformation $\eta : 1_{\mathbf{Set}} \longrightarrow UM$ denoted as the insertion of generators, i.e. the function that for every set S takes every $x \in S$ to itself considered as a word.



This is natural because the homomorphism $M(f)$ is completely determined by what f does to the generator.

Natural transformations can also be viewed as *morphisms between functors* and in fact there also exists a category $\mathbf{Fun}(\mathbf{C}, \mathbf{D})$ with functors from $\mathbf{C}$ to $\mathbf{D}$ as objects and natural transformations as morphisms. Isomorphisms in this category are natural isomorphisms (Awodey, 2010, section 7.5).

2.6. Products and Coproducts

Products are a well known concept in many areas of mathematics. One example of products are Cartesian products of sets. The Cartesian product of two sets A, B is a set $A \times B$ of pairs given by

$$A \times B = \{(a, b) \,|\, a \in A, b \in B\}.$$

There are so called coordinate projections π_1, π_2

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

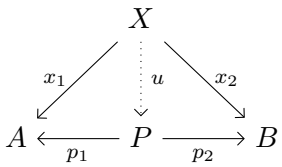
with

$$\pi_1 (a, b) = a$$
$$\pi_2 (a, b) = b.$$

In category theory this can be generalized to products. A product diagram for objects A and B in a category **C** consists of an object P and morphisms $p_1 : P \longrightarrow A$, $p_2 : P \longrightarrow B$ satisfying the universal mapping property that given any diagram of the shape

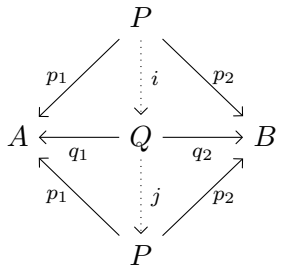
$$A \xrightarrow{x_1} X \xleftarrow{x_2} B$$

there exists a *unique* morphism u such that $x_1 = p_1 u$ and $x_2 = p_2 u$, i.e. the diagram



commutes.

From this definition follows that products are unique up to isomorphism. Considering the diagram



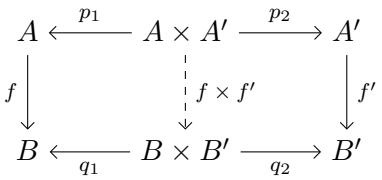
it can easily be shown that P and Q are isomorphic by

$$p_1 \circ j \circ i = p_1 \tag{2.1}$$

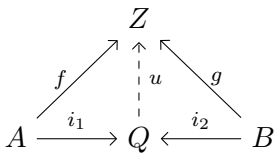
$$p_2 \circ j \circ i = p_2 \tag{2.2}$$

$$j \circ i = 1_P. \tag{2.3}$$

Binary products can also be defined as an functor $\times : \mathbf{C} \times \mathbf{C} \longrightarrow \mathbf{C}$ as described in the diagram below. In order for this diagram to commute we define $f \times f' = \langle f \circ p_1, f' \circ p_2 \rangle$.



The dual of a product is called a *coproduct*. Following the informal way of "reversing the arrows" to get to the dual definition leads to the following diagram

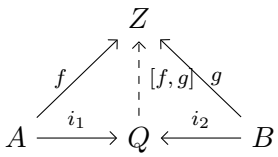


The morphisms $i_1 : A \to A + B$ and $i_2 : B \to A + B$ are called *injections*. They are not necessarily injective but are called this way as they can be used to lift values into the coproduct. As coproducts are dual to products, for any product defined in a category $\mathbf{A}$, there exists a product in $\mathbf{A}^{op}$.

Example 2.6.1. In **Set** the coproduct of sets A, B is the *tagged union*

$$A + B = \{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\}$$

Given any two functions f, g as in



we define

$$[f,g]\,(x,\delta) = \begin{cases} f(x) & \delta = 1 \\ g(x) & \delta = 2 \end{cases}$$

2.7. Adjoints

Comparing the properties of two categories and inpecting their relation to each other is often beneficial. Consider categories **C** and **D** and functors F and G between them:

$$\mathbf{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathbf{D}$$

C and **D** are said to be isomorphic if the following conditions hold:

$$1_C = GF$$
$$FG = 1_D$$

As stated in the beginning of this section instead of proving that two objects are identical it is often sufficient to show that they are isomorphic. This leads to the weaker definition of *equivalence* of categories. Categories **C** and **D** as given above are equivalent if

$$1_C \cong GF$$
$$FG \cong 1_D.$$

One way to describe the difference to the situation when **C** and **D** were isomorphic is to say that the identity natural transformations have been replaced with natural isomorphisms. Thus equivalence could be described as "isomorphism up to isomorphism".

An even more lenient way to describe the relation of to categories is called *adjunction*. $F \dashv G$ (read "F is left adjoint to G") means that there are two natural transformations

$$\eta : 1_{\mathbf{C}} \Longrightarrow GF \qquad \text{(called unit)}$$
$$\varepsilon : FG \Longrightarrow 1_{\mathbf{D}} \qquad \text{(called counit)}$$

satisfying the triangle identites, i.e. making the following diagram commute:

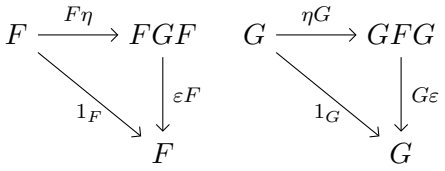


Figure 2.3.: Triangle identities

While this definition provides us with tools (the natural transformations) it is not immediately clear what their use is. Fortunately another definition makes this more easily understandable.

An adjunction consists of functors

$$F : \mathbf{C} \rightleftarrows \mathbf{D} : G$$

and a natural isomorphism

$$\phi : \hom_{\mathbf{D}}(FC, D) \cong \hom_{\mathbf{C}}(C, GD) : \psi.$$

The existence of the natural isomorphism can be interpreted in that for every morphism $t : FC \longrightarrow D$ in $\mathbf{D}$ there exists a unique morphism $\bar{t} : C \longrightarrow GD$ in $\mathbf{C}$.

Unit and counit can be constructed via

$$\eta_C = \phi(1_{FC})$$
$$\varepsilon_D = \psi(1_{GD}).$$

Example 2.7.1. The free functor F is left adjoint to the forgetful functor U which can be expressed as follows:

$$\mathbf{Set} \underset{U}{\overset{F}{\rightleftarrows}} \mathbf{Mon}$$

In this case the unit η sends an element a of a set A to a monoid structure in **Mon** containing the word a of length one. The counit ε then maps each monoid structure $(A^*, *)$ to the underlying set of words A^* in **Set**.

Concerning adjoint situations it is generally more interesting to inspect the relations between the functors than the relations between the categories involved. In fact adjoint situations may even arise between endofunctors as in the following example 2.7.2.

Example 2.7.2. The covariant power-set functor $\mathcal{P}$ is left adjoint to the identity functor I, written $\mathcal{P} \dashv I$.

$$\mathbf{Set} \underset{I}{\overset{\mathcal{P}}{\rightleftarrows}} \mathbf{Set}$$

As $\mathcal{P}$ sends each set to its power set the resulting object is again a set, thus clearly in **Set**.

2.8. Monads

Monads are a construct to describe abstract algebras. A monad on a category **C** is a triple (T, η, μ) consisting of an endofunctor T, and two natural transformations $\eta : 1_{\mathbf{C}} \to T$ and $\mu : T^2 \to T$ satisfying the following diagrams:

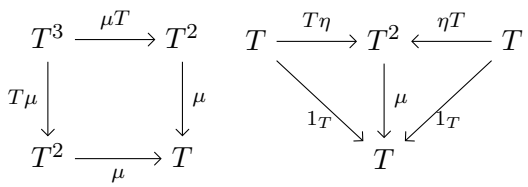


Figure 2.4.: The monad laws

leading to the two monad laws

$$\mu \circ \mu_T = \mu \circ T\mu \qquad \text{(associativity law)}$$
$$\mu \circ \eta_T = 1 = \mu \circ T\eta \qquad \text{(unit law)}$$

Given any adjoint situation $\mathbf{C} : F \dashv G : \mathbf{D}$ there is an associated monad (T, η, μ) with $T = G \circ F$.

The natural transformation η is called the unit of the monad, μ is called multiplication. The unit enables us to lift objects into the context of the monad and the multiplication can be used to flatten a nesting of functors. What is meant by flattening can differ depending on the properties of the adjunction.

Example 2.8.1. The adjoint situation $\mathcal{P} \dashv I$ from example 2.7.2 can be expressed as a monad (T, η, μ) in the following way:

$$T : \mathbf{Set} \to \mathbf{Set} \qquad T = \mathcal{P}$$
$$\eta_X : X \to \mathcal{P}(X) \qquad \eta_X(x) = \{x\}$$
$$\mu_X : \mathcal{P}(\mathcal{P}(X)) \to \mathcal{P}(X) \qquad \mu_X(\alpha) = \bigcup \alpha$$

Here the unit η_X constructs an element of $\mathcal{P}(X)$ by generating a set containing a single element of X. The multiplication μ can be used to flatten a nested power set structure by generating the union of all set elements.

The monad is valid as both monad laws are satisfied. The unit law holds as creating a singleton set from a set and generating the union of all subsets yields the set itself, as does wrapping each element of the set and generating the unit of these. For the associativity law to hold it has to be shown that given a set $x \in \mathcal{P}(\mathcal{P}(\mathcal{P}(X)))$ we can apply the multiplication twice to generate a set $x' \in \mathcal{P}(X)$ and that this set is uniquely determined no matter if we start by creating the union from the outermost level or the innermost level, which can be expressed as the following equation

$$\bigcup\bigcup\{\alpha_1, \ldots, \alpha_n\} = \left(\bigcup \alpha_1\right) \cup \ldots \cup \left(\bigcup \alpha_n\right)$$

As the union operator is associative the condition expressed by this equation clearly holds.

Example 2.8.2. The adjoint situation $F \dashv U$ between free and forgetful functor has an associated monad. Given words $\alpha, \beta, \gamma \in X^*$

$$\alpha = \alpha_1 * \ldots * \alpha_i$$
$$\beta = \beta_1 * \ldots * \beta_j$$
$$\gamma = \gamma_1 * \ldots * \gamma_k$$

it is possible to generate a word $x \in (X^*)^*$ which is a word where the characters themselves are words. The question that immediately arises is how the structure is generated just given the functor and natural transformations. Mapping a set to a monoid structure lets us perform the actions of the monoid given as morphism in the category **Mon**. In this case it is possible to generate the words of length one and using the monoid operation to concatenate them. The forgetful functor can then be used to forget about the monoid structure and map the results to the underlying set.

$$x = \alpha * \beta * \gamma \qquad x \in (X^*)^*$$
$$\mu_X(x) = \alpha_1 * \ldots * \alpha_i * \beta_1 * \ldots * \beta_j * \gamma_1 \ * \ldots * \gamma_k \qquad \mu_X(x) \in X^*$$

It is immediately clear that the monad given by the adjunction $F \dashv U$ satisfies the monad laws. The unit law holds as mapping each character of a word to a word of length one and then concatenating the resulting words produces the word itself. Generating a one character word containing the word w and applying the multiplication to the resulting word also yields the word x. Also the associativity law is satisfied given the fact that the concatenation operation itself is associative. For a nested structure generated by the free monoid the resulting word is the same no matter if the concatenation is performed starting with the innermost words and concatenating these results or if the multiplication is performed twice starting from the outermost word.

3. Functional Programming in Scala

3.1. Functional Programming

Today there are many definitions of what a functional programming language is. The only property that is shared by all definitions is that functions are treated as first class values in that language. In this thesis we will focus on the features outlined in (Hudak, 1989). We will also see that Scala does not necessarily enforce certain features stated as a requirement for being a functional programming language in that definition but encourages the programmer to satisfy them.

3.2. Referential Transparency and Pure Functions

An expression is said to be referentially transparent if it can be replaced by its value without changing the behavior of the program. This implies also that referentially transparent expressions may not cause side effects. Side effects include changing global state in a program, changing a program's control flow by for example throwing exceptions or performing input/output operations. Clearly all constants in a program are referentially transparent. Variables that can be mutated are not referentially transparent as their value changes over time and thus it is impossible to replace all of their occurrences with their respective value. For every external input the same is true as the value of an expression reading from a source depends on the current contents of a file or in the case of user input from what exactly the user enters.

Related to the concept of referential transparency is the notion of *pure functions*. A function is said to be pure if it is referentially transparent for all referentially transparent parameters. In practice this means that the output of a function is completely determined by its input arguments. Problems arise when there are side effects that cannot be avoided. Input and output of a program are side effects that actually make a program do something useful. Additionally a library might be impure because it car-

ries internal state. There are numerous techniques how a program can structured to separate pure from impure code.

Monads are one elegant solution to this problem. We will see in Section 5.2that monads can be used to represent computations. In purely functional programming languages there are techniques for building a structure that represents an impure computation but that is constructed in a pure context. This structure then has to be evaluated in an impure context to perform the unsafe operations. The type system of these languages can be used to clearly separate pure and import contexts.

Another solution are so called *effect systems*. These are mainly of interest in the context of impure languages. A programmer can express the need for certain parts of the code to be referentially transparent. The compiler will then use the effect tracking system to ensure that these requirements are met. While being more complex, effect tracking systems offer the ability to locally use impure code which can lead to a better overall performance. An effect tracking system will ensure that the impure code does not affect any global state in the program thus eliminating global side effects. The guaranteed absence of side effects in parts of a program also enables compilers to perform more aggressive optimizations.

Scala is an impure language and while it encourages the programmer to use concepts from purely functional programming languages it does not enforce them. In Section 5.2 we will see that it is possible to implement monads in Scala. Nevertheless it is also up to the implementor to guarantee the absence of side effects. Currently there are efforts to implement a combined type-and-effect system in Scala (Lukas Rytz, 2012) but at the time there is no possible way to enforce referential transparency.

3.3. Concepts in Scala

Scala combines features of object oriented languages with features known from functional programming. It was designed as a scalable language in the sense that it should be easily extensible by library code. Scala is an object-oriented or - more precisely - a class-oriented programming language where inheritance is achieved by defining classes of objects. Like in many functional languages, everything in Scala is an expression, for example even an `if-then-else`-statement yields a value.

Many of its features also aim to provide syntactical flexibility and extension points for developers to integrate different concepts in a natural way. Examples for this kind of integration are the actor libraries, various domain specific languages and also the

type classes introduced in this thesis. The flexible syntax allows developers to make the usage of code provided as library appear as if that code was implemented as a language feature.

3.4. Types

3.4.1. Classes, Traits and Objects

Classes in Scala are similar to classes in Java with some important differences. Besides the syntactic differences the main difference is that a class has exactly one primary constructor that has to be called by all secondary constructors.

A sample class definition in Scala could look like the one given in Listing 1.

```
class A(x: Int, val y: Int) {

  def this() = this(0, 0)

  val sum = x + y

  def addSum(q: Int) = q + sum

  println("Initialized")
}
```

Listing 1: A simple Scala class

Constructor parameters can be made class members by prefixing them with the keyword `val` for an unmodifiable or `var` for a modifiable value member. All statements inside the body of a class are promoted to the constructor of the class.

There are two different types of fields in Scala. Member values are immutable fields that cannot be changed after they are initialized and are declared using the `val` keyword. Variables declared using the keyword `var` on the other hand can be mutated after initialization.

Methods in Scala are introduced with the keyword `def`. The head of a method can either be followed by a block or by an equals-sign 'assigning' the method body to the head of the method. In the first case the method's return type will be `Unit` which is similar to Java's `void` type. An important conceptual distinction has to be made between these types. A method declaring `void` as a return type does not return a value at all.

This is opposed to Scala's concept of treating everything as an expression. Like many functional programming languages, Scala thus defines the `Unit` data type of which exactly one instance, namely `()` exists. When the method is declared by assigning a body to it the right hand side can be an arbitrary expression. The methods return type can be defined by adding a type ascription to the method's head. A type ascription is not necessary though if the type of the expression on the right can be determined by the compiler. Scala supports type inference, which means that in many cases the compiler is able to type an expression correctly without the programmer needing to supply any type information. Methods in Scala usually do not explicitly return a value unless it is needed, for example when type inference is impossible or the inferred type is to narrow.

In contrast to Java, Scala supports multiple parameter lists for a method. This is one important feature that lets the programmer implement methods which when invoked are visually similar to control structures.

Like Java classes, Scala classes can be marked abstract by using the keyword `abstract`. Abstract members are introduced only as declaration of the member without assignment of a value or body in case of methods. In contrast to other languages the `abstract` keyword is not used for member declaration.

Multiple inheritance in Scala is supported via *traits*. Traits are similar to interfaces but they may also contain implementations of methods or value members. Traits can be composed using the `with` keyword.

Both traits and classes can specify a *self-type* to express a dependency on another type being part of the resulting type. Technically specifying a self type is done by typing the self reference of a type. This self reference is just an identifier at the beginning of a type's body followed by the characters `=>` and is used to define an alias for the value identified by `this`. A self type is then defined as a type ascription on the self type. It is common to use the identifiers `this` or `self` for the self reference but any name can be used. A trait or class with a self type declaration may only be instantiated if the types are mixed in. This can either be done by explicitly creating a subtype that either that includes the required type in a `with`-clause or by dynamically mixing in required traits upon instance creation. The first method is shown in `class C` in Listing 2, the second method is used at the initialization of the value `x`. Trait `E` shows another use case for self references by making the outer instance accessible inside the traits body via the name `outer`.

```
trait A { def foo: Int }

trait B { this: A =>
  def bar = foo + 1
}

class C extends A with B { def foo = 0 }

trait D { def foo = 2 }

val x = new A with B with D

trait E { outer: A with B =>
  class Innner { inner =>
    val foo = 23
    def bar = outer.foo + inner.foo
  }
}
```

Listing 2: Self types

Scala is completely object oriented and does not have a notion of static members. As a replacement for these Scala has singleton objects. Singleton objects are declared using the keyword `object` and the compiler will automatically generate the code necessary to implement the singleton pattern correctly. Being real instances singleton objects can extend classes and traits. Static members defined in Java code can be used without restrictions in Scala.

```
object Util extends SomeTraitOrClass {
  val prefix = "x.y.z"

  def add(x: Int, y: Int) = x+y
}
```

Listing 3: A singleton object

3.4.2. Companion Objects and Case Classes

A special case of objects are companion objects. A companion object for a class is an object with the same name as the class, specified in the same compilation unit. Com-

panion objects share the scope of the class and thus can access private members of the class and the class (called *companion class*) can access private members of the companion object. They are often used for defining constants or utility and factory methods for classes.

As classes are often used for defining record types, Scala has a special construct called *case classes* for facilitating the definition of immutable record types. When a class declaration is prefixed with the keyword `case` the compiler will generate additional code for this class. The first difference to regular classes is that all constructor arguments are stored as fields in the class and public accessors are generated. The compiler generates `toString`, `equals` and `hashCode` methods based on the costructor parameters and also a method called `copy` that can be used to create a modified version of the instance. In addition, a companion object defining an `apply` method for creating instances and an `unapply` method that is used for pattern matching is automatically generated. Listing 4 shows a definition of a case class and an example use. The keyword `new` is not necessary when creating an instance as the call to `Person(...)` is dispatched to the `apply`-method of the companion object. The second assignment shows the use of the generated `copy`-method. This method supplies default values for all of its parameters. Scala supports named arguments which means that the parameters can be given in any order when prefixed with their names and the = sign. These two features combined allow the use of the `copy`-method in the listing.

```
case class Person(lastName: String, firstName: String, age: Int)

val peter = Person("Lustig", "Peter", 75)
val petra = peter.copy(firstName = "Petra", age = 73)
```

Listing 4: Case classes

3.4.3. Polymorphic Expressions

Polymorphism allows handling of different data types using a uniform interface. In this subsection we will inspect three different kinds of polymorpisms in programming languages, namely *subtype polymorphism*, *parametric polymorphism* and *ad-hoc polymorphism*. While the first two kinds of polymorphism should be familiar to the reader the latter is not that commonly known.

Java, until version 1.5 had only one user controllable feature for implementing polymorphism, namely *subtype polymorphism*. In 1996 Martin Odersky and Philip Wadler

initiated a project called Pizza that had the goal to bring some of the features of functional programming to the Java Platform. About one year later Gilad Bracha and David Stoutamire approached them and expressed their interest in the language's support for *parametric polymorphism* and they decided to start another project called *GJ*. Most of the work on parameteric polymorphism done in Pizza was incorporated into GJ which later became the *generics* in Java 1.5. Pizza also included some other concepts from functional programming and parts of that became the foundation of what we know as Scala today.

Subtype Polymorphism

Subtype polymorphism is commonly used in object oriented programming languages. Formally it defines substitution rules via an ordering relation $<:$ on the types used in the programming language. Any term of type T can be safely substituted by a term of type S whenever $S <: T$ holds. So S *is a subtype of* T. Behaviour of subtypes can be adjusted by overriding methods in the subtype. In most object oriented programming languages like Java and Scala the technique used to implement subtype polymorphism is called *dynamic dispatch*. As a compiler may not be able to determine the method being called statically it generates code that dispatches the method call dynamically at runtime.

Parametric Polymorphism

As mentioned parametric polymorphism is commonly used in statically typed, functional programming languages. It enables the programmer to define *generic functions* and *generic datatypes* by adding type parameters to their respective declarations. Generic functions and datatypes can in general handle all types that are supplied as arguments. The code being executed is generally shared for all types and this is in contrast with *ad-hoc polymorhism*. While that code is shared the integration of parametric polymorphism with subtype polymorphism is difficult. One significant problem arising immediately is the loss of the ability to decide whether or not the rules for substitutionability hold. For example, given types `A <: B` and a type constructor `type F[X]` no assumptions about the relation between `F[A]` and `F[B]` can be made. When there is no relation the type parameter `X` of `type F[X]` is said to be *invariant*. We can actually express the fact that for all types `A <: B` either `F[A] <: F[B]` or `F[B] <: F[A]` holds. In the first case our type parameter would need to be *covariant* as in `type F[+X]` and *contravariant* in the

second case, denoted `type F[-X]` respectively. The +/– signs are called *variance annotations* and a type parameter is invariant when they are omitted. This feature is one of the more important differences between the Java and the Scala type system. Java only allows the variance annotations at call site as a generic wildcard, while Scala supports variance annotations at definition site.

Ad-hoc Polymorphism

Ad-hoc polymorphism is a kind of polymorphism where the code executed when evaluating an expression depends on the types of the parameters.

Many readers will have already been exposed to it when writing code in a language like Java and will know it as method overloading in that context. This allows the definition of multiple methods with the same name differing only in the types of their parameters. The compiler will then choose the correct implementation. The Java compiler also has builtin support for ad-hoc polymorphism regarding the operators defined in the Java Language. As an example we choose the operator `+` as its behaviour is overriden in multiple ways. Considering the expression `a + b`, its type clearly depends on the types of `a` and `b`. If one of both is of type `String`, the expression will also be typed as `String` and string concatenation will be performed. When both arguments are numeric types the compiler will generate code that performs the addition specific to that primitive type. In case both arguments are of different type it will also generate code to convert one of the two parameters to the other as defined by the rules for numeric widening in the Java Specification. All of this behaviour is built into the compiler and thus not extensible by a user in application code.

There are other languages though that do not support subtype polymorphism but that support ad-hoc polymorphism. Clearly these languages must provide other means of defining operations for types. One possible solution are type classes that are introduced later in Section 3.5.

3.4.4. Generic Types

Generic types, as already known from Java are used to implement parameteric polymorphism in Scala. Classes, traits and methods may be parameterized by type. Type parameters are enclosed in square brackets and may be restricted by defining bounds on the parameters. In contrast to Java, Scala supports variance annotations at definition site of a type parameter. An upper bound `U` on a type parameter `A`, written `A <:`

`U` restricts the type parameter to subtypes of `U`, a lower bound `L` as in `A >: L` to supertypes of `L` accordingly. When no variance annotations are present on a type parameter this type parameter is treated as *invariant*. Whenever there are two types `T[A]`, `T[B]` no subtype relation between the two types exists, unless of course both `A` and `B` are the same type. *Covariance* is expressed by prefixing the type parameter with a +-sign, *contravariance* by prefixing it with a –-sign. If a type parameter is marked as covariant as in `class T[+X]` this means that for types `A`, `B` with `A <: B` also `T[A] <: T[B]` holds. A type parameter that is declared contravariant like in `class V[-X]` results in `V[A]` being a supertype of `V[B]` (written `V[A] >: V[B]`) whenever `A <: B`. Type parameters for classes and methods can be further restricted by *view bounds* or *context bounds* introduced in section3.4.9.

3.4.5. Function Types

Being a functional programming language, Scala of course supports function types. As a Scala program is compiled to Java bytecode and the Java Virtual Machine has no notion of function types these are represented as Java classes at runtime. The Scala standard library defines traits `Function1` to `Function22` where the number at the end of the name represents the arity of the function. The limit of 22 parameters is arbitrarily chosen as it is sufficient for most use cases and has no technical reasons. Every `FunctionN` trait is parameterized by type with one type parameter for the return type and one type parameter for every argument type. Each of these traits defines a method called `apply` with given arity.

Given two types `A` and `B` the type of a function from `A` to `B` is denoted by `A => B`. n-ary function types are denoted by enclosing the argument types in parentheses as in the type `(A,B,C,C) => A`. The compiler will resolve these types to the corresponding `FunctionN` trait.

Defining functions can be done in several ways in Scala. Listing 5 shows some commonly used styles. The first four functions defined in that example simple convert an integer value into a string. The first two examples use type ascription to tell the compiler the type of the parameters of the function. A function literal can be given by specifying a parameter list followed by the symbols => and the body of the function. If the body consists of more than one expression the function literal has to be enclosed in braces as is done for functions `f` and `h`. A commonly used abbreviation is to use an underscore in a function literal that marks the position where the parameter shall be inserted.

```
val f: Int => String = { x => x.toString }
val g: Int => String = _.toString
val h = { (x: Int) => x.toString }
val k = (_:Int).toString

val add = (x: Int, y: Int) => x + y
val addThree = add(3, _:Int)
```

Listing 5: Different definitions of functions

Scala distinguishes between methods and functions. Methods are no first class objects so that they cannot be stored in a field or passed to a function as a parameter. There is however a way of converting methods into functions called η-expansion. Listing 6 shows two examples of η-expansions by converting the method `foo` to a function and storing it as a value. This conversion can be triggered in multiple ways. An expression referencing a method will be converted to a function whenever the compiler has sufficient information about the expressions expected type. The value `f` in the example code shows exactly this case. In the case where there is not enough information about the expected type it is possible to trigger η-expansion by suffixing the expression referencing the method with an underscore as it is done in the initialization of value `g`.

```
def foo(x: Int) = x.toString
val f: Int => String = foo
val g = foo _
```

Listing 6: η-expansion

In Scala, values that are not functions but define a method called `apply` can be used as if they were functions. The compiler will translate the function application to a call to the value's `apply` method.

```
object NotAFunction {
  def apply(x: Int) = x + 1
}

println(NotAFunction(12)) // prints 13
```

Listing 7: Apply method

3.4.6. Algebraic Data Types and Pattern Matching

Algebraic data types are composite types that are commonly found in functional programming languages. As already mentioned most functional programming languages do not support subtype polymorphism. For practical reasons of course they have to offer the ability to define custom data types.

Algebraic data types fall into two categories, namely *product types* and *sum types*. Product types can be used to describe record types or tuples. Sum types offer the ability to define a type given by multiple data constructors. The name stems from the fact that a sum type can be viewed as a tagged union of types. It is important to stress out that in most languages supporting algebraic data types as a language feature there is no subtype relation required between these types.

Scala has no direct support for algebraic data types but its features can be used to emulate them. Product types can simply be implemented as case classes. Sum types can be implemented using sealed class hierarchies.

Example 3.4.1. A linked list can be expressed as an algebraic data type. The list has two data constructors: the empty list as a constant and the `Cons` data constructor that takes an element and a list to construct a new list. With this recursive definition the data type list is fully described. This definition can also be considered as sum type: a list is either the empty list or a non-empty list, written $List(A) = Nil + (A \times List(A))$.

```
trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](head: A, tail: List[A]) extends List
```

Listing 8: List data type in Scala

Algebraic data types are not directly supported by Scala but can be emulated using case classes for implementing product types and subtype polymorphism for implementing sum types. Pattern matching is a useful feature when working with algebraic data types. Visually similar to `switch-case` statement it appears similar but has some important differences. The clauses inside a pattern matching expression are not restricted to constants but can also contain so called extractor patterns. It is possible to declare variables inside an extractor pattern that will be bound to the corresponding values when the expression matches the supplied type. Examples of the syntax of pattern matching in Scala is given in Listing 3.4.2. Following the rule that everything is an expression, a `match-case`-block returns a value.

Example 3.4.2. Using the list data type as defined in example 3.4.1 we can define several operations on that data type.

```
def length[A](list: List[A]): Int = list match {
  case Nil => 0
  case Cons(_, tail) => 1 + length(tail)
}
```

The length is computed by recursively traversing the list. Two cases are contained in the `match`-clause: the case handling the empty list that yields the value `0` and the case handling a nonempty list. In this case 1 is added to the length of the list's tail. This case is written using an extractor pattern binding the list's tail to the name `tail` and ignoring the value of the list's head by providing an underscore in the pattern.

```
def headOption[A](list: List[A]): Option[A] = list match {
  case Nil => None
  case Cons(x, _) => Some(x)
}
```

The method `headOption` provides a safe way of accessing the list's head. `Some` and `None` are the only subtypes of a trait called `Option` that represents an optional result. Pattern matching is performed to check if the argument is a non-empty list and in that case the value `Some(x)` is returned, where `x` is bound to the list's head. The value `None` is returned in case an empty list is passed.

Example 3.4.3. A binary tree with values in the branches can be defined as an abstract data type in the following way:

```
sealed trait Tree[+A]
case object Empty extends Tree[Nothing]
case class Branch[+A](value: A, left: Tree[A], right: Tree[A]) extends Tree[A]
```

The keyword `sealed` ensures that the class hierarchy can only be extended in the file that defines the trait. There are two data constructors defined for the tree, the constant `Empty` that describes the empty tree and the case class `Branch`, a ternary tuple containing the value and two child nodes.

The variance annotation on type parameter `A` tells us that for any two types `T`, `U` for which `T <: U` holds, also holds `Tree[T] <: Tree[U]` (see Subsection 3.4.3).

```
def traverse[A,B](t: Tree[A], f: A => B) = {
  def exec(t: Tree[A], ys: List[B]): List[B] = t match {
    case Empty => ys
    case Branch(x, l, r) =>
      exec(r, exec(l, f(x) :: ys))
  }

  exec(t, Nil).reverse
}
```

It should be noted that the examples involving recursive definitions used in this chapter are for illustrative purposes only. As recursive invocation of the methods uses linear stack space this implementations will fail for highly nested data structures. The methods can alternatively be given as a tail-recursive definition, which means that the last call inside the method is a call to itself. In this case an optimization technique called tail-call-elimination can be used to transform the code into an equivalent loop.

3.4.7. Higher Kinds

Higher kinds are another example of type constructs that do not exist in languages like Java. For a better understanding of the idea behind this concept we first need to introduce the notion of *type constructors*. A type constructor can be used to generate a new type by applying type parameters to it, just like a value constructor is used to construct a new value by applying it to value parameters Moors et al. (2008).

The most prominent example for type constructors are generic collection types. For example a generic list type that is parameterized by the type of its elements is not a type by itself but a type constructor. So *List[String]* is a type while *List* is the type constructor.

Scala supports higher kinded type parameters, meaning that type parameters can be type constructors. An example application is shown in Listing 9. A type constructor can be specified as a type parameter by chaging the parameters' occurrences to an underscore. Type constructors of arbitrary arity are supported, for example unary type constructors are written `F[_]`, binary type constructors `F[_,_]` and so on. Inside the body of the parameterized type the type constructor can be applied to a given type as can be seen in the method `lift` in the example. It is also possible to restrict the type constructors further by adding variance annotations like for example `M[-_,+_]` expresses

that the first parameter of the type constructor is contravariant while the second is covariant.

```
trait Lift[F[_]] {
  def lift[A](x: A): F[A]
}

object LiftList extends Lift[List] {
  def lift[A](x: A) = x :: Nil
}
```

Listing 9: Higher kinded type parameters

3.4.8. Type Aliases and Type Members

In Scala objects may not only have value members but in addition to that they can have type members. Just like value members can be used to store a reference to a value, or for primitive types the value itself, type members reference another type. Also like value members, they can either be abstract or concrete. Abstract type members can be overridden in subtypes . But once a type member is fixed it cannot be overridden anymore. Implementing an abstract type declaration is either done by aliasing another type (Listing 10, trait `X` and object `B`) or by adding a concrete type named exactly like the type member to the extending type (Listing 10, trait `Baz` and object `A`).

It seems confusing at first that there is no subtype relation between the traits declaring the type member and those defining the type. This example is chosen on purpose to point out that such a relation does not have to exist. The declaration of the type member only expresses that it can be set somewhere in a subtype but does not specify where. Type members can be fixed by any type that is mixed in and may even be defined in a structural refinement (see Section 3.4.10).

Scala adds the notion of *path-dependent-types*, which are types associated to a value that also depend on the selection path of that value. The implications are best explained using the simple example from Listing 11

The listing defines a class `A` with a nested case class `Value` and a mutable field `value` of that type. Two instances called `a` and `b` are created. The code in the next line that tries to set `b`'s value into a will cause the compiler to raise a type error. This is due to the fact that inner classes in Scala are path-dependent types.

```
trait Foo {
  type Bar
  val theValue: Bar
}

trait X {
  type Bar = Int
}

trait Baz {
  case class Bar(i: Int)
}

object A extends Foo with Baz {
  override val theValue = Bar(12)
}

object B extends Foo with X {
  override val theValue = 0
}
```

Listing 10: Abstract type members

Paths are not types themselves but can be part of named types. In the example code `a` is not only of type `A` but also carries more specific type information, namely the type `Program.a.type`. The most specific type of the value `b` is `Program.b.type`. Inner types of the instances are members of these types respectively leading to the aforementioned type error. In the code above the field `a.value` is of type `Program.a.Value.type` while `b.value` is of type `Progran.b.Value.type`. Even if both values `a` and `b` are initialized to exactly the same instance the compiler will refuse to typecheck the assignment operation unless a type ascription is added to the declaration of the referencing value like for example `val b: a.type = a`. In that case `b` would be of both mentioned path dependent types.

The last line of that listing gives an example of a *type projection* to reference the type disregarding the selection path.

As type declarations and type members can be equipped with a type parameter clause they can be used for type level programming. Listing 12 shows some example uses of this feature.

```
class A {
  case class Value(x: Int)
  var value: Value = Value(0)
}

object Program {
  val a = new A
  val b: a.type = a

  a.value = b.value // type error!

  val x: A#Value = a.value
}
```

Listing 11: Path dependent types

```
object Example {

  type paM[A,B] = Map[B,A]

  type StringMap[A] = Map[String,A]

  type IdObjectMap[A <: AnyRef] = Map[A,A]

  type Id[A] = A
}
```

Listing 12: Type aliases

The first declaration shows that it is possible to swap the type parameters of a higher kind using type aliases. This is useful when working with higher kinds and the kind to be used as a type parameter only differs in order of the type parameters.

The second declaration `StringMap` shows an example of partial application of type parameters. A unary type constructor is created by fixing the first type parameter of the binary type constructor `Map` to `String`.

The type alias `IdObjectMap` restricts the type parameter to be a subtype of `AnyRef`, thus referring to a class type. The type parameter is then inserted for both type parameters of the `Map` type constructor, thus yielding a type that describes a map that maps objects of a certain type to objects of the same type.

The declaration of type alias `Id` deserves some special attention. This declaration is called the *identity type constructor* which for any type just 'constructs' the type itself. We will later see that by using higher kinds we are able to lift the level of abstraction in parts of our code. By having the identity type constructor some of the operations that normally require a higher kind as a type parameter can be performed to simple types by introducing the identity type constructor.

3.4.9. Implicit Parameters

A very powerful concept used in the Scala Language are implicit parameters. As their name suggests these are parameters that are not explicitly passed by the programmer but inserted implicitly by the compiler. For this to work the value of the parameter has to be fully determined by its type. A parameter can be marked implicit by prefixing it with the keyword `implicit`. The implicit parameters always have to be specified in the last parameter list of a function and the list may also only contain implicit parameters. Members labeled with the keyword `implicit` are eligible for being passed as an implicit parameter by the compiler.

Types of implicit parameters can also include any type in the surrounding scope. This includes type parameters of the method and type members or type parameters of the enclosing classes.

Implicit parameters are made available to the compiler by importing them directly or by placing them in the *implicit scope* of their type. The implicit scope of a type T consists of all companion objects that are a part of type T. The parts of a type T are defined as

- for a compound type T_1`with`$\dots T_n$ the union of all types $T_1, \dots, T_n$ and T itself
- for a parameterized type $S[T_1, \dots, T_n]$ the union of the types $S, T_1, \dots T_n$
- for a singleton type p.`type` the parts of the type p
- for a type projection $S\#U$ the parts of S and T itself
- in all other cases just T itself

Implicit values defined at call site of a method or explicitly imported always have higher priority. Whenever there are multiple eligible arguments in scope the most specific one with regards to overloading resolution is chosen (Odersky, 2011, 6.26.3). If the

compiler cannot uniquely determine an implicit argument to a function it will generate an error.

Generally it is considered good style to declare the implicit values or methods inside of a companion object. The priority of an implicit value can be lowered by declaring it in a super type of the companion object. This enables library users to override the values in their own code without having to resort to importing the implicit declarations explicitly.

Implicit Conversions and View Bounds

Unary functions as implicit parameters can be used by the Scala compiler to implement implicit conversions. Implicit conversions are used by the compiler in some cases when an expression does not typecheck. Given an expression e of type `A` there are two cases that trigger the compiler to try to implicitly convert this expression. The first case is when the expression is used in an incompatible way, for example when the expression is passed as a parameter to a function that takes a parameter of a different type `B` would cause the compiler to check whether an implicit conversion `A => B` is present. The second case is when a selection to a member is performed, like in `e.x` where type `A` does not define a member `x`. This is the more complex case as in this case the compiler has to check all implicit conversions from `A` to any other type to determine whether there is exactly one type defining a member `x`. When there are multiple conversions to different types defining that member the compiler generates an error.

Implicit conversions can either be defined as an implicit value of the function type or even as a method. The latter also enableds the programmer to define generic conversions by parameterizing the method on parameter or return type. It is also possible for methods that provide implicit conversions to take implicit parameters.

View bounds offer an abbreviated way of expressing a dependency to an implicit conversion. Instead of adding an implicit parameter of type `A => B` it is possible to add view bounds to the parameter list. This kind of dependency would be expressed by restricting the type parameter `A` with the view bound `A <% B`. Internally the compiler translates this to the exact same signature as before.

The implicit conversions are of course eligible for being passed as an implicit parameter inside the methods body. Furthermore the compiler will try to apply the implicit conversion if inside the body a selection of a member not present on the value is performed or if the value is used in a way that is not compatible to the parameter's type.

Example 3.4.4. Given the method declaration of `foo` as follows

```
def foo[A <% String](x: A, y: A) = x concat y
```

the compiler will translate the method to a method with the following signature

```
def foo[A](x: A, y: A)(implicit evidence$1: A => String): String
```

As no information about type `A` is available to the compiler it cannot find a method named `concat` defined on that type. It will apply the conversion to `String` as that defines a method with that name. This method only accepts an instance of type `String` as an argument, leading to the conversion to be also performed for parameter `y`. The body of the method `foo` would thus be rewritten to

```
evidence$1(x).concat(evidence$1(y))
```

One main use of implicit conversions is to add methods to existing types. Technically these methods are not added to the type itself but defined in a wrapper class. The wrapper class holds a reference to the value that is converted from and implements the method that shall be added. This technique is also known as *enrich-my-library pattern*. The automatic conversion that is triggered by selection of undefined members will then create the wrapper class and invoke the method on it. It is however not possible to overwrite or even overload methods using this pattern as the compiler will not attempt to convert a value when the selection refers to an existing member. An example adding the method `square` to the `Int` type can be seen in Listing 13.

```
class RichInt(x: Int) {
  def square = x*x
}

implicit def enrich(x: Int) = new RichInt(x)

println(12.square) // prints 144
```

Listing 13: Enrich-my-library pattern

Type Classes and Context Bounds

Implicit parameters can be used to model the concept of type classes known from other programming languages, Haskell being the most prominent example. Type classes are

a technique for seperating data type definitions from operations. A type class can be considered a set of operations defined for a certain data type.

For a simple example we will explain the concept of a monoid modelled as a type class. A monoid $M = (A, \cdot, e)$ on a set A with the binary operation $\cdot : A \to A$ and unit $e \in A$ can be modelled as the trait in Listing 14. The set A is represented by the type parameter `A`, unit and operation are defined as methods. This trait is the definition of the type class but to make use of it we will need an instance for a fixed data type. One possible instance would be the monoid for integer addition $(\mathbb{Z}, +, 0)$ as implemented by the class `IntAdditionMonoid`.

```
trait Monoid[A] {
  def unit: A
  def append(x: A, y: A): A
}

object Monoid {
  implicit def intAddition: Monoid[Int] = new Monoid[Int] {
    def unit = 0
    def append(x: Int, y: Int) = x + y
  }
}
```

Listing 14: Monoid type class and instance for integer addition

In this example the method for supplying values eligible as implicit parameters that was introduced in 3.4.9. By adding the declaration of `intAddition` to the companion object `Monoid` it is globally available to the compiler.

We encounter one problem with the aproach of using type classes for implementing monoid operations right here. The type signature of the `Monoid` type class instance contains only the underlying set of the monoid, i.e. `Int` representing the set of integers $\mathbb{Z}$ but no information about the operation of the monoid. Adding this information to the datatype can be done in several ways - the easiest being to introduce a wrapper class. An example can be seen in Listing 15 where a new type `IntMultiplication` is introduced for integer multiplication.

```
case class IntMultiplication(value: Int)

object Monoid {
  // ...
  implicit def intMultiplication: Monoid[IntMultiplication] =
    new Monoid[IntMultiplication] {
      def unit = 1
      def append(x: IntMultiplication, y: IntMultiplication) =
        IntMultiplication(x.value + y.value)
    }
}
```

Listing 15: Monoid type class for integer multiplication

For expressing the dependency to the type class instances we simply declare them as implicit parameters to the methods that use the type class. The methods can also abstract over the actual type of the parameter now. This is exactly the type of ad-hoc polymorphism that was mentioned before. An example usage can be seen in Listing 16. The method `foldLeft` used in this example takes two arguments: a start value and a binary function. This function will be called for every element, passing the result of the previous computation and that element of the list. As the name of the method already suggests it is used to compute the sum of the given list using the `Monoid` instance.

```
implicit def sum[A](xs: List[A])(implicit m: Monoid[A]) =
  list.foldLeft(m.unit) { (a,b) => m.append(a, b) }
```

Listing 16: Usage of the monoid type class

Context bounds offer an alternative syntax for declaring a dependency on an implicit parameter of a type that results from a type constructor application. Context bounds are mainly used when the implicit parameter is not used directly but only needed as an implicit parameter for other method invocations. To add a context bound to a type parameter the parameter is suffixed with a colon and the name of a unary type constructor. The following Listing 17 shows an example use.

```
def sums[A : Monoid](xs: List[List[A]]) = xs map sum
```

Listing 17: Context bounds

The compiler translates this code into a method taking an implicit parameter of type `Monoid[A]`. As it is eligible for being passed as an implicit parameter inside the method it will be passed to the method `sum` from Listing 16.

Context bounds can also be used for expressing the intent of the implicit parameter to be used as a type class. Scala's standard library defines the method `implicitly[A]` that takes an implicit parameter of type `A` and simply returns that value. This method can be used to access implicit parameters inside a method with context bounds.

3.4.10. Structural Types and Type Lambdas

Although Scala mainly uses nominal typing it has limited support for structural typing as well. Structural types are *type refinements* describing the structure of a type.

```
type HasActMethod = {
  def act(): Unit
}
type HasCloseMethod = {
  def close(): Unit
}

def safeAct(ac: HasActMethod with HasCloseMethod) =
  try { ac.act() } finally { ac.close() }
```

Using instances of structural types in programs should generally be avoided. Structural types are only available at compile time. Programs performing runtime checks against structural types can be compiled emitting a warning but will fail at runtime. Secondly from a performance perspective as member access is implemented using reflection and thus is always slower than using static calls.

Structural types have other uses in Scala though. As the refinement part of a structural type can be arbitrarily complex they are useful when programming in the type system. The techniques used are the same that we have seen in part 3.4.8 but offer more flexibility. A common use case is to implement partial application of type parameters using structural types as shown in Listing 18.

```
type FixOne[M[_,_],A] = {
  type Apply[B] = M[A,B]
  type Swap[B] = M[B,A]
}

trait X[F[_]]

class A extends X[FixOne[Map,String]#Apply]
class B[M[_,_]] extends X[FixOne[M,Int]#Swap]
```

Listing 18: Partial application of types using structural types

Nested type declarations inside of the structural refinement can refer to the abstract types parameter and perform partial application and switch parameters just as is possible when using type declarations. The difference is though, that structural types can be used to define libraries on a type level as the nested types can be viewed as something similar to functions but on a type level.

In the example class `A` extends the trait `X` with the type parameter `F` set to type constructor `Map` with the first type parameter set to `String`, the second one unset yielding a unary type constructor. This could also be implemented using type declarations, the difference being only that one declaration has to be added for each application. The second example is more interesting as it shows functionality not possible with just type declarations. Here a higher kinded type is 'passed' to the `FixOne` type and at the same time the parameters are swapped in that type. The result is that `B` extends `X` with the parameter `F` set to the provided type constructor `M` with the first parameter unset and the second one fixed to the type `Int`.

This concept was even extended to implement a feature called *type lambdas*. The idea behind type lambdas is that for partial application it should not be necessary to have to define a type declaration for every combination of types possible. Type lambdas define an anonymous structural type that declares the type member representing the kind. This type member is by convention often called λ. A type lambda can be passed directly as a type parameter and can refer to all types defined in the current scope including the type parameters given to a class. An example application setting can be seen in Listing 19. Here the type lambda is used to fix the first parameter of the `Map` type constructor to the class' parameter `T` yielding a unary type constructor.

```
trait Functor[F[_]] {
  def map[A,B](t: F[A], f: A => B): F[B]
}

class MapFunctor[T] extends Functor[({type λ[α] = Map[T,α]})#λ] {
  def map[A,B](t: Map[T,A], f: A => B) = t.mapValues(f)
}
```

Listing 19: Type Lambda

3.5. Type classes

Type classes are a type level construct that supports implementing ad-hoc polymorphism. For the reader to see why type classes are useful, we first need to take a look at the concept of ad-hoc polymorhism. This kind of polymorphism allows a value to exhibit different behaviour when viewed at different types. In object oriented programming languages this can be achieved by overloading methods, which is done by providing multiple versions of the method with different type signatures. At compile time the correct method will be resolved and thus the correct implementation for that type is chosen.

There are some problems connected to this way of implementing ad-hoc polymorphism though. In impure object oriented languages the concept of encapsulation is used to limit the effects to operations inside a type's scope. Data hiding is another paradigm that is applied to narrow the scope of data mutations even further. These to concepts lead to the fact that it is determined at design time whether or not the data type is extensible.

In addition to these problems there is a technical problem concerning overloading resolution when combined with parametric polymorphism. Many languages perform type erasure at compile time which means that the compiled code does not retain all information of the programs type parameters. When a method signature contains a parameterized type this information will be replaced by a more generic type in the compiled code.

An alternative way of implementing ad-hoc polymorphism are the so called type classes. In programming languages like for example Haskell they are the default construct used for implementing ad-hoc polymorphism. To understand how they work

without an introduction to Haskell we will just look at the concepts used in the language around type classes.

We can avoid the problems of ad-hoc polymorphism mentioned before by only allowing immutable data structures and by keeping the definitions of operations separate from the data type definitions. A type class is basically a set of operations supported for a given type. In Scala this we can define something like this as a parameterized trait. All operations that are supported by this type class are just methods declared inside the trait. The operations can then be implemented in an object.

```
case class Point(x: Int, y: Int)

trait Eq[A] {
  def equal(x: A, y: A): Boolean
}

object EqPoint extends Eq[Point] {
  def equal(p: Point, q: Point) = (p.x == q.x) && (p.y == q.y)
}
```

Listing 20: A type class for defining equality

With just this declaration and instance definition we have yet no way for the compiler to resolve the correct type class for a given type and no means of expressing a dependency on a type class. For this we use the implicits feature discussed in Section 3.4.9. We can make a type class instance available to the compiler by creating an implicit definition. A good place for this is the companion object of either the data type or the trait defining our type class. As implicit definitions are searched in the companion objects of all parts of the type of the implicit parameter this is sufficient. Should the need arise to override the behaviour of a certain type class the implicit definition that should be used instead can be explicitly imported. Another common use case is to define operations for data types that the programmer has no control over. Type class instances can then be defined anywhere in the code and explicitly imported in the current scope. An explicitly imported type class instance always has higher priority than one defined in the implicit scope of a type and thus may also be used to override the operations in the current context.

```scala
// Companion object for point
object Point {
  implicit def eq: Eq[Point] = EqPoint
}

object EqTests {
  def testSymmetric[A](a: A, b: A)(implicit eq: Eq[A]) =
    !(eq.equal(a, b) ^ eq.equal(b, b))

  def testIdentity[A : Eq](a: A) = implicitly[Eq[A]].equal(a, a)
}
```

Listing 21: Implicit definition of type class instance

3.6. Type Hierarchy

Not only with regards to the next chapter it is reasonable to further inspect Scala's type hierarchy. As Scala tries to unify the type system it adds a common supertype for objects and primitive values. This super type is called `Any` and has two subtypes `AnyVal` and `AnyRef`. `AnyVal` is a common super type to all primitive values, namely `Int`, `Short`, `Long`, `Float`, `Double`, `Char` and `Unit`. It does not define any operations however as there is simply no meaningful way to do so. `AnyRef` is the common supertype of all class types.

In terms of type theory the type `Any` as super type of all of Scala's types is called the *top type*, denoted $\top$. We have already seen the type `Nothing` in the definition of the `List` data type in example 3.4.1. This type is called *bottom type*, denoted $\bot$ and is a subtype of all types in the Scala type system. It is impossible to create an instance of this type as it simply is not possible to define an instance that satisfies all conditions implied by that.

A type is called *inhabited* whenever a value of that type exists, so in other words we say that the type `Nothing` is uninhabited. Every singleton Scala object has a singleton type associated that is singly inhabited, as exactly one instance of this type exists.

`Nothing` is not the only bottom type that exists. Scala also defines the type `Null` that is the bottom type of the `AnyRef` type hierarchy. This type is singly inhabited by the value `null` which leads to a problem when working with class types in Scala. Every function that defines a class type as return type can by the substitution rules just return `null`.

In theory this turns every total function into a partial function as this escape route is offered. For this reason it is reasonable to avoid using the value null in programs.

The reasons for having an uninhabited bottom type might not be clear to the reader immediately so we will have a look at some examples of proper uses of the type `Nothing`. Every expression that does not terminate normally can be typed as `Nothing`. This applies to both expressions that upon evaluation throw an exception as well as expressions that do not return at all. The use case that we have seen before is using `Nothing` as an argument for a covariant type parameter for a type describing the empty structure. Being a subtype of every other type it follows that this empty structure is a subtype of every type describing that structure. Operations for accessing members of this structure can never terminate and thus has the return type `Nothing`.

```
def forever(f: () => Any): Nothing = {
  f()
  forever(f)
}

def err(msg: String): Nothing = throw new IllegalArgumentException(msg)

trait Seq[+A] { def head: A }
object EmptySeq extends Seq[Nothing] {
  def head = throw new UnsupportedOperationException("Head on empty seq")
}
```

Listing 22: Bottom type as return value

3.7. The Category of Scala Types

As mentioned in example 2.1.3 a functional programming language gives rise to a category. In this section we will inspect the properties of the category **Scala** with the Scala types as objects and the pure functions as morphisms.

- *Objects:* data types of the Scala language
- *Morphisms:* pure functions expressable in the Scala Language
- *Composition:* function composition
- *Identity:* the identity function returning its argument

As functions and composition are just a model of mathematical functions and mathematical function composition the required laws of associativity and unit hold.

3.7.1. Products in Scala

The category **Scala** has n-ary products. More exactly there are even two ways of defining products. First, there are the record types which clearly are products. The classes `Tuple1` through `Tuple22` define generic products of given arity. Custom products can be defined as case classes. In fact all case classes in Scala extend the trait `Product` that makes this even more clear.

There is another construct in Scala that can be used to construct products, namely subtyping combined with multiple inheritance. Given traits A, B, C the product $A \times B \times C$ can be defined by creating a new type that extends these traits. This way of defining products will not be considered in this thesis as it introduces various problems.

3.7.2. Coproducts in Scala

Categorical coproducts are often implemented so called sum types in programming languages. Scala offers no direct support for sum types but it is possible to use its object oriented features for implementing something similar. The standard technique to implement coproducts is to create a common super trait for all choices. This trait is marked as `sealed` so that it cannot be extended outside the file that defines the type itself.

```
sealed trait Tree[+A]
case class Node[A](left: Tree[A], right: Tree[A]) Tree[A]
case class Leaf[A](value: A) extends Tree[A]
```

Listing 23: Tree data type as sealed class hierarchy

There is a generalized form for binary coproducts of the shape $A + B$ in Scala's standard library called `Either[+A,+B]`. `Either` is implemented as shown above with the two subtypes `Left[+A,+B]` and `Right[+A,+B]` that represent the both possible cases. Each of this classes is implemented as a single parameter case class that can store a value. This type is often used as a result of computations that might fail and provide an error message or a value describing the reason for the failure. By convention a value wrapped in `Left` describes the case of failure while a value inside a `Right` represents the result of the computation.

Another commonly needed type is representable by the coproduct $1 + A$. This type, called `Option[+A]` in Scala describes an optional value. Using it as a return type for a function clearly marks it as a partial function.

3.7.3. Endofunctors in Scala

Both covariant and contravariant endofunctors can be expressed in the Scala Language.

For defining a functor we need a mapping from objects to objects, i.e. from Scala types to Scala types and a mapping from morphisms to morphisms, i.e. Scala functions to Scala functions.

According to the definition of a functor given in Section 2.3 a mapping of objects $F(A) \longrightarrow F(B)$. This can be expressed in Scala by using a type constructor and applying it to the given types. The actual action that will be performed is given a function $f : A \to B$ we create the function $F(f) : F(A) \longrightarrow F(B)$. So we have to provide a function or method with the signature `(A => B) => (F[A] => F[B])`. For practical reasons we rewrite this to a method and define that in the trait `Functor` given in Listing 24. In addition to implementing this interface the implementor has to ensure that functor laws hold.

```
trait Functor[F[_]] {
  def fmap[A,B](fa: F[A], f: A => B): F[B]
}

trait ContravariantFunctor[F[_]] {
  def comap[A,B](fa: F[A], f: B => A): F[B]
}
```

Listing 24: Functors as traits

It is not immediately clear from the code how the methods implement mapping of morphisms. The signature `(A => B) => (F[A] => F[B])` describes a function that given a function `A => B` returns a function `F[A] => F[B]`. This has some performance implications as every call to this function would lead to a new function object being allocated. The signature defined in the `Functor` trait above is isomorphic to this signature though. The techniques used to transform the signature are called currying and uncurrying respectively. Currying transforms a function taking multiple arguments as in $(A \times B) \to C$ into a chain of functions taking one argument each - in this case $A \to B \to C$. Uncurrying is the reverse transformation.

We can define the following isomorphic representations of the given siganture:

$$\begin{aligned} & (A \longrightarrow B) \longrightarrow (F\,(A) \longrightarrow F\,(B)) \\ \equiv\ & (A \longrightarrow B) \longrightarrow F\,(A) \longrightarrow F\,(B) \\ \equiv\ & ((A \longrightarrow B) \times F\,(A)) \longrightarrow F\,(B) \end{aligned}$$

In the second line currying was applied to the function on the right hand side of the signature. In the third line uncurrying is applied to the first two functions in the chain. After changing the order of the arguments this signature is identical to the signature given in the code example.

Having defined the interface we will now have a look at possible implementations. Most collection types can be treated as a functor. In fact the data type `Set` models the power-set functor $\mathcal{P}$ from example 2.7.2.

Example 3.7.1. The type constructor `List` as most collection type constructors has a covariant functor associated with it. The implementation of `fmap` will in this case generate a new list by applying the given function to each element of the source list. Applying it to the identity function will build the same list thus mapping it to the identity function for the given list type. It also preserves composition as building an intermediate list by applying each element to the first function before building the result list by applying the second function yields the same result. This obviously satisfies both required functor laws.

3.7.4. Natural Transformations in Scala

Section 2.5 states two different definitions of natural transformations. For the implementation in Scala we will treat them as a family of morphisms between the objects constructed by a functor. The functors kinds are fixed for a natural transformation and thus can be added as type parameters to a trait. We need to be able to universally quantify over the type parameter passed to the type constructor described by these kinds. This means nothing more that the method implementing the natural transformation has a type parameter.

```
trait Natural[F[_],G[_]] {
  def apply[A](fa: F[A]): F[A]
}
```

Listing 25: Natural transformations in Scala

Example 3.7.2. Considering all sequence types in Scala, like for example `List`, `Vector` and `Array` there exists a natural isomorphism between each two of them. This is because the structure represented by each of the implementations just refers to a sequence of elements.

Example 3.7.3. Lifting a value inside a functor is a natural transformation from the identity functor to the concrete functor. Given the identity type constructor `type Id[A] = A` it is possible to define an instance of this type class that wraps the element given in a data structure like a list, a set or for example in `Some` to create an `Option` instance from the value.

4. Abstract Petri Nets

4.1. Petri Nets

Petri nets offer both a mathematical model as well as a graphical model for describing and analyzing processes. The graphical representation of A Petri net is a directed, bipartite graph. The nodes represent either *transitions*, denoted as squares or *places*, denoted as circles. Based on the original model that is now known as *elementary nets* there are various extensions. This section will introduce two the concrete instances of Petri nets and their respective algebraic representation.

4.1.1. Elementary Nets

Generally an elementary net is defined as a triple $N = (P, T, F)$ with

- the set of places P
- the set of transitions T
- $F \subseteq (P \times T) \cup (T \times P)$ called the flow relation

The set P of places and the set T of transitions are disjoint. Every element $x \in P \cup T$ has an associated pre-domain ${}^\bullet x$ and post-domain $x^\bullet$. Graphically the pre domain denotes the nodes from which there is an incoming connection, the post domain the nodes to which there is an outgoing arc respectively. For every set of elements $X \in \{P, T\}$ there exist sets

$${}^\bullet X = \bigcup_{x \in X} {}^\bullet x \text{ and } X^\bullet = \bigcup_{x \in X} x^\bullet$$

For each elementary net N a marking $M \subseteq P$ can be given. A transition $t \in T$ is called enabled under the marking M, written $M\,[t\rangle$ if ${}^\bullet t \subseteq M$ and $t^\bullet \cap M = \emptyset$. The follower marking M' that results from firing t, denoted $M\,[t\rangle\,M'$ is defined as $M' = (M \backslash {}^\bullet t) \cup t^\bullet$.

Petri nets also allow parallel firing of multiple transitions. A set $U \subseteq T$ of transitions is enabled if ${}^\bullet U \subseteq M$ and $U^\bullet \cap M = \emptyset$.

An alternative definition of an elementary net in universal algebraic representation is given by $N = (P, T, pre, post)$, where

- the set of places P
- the set of transitions T
- the function $pre \ : \ T \to \mathcal{P}(P)$, which describes the *pre-domain* of the transitions
- the function $post \ : \ T \to \mathcal{P}(P)$, which describes the *post-domain* of the transitions

This definition does not explicitly include the pre and post domain of places. It is an isomorphic representation as the flow relation can be reconstructed by

$$F = \bigcup_{t \in T} \{(p,t) \mid p \in pre(t)\} \cup \{(t,p) \mid p \in post(t)\}$$

Conversely the pre and post domain can be extracted from the flow relation in the following way:

$$\begin{aligned} pre(t) &= \{p \mid (p,t) \in F\} \\ post(t) &= \{p \mid (t,p) \in F\} \end{aligned}$$

Example 4.1.1. The elementary net N in Figure 4.1 can be described as follows:

$$\begin{aligned} T &= \{t_1\} \\ P &= \{p_1, p_2, p_3, p_4, p_5, p_6\} \\ pre(t_1) &= \{p_1, p_2, p_3\} \\ post(t_1) &= \{p_4, p_5, p_6\} \\ m &= \{p_1, p_2, p_3\} \end{aligned}$$

The transition t_1 is enabled under marking m with follower marking $m' = \{p_4, p_5, p_6\}$, as in $m \,[t_1\rangle\, m'$.

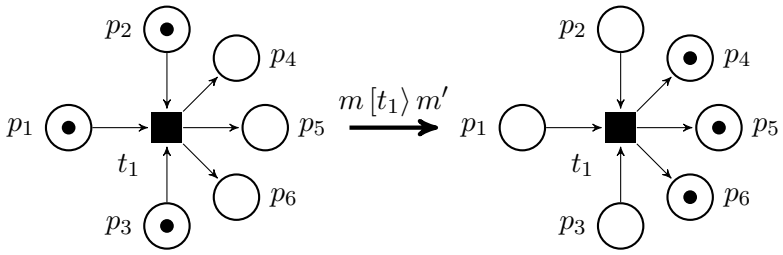


Figure 4.1.: A firing rule for an Elementary net

The class of elementary nets together with elementary net morphisms form a category. Given elementary nets $N_i = (P_i, T_i, pre_i, post_i)\,,\ i \in \{1, 2\}$ an elementary net morphism $f = (f_P, f_T) : N_1 \to N_2$ is a pair of functions $f_T : T_1 \to T_2$, $f_P : P_1 \to P2$, such that the following diagram commutes:

$$\begin{array}{ccc} T_1 & \overset{pre_1}{\underset{post_1}{\rightrightarrows}} & \mathcal{P}(P_1) \\ {\scriptstyle f_T}\downarrow & & \downarrow{\scriptstyle \mathcal{P}(f_P)} \\ T_2 & \overset{pre_1}{\underset{post_1}{\rightrightarrows}} & \mathcal{P}(P_2) \end{array}$$

The category $\mathbf{EN} = (\mathcal{O}_{\mathbf{EN}}, \hom_{\mathbf{EN}}, \circ_{\mathbf{EN}}, id_N)$ is given by

- $\mathcal{O}_{\mathbf{EN}} = \{N \mid N \text{ is an elementary net}\}$, the class of elementary nets
- $\hom_{\mathbf{EN}}$ the family of sets of elementary net morphisms with elements

$$\hom_{\mathbf{EN}}(N_1, N_2) = \{f\ :\ N_1 \to N_2 \mid f \text{ is an elementary net morphism}\}$$

- the composition $\circ_{\mathbf{EN}}$ defined as

$$\forall f\ :\ N_1 \to N_2,\ g\ :\ N_2 \to N_3 :\ (g \circ_{\mathbf{EN}} f)(x) = g(f(x))$$

- the identity $id_N = (id_P, id_T)$, defined componentwise on P and T for an elementary net $N = (P, T, pre, post)$

4.1.2. Place/Transition Nets

Place/transition nets are an extension of elementary nets in that they allow multiple tokens to be present in a place and add weights to the arcs. The arc weights denote

the minimal number of tokens that have to be present on the connected place in the pre-domain for the transition to be enabled. When a transition is fired the respective number of tokens is removed from the places in the pre-domain. The weights on the outgoing arcs denote how many tokens will be placed in the connected place in a firing step. Another view at the relation between elementary and place/transition nets is that elementary nets are a special case of place/transition nets where the arc weights are always one.

A place/transition net is given by $N = (P, T, pre, post)$ consisting of

- the set of places P
- the set of transitions T
- the function $pre : T \to P^{\oplus}$, which describes the *pre-domain* of the transitions
- the function $post : T \to P^{\oplus}$, which describes the *post-domain* of the transitions

where $P^{\oplus}$ denotes the free commutative monoid over the set P.

As for elementary nets morphisms between nets $N_i\,(P_i, T_i, pre_i, post_i)\,,\ i \in \{1, 2\}$ are defined as a pair of functions (f_P, f_T) with $f_P : P_1 \to P_2$ and $f_T : T_1 \to T_2$ such that the following diagram commutes:

$$\begin{array}{ccc} T_1 & \overset{pre_1}{\underset{post_1}{\rightrightarrows}} & P_1^{\oplus} \\ {\scriptstyle f_T}\downarrow & & \downarrow{\scriptstyle f_P^{\oplus}} \\ T_2 & \overset{pre_1}{\underset{post_1}{\rightrightarrows}} & P_2^{\oplus} \end{array}$$

The obvious difference to the definition of elementary nets is that every occurrence of the power set functor $\mathcal{P}$ is replaced with the free commutative monoid.

Example 4.1.2. Consider the place/transition net $N = (P, T, pre, post)$ given in Figure 4.2 with two markings $m_0,\ m_1$.

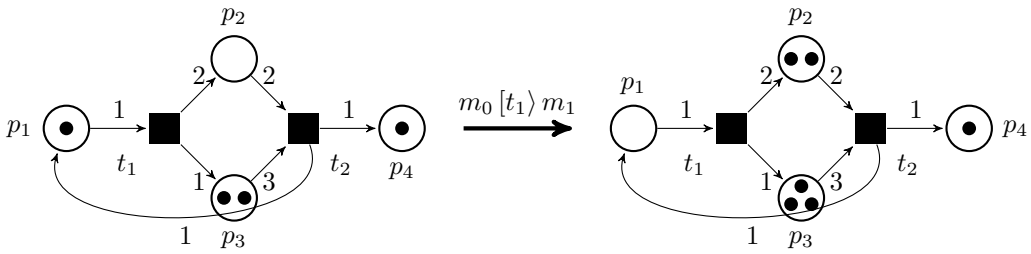


Figure 4.2.: A place/transition net

This net is described by

$$T = \{t_1, t_2\} \qquad P = \{p_1, p_2, p_3, p_4\}$$
$$pre\,(t_1) = p_1 \qquad pre\,(t_2) = 2p_2 + 3p_3$$
$$post\,(t_1) = 2p_2 + p_3 \qquad post\,(t_2) = p_1 + p_4$$

and the markings

$$m_0 = p_1 + 2p_3 + p_4 \qquad m_1 = 2p_2 + 3p_3 + p_4$$

4.2. Low-Level Abstract Petri Nets

Abstract Petri Nets(Padberg, 1996) are an effort to provide a unified view at different kinds of Petri nets. Abstract Petri Nets are an effort to provide a uniform approach to Petri nets. The notion of Abstract Petri Nets offers a universal description of the concepts and structures present in the theory of Petri nets. Results achieved on the conceptual level of Abstract Petri Nets can be instantiated on many kinds of net classes. The algebraic descriptions of both kinds of nets presented in the last section shared some resemblance and this section will illustrate how this relates to Abstract Petri Nets. This section will summarize the findings in the area of Abstract Petri Nets related to low-level net classes.

The structure of a net is determined by an adjoint situation **Sets** : $F \dashv G$: **Struct**. The endofunctor $Net = G \circ F$: **Set** $\rightarrow$ **Set** resulting from the composition of both functors is called *net structure functor*. Furthermore the category $Struct$, called *category of structure* is restricted to being a subcategory of the category **CSGroup** of commutative semigroups.

A low-level Abstract Petri Net is $N = (P, T, pre, post)$ is given as

- the set of places P,
- the set of transitions T,
- the function $pre: T \to Net(P)$ called precondition and
- the function $post: T \to Net(P)$ called postcondition.

Every morphism $f: T \to G \circ F(P)$ has a unique extension $\bar{f}: F(P) \to F(T)$. Given this it is possible to define marking, enabling and firing:

- The marking of an Abstract Petri net is given by $m \in F(P)$.
- A *transition vector* is defined by $v \in F(\{t\})$.
- $v \in F(\{t\})$ is enabled under $m \in F(P)$ if there exists a unique extension $\bar{m} \in F(P)$ such that $m = \overline{m} + \overline{pre}(v)$.
- A follower marking m' resulting from firing v under m is given as $m' = \overline{m} + \overline{post}(v)$.

Example 4.2.1. Given **Struct** $=$ **PSet** and the adjunction $F \dashv G$ as $\mathcal{P} \dashv I$ where $\mathcal{P}$ is the power set functor as in Example 2.3.1 and I denotes the identity functor. The net structure functor is given as $Net = I \circ \mathcal{P} = \mathcal{P}$ leading to the definitions of elementary nets as

$$E = (T \underset{post}{\overset{pre}{\rightrightarrows}} \mathcal{P})$$

This definition of elementary nets is referred to as *unsafe elementary nets* as this definition does not account for tokens in the post-domain. *Contextual elementary nets* solve this problem by defining the net structure as $\mathcal{P}_d \dashv Id$. $\mathcal{P}_d$ is the power set functor with distinct union that defines the union of a two sets as empty when they are not disjoint. This makes it impossible to assign two tokens to the same place in the post-domain.

Example 4.2.2. Place/transition nets as defined in Subsection 4.1.2 can be expressed in terms of Abstract Petri Nets. Let **Struct** $=$ **CMon**, the category of commutative monoids together with the adjunction **Set** $: (\_)^{\oplus} \dashv U :$ **CMon**, where $(\_)^{\oplus}$ is the functor that sends every set X to the free commutative monoid (see A.4). $X^{\oplus}$ and U denotes the forgetful functor assigning each structure to its underlying set.

4.3. High-Level Petri Nets

High-level Petri nets are an enhancement to low-level Petri nets in that they add structure to the tokens. They can add the notion of types and provide the means to express the enabling of a transition dependent on the tokens in its pre-domain. Two examples of high-level Petri nets are *algebraic high-level nets* ((Ramin and Kolagari, 2002)) and *coloured Petri nets* ((Jensen, 1991)). Coloured Petri nets define *colour sets* that act as a type for the tokens. The programming language ML is used to encode specifications in coloured Petri nets. In this section we will give an overview of algebraic high-level nets that take a similar approach but base it on algebraic specifications.

4.3.1. Typed Algebraic High-Level Nets

A typed algebraic high-level net $N = (SPEC, P, T, pre, post, cond, type, A)$ is given by

1. $SPEC = (S, OP, E, X)$, an algebraic specification with equations E and a family of variables X over the signature (S, OP),
2. a set of places P,
3. a set of transition T,
4. the pre-domain of the transitions $pre : T \to (T_{OP}(X) \otimes P)^{\oplus}$,
5. the post-domain of the transitions $post : T \to (T_{OP}(X) \otimes P)^{\oplus}$,
6. a function $cond : \mathcal{P}_{fin}(Eqns(S, OP, X))$, assigning to each transition $t \in T$ a finite set $cond(t)$ of equations over the signature (S, OP) with a family of sets of variables X,
7. a function $type : P \to S$, assigning to each place $p \in P$ a sort $type(p) \in S$, and
8. a $SPEC$-algebra $A = \left((A_s)_{s \in S}, (f_A)_{f \in OP}\right)$.

$T_O P(X)$ is the set of terms with variables X.
Given a typed algebraic high-level net (Ramin and Kolagari, 2002)

$$N = (SPEC, P, T, pre, post, cond, type, A),$$

a marking of N is an element of

$$PV = \left(\left(\biguplus_{s \in S} A_s \right) \otimes P \right)^{\oplus}$$

with $\left(\left(\biguplus_{s \in S} A_s \right) \otimes P \right) = \left\{ (a, p) \mid a \in A_{type(p)},\, p \in P \right\}$ and $\biguplus_{s \in S}$ is the disjoint union.

Firing a transition t is only possible when its related variables are assigned in a way that the conditions with respect to t are satisfied. This is expressed in the *consistent transition agreements*. The set of consistent transition agreements is defined as

$$CT = \left\{ (t, asg) \mid t \in T, asg : Var(t) \rightarrow \biguplus_{s \in S} A_S \right\},$$

such that the equations $cond(t)$ are satisfied by data elements in $\uplus_{s \in S} A_S$ in the assignment asg. These transition assignments are used to define functions that provide the data items consumed in the pre-domain and produced in the post domain.

Example 4.3.1. The algebraic high-level shown in Figure 4.3 (Ramin and Kolagari, 2002) models a reader-writer net with an arbitrary set of reading and writing processes. The specification $RW - SPEC$, set of variables X and algebra A are given as

$$\begin{aligned} RW - SPEC = \quad & sorts: \quad nat, process, type \\ & opns: \quad succ : nat \rightarrow nat \\ & \qquad\quad 0 :\rightarrow nat \\ & \qquad\quad w :\rightarrow type \\ & \qquad\quad r :\rightarrow type \\ & \qquad\quad p_{type} : process \rightarrow type \end{aligned}$$

$$\begin{aligned} X &= \{n : nat, p : process\} \\ A &= (\mathbb{N}, \mathbb{N} \times \{rs, wr\}, \{rs, wr\}, succ_A, 0_A, w_A, r_A) \end{aligned}$$

with

$$
\begin{aligned}
succ_A(x) &= x + 1 \\
r_A &:= rs \\
w_A &:= wr \\
0_A &:= 0 \in \mathbb{N} \\
p_{type_A} &= \begin{cases} rs & p = (n, rs), \\ wr & \text{else.} \end{cases}
\end{aligned}
$$

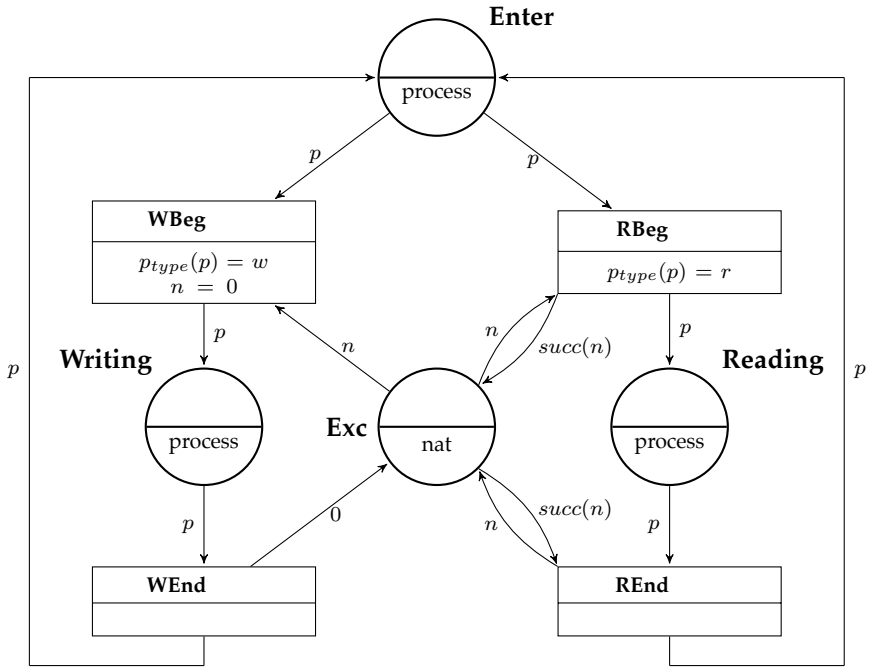


Figure 4.3.: Reader-writer net

Algebraic high-level nets together with algebraic high-level net morphisms form the category **AHL**.

4.4. Transformation of Petri Nets

In this section we will introduce two distinct notions of transformations possible on Petri nets.

4.4.1. Net Class Transformations

Every net class gives rise to a category. Transformations between net classes can be expressed as functors.

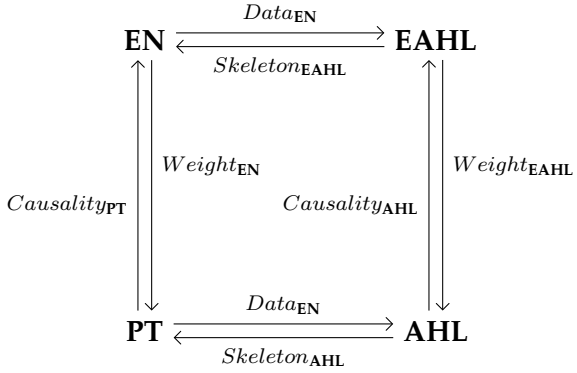


Figure 4.4.: Functors between categories of net classes

One possible view at the relation between categories **EN** and **PT** was that every elementary net is a place/transition net with arc weights of one. This relation is expressed as the functor $Weight_{\mathbf{EN}}$ that is defined as follows:

Given an elementary net $N = (P_N, T_N, pre_N, post_N)$, $Weight_{\mathbf{EN}}(N) = N'$ is defined as the place/transition net $N' = (P_{N'}, T_{N'}, pre_{N'} post_{N'})$ with

- $P_{N'} = P_N$,
- $T_{N'} = T_N$,
- $pre_{N'}(t) = \bigoplus_{p \in pre_N(t)} p, \quad t \in T_{N'}$, and
- $post_{N'}(t) = \bigoplus_{p \in post_N(t)} p, \quad t \in T_{N'}$.

The mapping of an elementary net morphism $f = (f_P, f_T) : N_1 \to N_2$ between elementary nets $N_1, N2$, to a place/transition net morphism f' is given by $Weight_{\mathbf{EN}}(f) = f' : Weight_{\mathbf{EN}}(N_1) \to Weight_{\mathbf{EN}}(N_2)$ with

$$f'_P(p) = f_P(p) \text{ for all } p \in P_{N'_1}$$
$$f'_T(t) = f_T(t) \text{ for all } t \in T_{N'_1}.$$

The functor $Causality_{\mathbf{PT}} : \mathbf{PT} \to \mathbf{EN}$ is defined in the opposite direction. Given a place/transition net $N = (P_N, T_N, pre_n, post_n)$ the elementary net $N' = Causality_{\mathbf{PT}}(N)$ is given by $N' = (P_{N'}, T_{N'}, pre_{N'}, post_{N'})$ with

- $P_{N'} = P_N$,
- $T_{N'} = T_N$,
- $pre_{N'}(t) = \{p \in P_{N'} \mid p \leq pre_N(t)\}, \quad t \in T_{N'}$, and
- $post_{N'}(t) = \{p \in P_{N'} \mid p \leq post_N(t)\}, \quad t \in T_{N'}$.

While both functors $Weight_{\mathbf{EN}}$ and $Causality_{\mathbf{PN}}$ are defined between categories **EN** and **PT** with opposite direction it is important to note that there is no adjoint situation involving both functors.

The functors describing transformations involving the high-level net classes will not be discussed in detail and are just mentioned for the sake of completeness. The category **EAHL** of elementary algebraic high-level nets was not introduced in this thesis. It can be considered the high-level equivalent of elementary nets in that the net structure is given by the powerset functor instead of the free commutative monoid as for algebraic high-level nets and place/transition nets. The functors $Data_{\mathbf{EN}}$ and $Data_{\mathbf{PT}}$ assign a single data type to all of the tokens to transform a low-level net into a high-level net. The functors in the opposite direction - $Skeleton_{\mathbf{EAHL}}$ and $Skeleton_{\mathbf{AHL}}$ - discard data types, operations and equations thus transforming a high-level net into a low-level net. The functors $Weight_{\mathbf{EAHL}}$ and $Causality_{\mathbf{AHL}}$ represent the high-level equivalents of the functors introduced in this subsection.

4.4.2. Petri Net Transformations Based on Morphisms

In this subsection we will inspect the transformations of Petri nets inside a net class. While there exist more advanced approaches to transformations of Abstract Petri Nets like rule based refinement (Padberg, 1996), this thesis will take a more basic approach in using the net morphisms to construct new nets. We will present a representation of the net morphisms that is more easily transferable to functional programming and can act as a foundation for implementing more advanced transformation rule systems on top of it.

The problems with the algebraic representation in Section 4.2 becomes obvious when some of the possible transformations of Petri nets are defined in terms of morphisms.

As a morphism is defined as a pair of functions (f_P, f_T) that maps one place to one place and one transition to one transition it is not possible to derive a function on this to delete places or transitions from a net or to insert places or transitions into a net. The mathematical definition for this is sound because the absence of a morphism is valid.

For the morphisms to be applied in a constructive way we define the morphism as a bijective map on the power sets of the places and transitions.

$$\tilde{f}_P : \mathcal{P}(P_1) \to \mathcal{P}(P_2)$$

Then transformations can then be expressed in the following way:

$$\begin{aligned}
\tilde{f}(\{p\}) &= \emptyset && \text{deletion of place } p \in P_1 \\
\tilde{f}(\emptyset) &= \{p, q, r\} && \text{insertion of places } p, q, r \in P_2 \\
\tilde{f}(\{p, q\}) &= \{r\} && \text{multiple places } p, q \in P_1 \text{ to one place } r \in P_2 \\
\tilde{f}(\{p\}) &= \{q\} && \text{one place } p \in P_1 \text{ to one place } q \in P_2 \\
\tilde{f}(\{p\}) &= \{q, r\} && \text{one place } p \in P_1 \text{ to several places } q, r \in P_2
\end{aligned}$$

The nonexistence of a morphism is expressed by the equations that include the empty set by assigning the empty domain or codomain respectively. The practical use of this approach is still limited as for an evaluation of the function requires generation of all 2^n subsets for a set of length n. Avoiding the generation of the subsets can be achieved by associating the images of the sets to the elements of the sets.

$$\begin{aligned}
g_X &: X \to \mathcal{P}(Y) & g_X(x) &= \{\alpha \in \mathcal{P}(X) \mid x \in \alpha\} \\
\tilde{f}'_X &: X \to \mathcal{P}(Y) & \tilde{f}'_X(x) &= \bigcup_{\alpha \in g_X(x)} \tilde{f}_X(\alpha)
\end{aligned}$$

Given these functions together with a set α_0 that represents the elements inserted it is possible to implement transformations of Petri nets that support deletion and addition of places purely based on morphisms.

5. Design and Implementation

5.1. Type Classes

In this section we model our software library using type classes. Some of the type classes that will be used in our model have already been mentioned in the examples in previous chapters but we will reiterate them here to also point out their respective use in the design. Most of the type classes and their instances are already implemented in a Scala Library called Scalaz. Scalaz is heavily inspired by Haskell and the type classes present in its standard library. It also includes some concepts that are beyond the scope of this thesis and as such this chapter will provide a simplified view on the type classes.

We have seen the definition of the `Functor` type class in Section 3.7.3. As mentioned in that section the implementor of a `Functor` instance has to make sure that the laws for functors hold for the implementation provided.

Example 5.1.1. The identity endofunctor can be implemented as a type class instance. The type part sending each object to itself can be expressed as the identity type constructor described in 3.7.3. As any morphisms is also mapped to itself the implementation of `fmap` will just apply the supplied function to the given value parameter. Proving that the functor laws hold is trivial as composition is effectively unchanged and thus just follows the general rules.

```
object IdFunctor extends Functor[({type λ[α]=α})#λ] {
  def fmap[A,B](a: A, f: A => B) = f(a)
}
```

Listing 26: Identity functor in Scala

In 3.7.4 we have seen how we can define type classes for natural transformations. One of the most commonly used natural transformation is a natural transformation of the shape $1 \Longrightarrow F$ where F represents some higher kinded type. In fact it is so common

that many type class libraries provide a distinguished type class for that called *pure* or sometimes also *point*.

```
trait Pure[F[_]] {
  def pure[A](a: A): F[A]
}
```

Listing 27: Pure type class

Another commonly used type class is the `Flatten` type class. Technically this refers to a natural transformation $T^2 \Longrightarrow T$ just like the monad multiplication in Section 2.8.

```
trait Flatten[F[_]] {
  def flatten[A](tt: F[F[A]]): F[A]
}
```

In Section 2.8 we have seen that some additional properties of adjoint situations can be expressed by the fact that every adjunction has a monad associated with it. A monad (T, η, μ) can also be expressed as a type class. For any kind `T[_]` for which type class instances of types `Functor[T]`, `Flatten[T]` and `Point[T]` exist we have all it takes to define the monad operations. As computing resources are limited the problem with this approach is building an entire structure in memory just for flattening it afterwards. To avoid this we will also provide a type class for an operation called *monadic binding* named `Bind` and express the monad laws in the following way(Wadler, 1992, 2.10):

$$\texttt{f(a)} == \texttt{bind(pure(a), f)} \qquad \forall \texttt{a, f}$$

$$\texttt{a} == \texttt{bind(a, x => pure(x))} \qquad \forall \texttt{a}$$

$$\texttt{bind(a, x => bind(f(x), g))} == \texttt{bind(bind(a, f), g)} \qquad \forall \texttt{a, f, g}$$

As all operations required by the monad type class are already defined in other type classes we can write a generic method constructing a monad instance for a given higher kinded type. We express the dependencies on other type classes as context bounds which the compiler will transform into an implicit parameter declaration for the method. The `flatten` operation can be expressed by means of the `bind` operation and the `bind` operation can be implemented using a `Functor` instance and the `flatten` operation. This leads to two possible factory methods for a monad instance inside of the `Monad` companion object.

```
def fromFunctorPureBind[T[_] : Functor : Pure : Bind]: Monad[T] =
  new Monad[T] {
    val pureInstance: Pure[T] = implicitly
    val functorInstance: Functor[T] = implicitly
    val bindInstance: Bind[T] = implicitly

    def fmap[A,B](t: T[A], f: A => B) = functorInstance.fmap(t, f)
    def pure[A](a: A) = pureInstance.pure(a)
    def bind[A,B](t: T[A], f: A => T[B]) = bindInstance.bind(t, f)
    def flatten[A](t: T[T[A]]) = bind(t, identity)
  }

def fromFunctorPureFlatten[T[_] : Functor : Pure : Flatten] : Monad[T] =
  new Monad[T] {
    val pureInstance: Pure[T] = implicitly
    val functorInstance: Functor[T] = implicitly
    val flattenInstance: Flatten[T] = implicitly

    def fmap[A,B](t: T[A], f: A => B) = functorInstance.fmap(t, f)
    def pure[A](a: A) = pureInstance.pure(a)
    def bind[A,B](t: T[A], f: A => T[B]) = flatten(fmap(t, f))
    def flatten[A](t: T[T[A]]) = flattenInstance.flatten(t)
  }
```

Listing 28: Construction of monad instances from other type classes

5.1.1. Type Classes as Evidence

Type classes can not only be used for defining operations but also to provide evidence for a certain property of a type. One interesting application is to encode axioms for types in types. An important notion in category theory in general is the notion of an isomorphism. Listing 29 shows the type class definition for such an isomorphism and an example application.

```
trait Iso[A,B] {
  def apply(a: A): B
  def reverse: Iso[B,A]
}

object Iso {
  implicit def idIso[A]: Iso[A,A] = new Iso {
    def apply(a: A) = a
    def reverse = this
  }
}

def applyEndo[A,B](a: A, f: B => B)(
    implicit iso: Iso[A,B]
  ): A =
    iso.reverse(f(iso(a)))
```

Listing 29: Type class for expressing isomorphisms

A natural isomorphism can be encoded in a similar way. The two type parameters are replaced with two higher kinded types and exactly as implemented in the `NaturalTransformation` trait a type parameter is added to the `apply` method. The natural isomorphism between data types `List` and `Vector` mentioned in Example 3.7.2 is given as a type class instance in Listing 30.

```
implicit val listVectorIso: NatIso[List,Vector] =
  new NatIso[List,Vector] { outer =>
    def apply[A](xs: List[A]) = Vector.empty ++ xs
    val reverse = new NatIso {
      def apply[A](xs: Vector[A]) = xs.toList
      def reverse = outer
    }
  }
```

Listing 30: Natural isomorphism between sequences

5.1.2. Net Structure as Data Type

Section 3.7.3 showed that endofunctors can be defined for data structures like lists, sets and multisets. Thus it is important to first look at the relation between data structures in Scala and abstract structures in category theory.

The net structure functor is an endofunctor in **Set** that is given by composition $G \circ F$ of two adjoint functors $F \dashv G$: **Struct** $\longrightarrow$ **Set**. As mentioned in Section 3.7.3 F and G are not representable in Scala's type system but $G \circ F$ being an endofunctor is.

The covariant power-set functor is used for describing the structure of an Elementary Net (see Example 4.2.1). To implement a data type representing an Elementary Net in a Scala program we thus need to express the values of pre and post domain as a data type. For any set $A \in$ **Set**, $\mathcal{P}(A)$ is the set containing all subsets of A. The data type that represents the power set for a data type `A` is `Set[A]`.

5.2. Categorical View of Data Structures

An important part of the implementation is to express how some of the categorical concepts in the Scala Language. Our design will exploit the fact stated in example 2.7.2 that every category where the morphisms are sets can be expressed as a category where the objects are sets and the morphisms are functions between them. This means that every time we leave the category **Set** we have to ensure that the functions and types of our implementing code correctly reflect the situation expressed in the categorical concept.

We have also seen that the model for Abstract Petri Nets defines the net structure as a functor. In the definition of Abstract Petri Nets the net structure functor was defined as

$$Net = G \circ F : \textbf{Set} \longrightarrow \textbf{Set}$$

where $F \dashv G$: **Struct** $\longrightarrow$ **Set**, and **Struct** being a subcategory of the category **CSGroup** of commutative semigroups.

This immediately leads to the question how the concept of an adjunction relates to the types involved in the program. In a first step we will discard all additional requirements and just consider the free monoid from example 2.7.1. We can express an adjoint situation $F \dashv U$: **Mon** $\longrightarrow$ **Set** between the *free functor* F, sending each object in **Set** to a word of length one in **Mon** and the *forgetful functor* U sending the monoid to the un-

derlying set. This adjunction can be used to generate words of arbitrary length in the category of monoids and map the result to the underlying set by forgetting everything about the structure of the monoid.

In functional programming the data structure representing a word is a list, so we should be able to define

- an endofunctor F : **Scala** $\longrightarrow$ **Scala**,
- a natural transformation $1 \Longrightarrow F$,
- a corresponding monad.

The functor part has to map a `List[A]` to a `List[B]` given a function of type `A => B`. There is only one intuitive way to achieve that, that is by generating a new list by applying the function to each item of the list. Fortunately the Scala Collection Library defines this method for all collection types as a method called `map` that just takes the mentioned function. Writing the functor as a type class instance we end up with the code in Listing 31.

```
implicit def ListFunctor: Functor[List] = new Functor[List] {
  def fmap[A,B](t: List[A], f: A => B) = t map f
}
```

Listing 31: Functor type class instance for List type

Next we need a natural transformation that turns a single object into a list. Following our intuition again we assume that this can be no other operation than creating a one element list out of it. Having unit and functor part of a monad we now only lack the multiplication part. In our example we need an operation that turns a `List[List[A]]` into a `List[A]` for any given `A`. Here we will assume that we get the expected result by simply flattening the list. It is important to note that monads in functional programming are not only used for representing data structures but offer a general abstraction for representing computations. The monads related to data structures can be seen as a special case where the result of a computation is a data structure.

Until now the implementations are only assumptions so we now check if the functor and monad laws hold. For this we will have a look at what the functor laws actually mean when applied to this example. The first monad law states that $\mu \circ \mu_T = \mu \circ T\mu$. In terms of lists this means that having a nested list with depth of three we can flatten it

twice to get a flat list. The additional requirement expressed in this law is that the result is in the same list independent from whether one starts flattening from the outside or the inside.

The second monad law requires that $\mu \circ \eta_T = 1_T = \mu \circ T\eta$.

This in the context of lists describes the fact that wrapping a list in a list and flattening it again yields the input list itself and the same holds for constructing a nested list by wrapping each element individually and flattening that.

Expressed as code the laws could be written as follows:

```
def validateFirstLaw[A](xs: List[List[List[A]]]) =
  xs.flatten.flatten == xs.map(_.flatten).flatten

def validateSecondLaw[A](xs: List[A]) =
  List(xs).flatten == xs && xs.map(x => List(x)).flatten == xs
```

Listing 32: Monad laws for List

We can now define our type class instances for our list as in Listing 33. As `List` is defined in the Scala standard library and thus it is not possible to include the instance definitions in the companion object they are added to the `Functor` companion object. This way they are globally available to the compiler but may be overridden in the local scope when needed (see Subsection 3.4.9). In this example the type classes are implemented as anonymous inner classes. The type ascriptions are necessary in this case as otherwise the compiler infers a compound type making it harder to override the value (see Subsection 3.4.9). The monad instance can be defined in means of the other type classes using the method definitions from Listing 28.

```
implicit def ListPure: Pure[List] = new Pure[List] {
  def pure[A](x: A) = x :: Nil
}

implicit def ListFunctor: Functor[List] = new Functor[List] {
  def fmap[A,B](t: List[A], f: A => B) = t map f
}

implicit def ListFlatten: Flatten[List] = new Flatten[List] {
  def flatten[A](list: List[List[A]]) = list.flatten
}

implicit def ListBind: Bind[List] = new Bind[List] {
  def bind[A,B](list: List[A], f: A => List[B]) = list flatMap f
}

implicit def ListMonad = Monad.fromFunctorPureBind[List]
```

Listing 33: Type class instances for List

Having defined the required type class instances we can now provide a more generic version of our code to check the monad laws. Instead of operating on the concrete type `List[A]` it is possible to express the generals structure using a higher kinded type parameter `T[_]` and applying it to the type parameter `A` as seen in Listing 34.

```
  def validateFirstLaw[T[_],A](t: T[T[T[A]]])(implicit m: Monad[T]) =
    m.flatten(m.flatten(t)) == m.flatten(m.fmap(t, m.flatten)

  def validateSecondLaw[T[_],A](t: T[A])(implicit m: Monad[T]) =
    m.flatten(m.pure(t)) == m.flatten(m.fmap(t, m.pure))
```

Listing 34: Checking monad laws for type class

These checks can now be executed for every kind `T[_]` for which a monad instance is available to the compiler. However, the resulting code is not idiomatic Scala code. In Section 5.4 we will provide a syntax layer that allows using type class instances in a way that the resulting code is indistinguishable from ordinary object oriented Scala code.

The definition of a place/transition-net given in Subsection 4.1.2 defines the net structure via the free commutative monoid $P^{\oplus}$ over the places (see Subsection 4.2). For this

we have to provide a functorial definition of the net structure functor $Net = F \dashv G$ that describes the structure. The free commutative monoid can be modeled as a multiset. While it is possible to implement an optimized data structure we exploit the fact that the data type multiset over a set A is isomorphic to a map with keys of type A and as values non-negative integers. The difference between both are the operations defined for each. In Listing 15 a similar situation arose for the two monoids over the set of integers which differed only in the operations defined. Following the same pattern the multiset will be implemented as a wrapper class that defines the operations and acts as a type parameter for type class instances. An outline of the implementation is given in Listing 35. It defines the methods `map` and `flatMap` thus the type class instances are implemented similarly to the ones for `List` in Listing 33. In contrast to the type class instances for the `List` data type the instances for `Multiset` can be defined in the companion object of `Multiset` for making them available to the compiler.

```
final case class Multiset[A](values: Map[A,Int]) {
  def freq(a: A) = values.getOrElse(a, 0)

  def map[B](f: A => B): MultiSet[B] = {
    def elements = for {
      (x,n) <- values.iterator
      y = f(x)
    } yield (y,n)

    MultiSet(elements.foldLeft(Map.empty[B,Int]) {
      case (m,(x,n)) => m.updated(x, m.getOrElse(x, 0) + n)
    })
  }

  def flatMap[B](f: A => MultiSet[B]) = {
    def elements = for {
      (x,n) <- values.iterator
      (y,k) <- f(x).values.iterator
    } yield (y, n*k)

    MultiSet(elements.foldLeft(Map[B,Int]()) {
      case (m,(x,n)) =>
        m.updated(x, m.getOrElse(x, 0) + n)
    })
  }
  // ...
}
```

Listing 35: Example of a Multiset implementation

The operations defined above together with the monad unit that takes every value $a \in A$ to $a \in A^{\oplus}$ a monad instance can be defined. The monad laws hold for this representation. This follows immediately from the representation as linear sums. The free commutative monoid $A^{\oplus}$ over a set A can be represented as

$$A^{\oplus} = \sum_{a \in A} \lambda_a a.$$

Due to the associativity of sums the results are identical whether the sum is evaluated inside out or starting from the outside.

5.3. Data Types

Being a tuple, an Abstract Petri Net $N = (P, T, pre, post)$ can be expressed in Scala as a case class. In the design introduced in the following pages we will treat the data type describing a Petri net as a marked net that is the marking is also included in the data type. A first attempt to define the data type for a Petri net can be seen in Listing 36. Net structure and marking are given as higher kinded type parameters but there are yet no restriction imposed upon these concerning the existence of type class instances implementing the categorical concepts. It is important to note that two distinct higher kinded parameter for both net structure functor and for representation of markings are given. This is due to the fact that the marking of an Abstract Petri Net is given as $F(P)$ while the net structure functor is defined as $Net = G \circ F$. The functor F can not be expressed for all choices of functors as in many cases F will not be an endofunctor. It is possible that a second adjoint situation is needed for expressing the marking in Scala. One example that defines markings in terms of a different adjunction are constructions of *S-Graphs* based of Abstract Petri Nets (Padberg, 1996, Example 2.4.5).

```
case class Net[P,T,Net[_],Mark[_]](
  places: Set[P],
  transitions: Set[T],
  pre: T => Net[P],
  post: T => Net[P],
  marking: Mark[P]
)
```

Listing 36: Case class representing a Petri net

While in general we tried to avoid subtyping for ad-hoc polymorphisms in our code we will make use of it when defining our Net Classes. Traits can not only be viewed as a blueprint for classes but can also be used to define a structure of a module.

```
trait NetClass {
  type Place
  type Transition
  type Net[X]
  type Mark[X]

  case class PetriNet(
    places: Set[Place],
    transitions: Set[Transition],
    pre: Transition => Net[Place],
    post: Transition => Net[Place],
    marking: Mark[Place]
  )
}
```

Listing 37: Trait representing a net class

Using traits for the definition also allows composing traits that act as a feature of the net class. In Listing 38 two traits are presented that implement such features. The trait `Labeled` implements the feature of labeled places and transitions by providing implementations of the types representing these. The basic feature of an elementary net is implemented in the trait `Elementary` by defining the kinds representing the functors used for net structure and marking. A labeled elementary net is then defined by composing both traits with the net class trait in the object `LabeledElementaryNets`.

```
trait Labeled { self: NetClass =>
  case class Place(label: String)
  case class Transition(label: String)
}

trait Elementary { self: NetClass =>
  type Net[X] = Set[X]
  type Mark[X] = Set[X]
}

object LabeledElementaryNets extends NetClass
                                with Elementary
                                with Labeled
```

Listing 38: Modular composition of net class features

This implementation already determines the operations to be used as when provided via type classes they are dependent on the types and kinds involved. Changing the functor used to describe the net structure is done by changing the type members `Net` and `Mark` to represent the type constructor that is associated to the functor to use. Enabling of transitions and firing can be implemented by using the operations defined by the type classes.

The functors describing transformations between net classes described in Subsection 4.4.1 are not modeled as classes implementing the `Functor` trait as in this context they represent operations. Instead they can be implemented as functions working on the type constructors that define the net structure and marking. The functors are defined componentwise on the elements describing the Petri net and as such the implementation provides a way of specifying the transformation of every component. They are fully defined by the way they map the net structure and marking and as such are implemented as a morphism between these functors. While the interface to define these kinds of transformations is already given by the trait `NaturalTransformation` it has to be pointed out that this interface cannot be used in this context. The trait `NaturalTransformation` does not only define the interface on the language level but also encodes that instances of this trait can be used as a natural transformation. The transformation performed by the functors is not necessarily a natural transformation though.

Example 5.3.1. A valid natural transformation has to satisfy the naturality condition expressed in the naturality square in Figure 2.2. The naturality condition states that

$$\nu_{C'} \circ F(f) = G(f) \circ \nu_C.$$

An example for a transformation performed by a functor between net classes that acts as a counterexample is the functor $Weight_{\mathbf{EN}} : \mathbf{EN} \to \mathbf{PT}$. The transformation constructs a free commutative monoid from the power set for all pre-domains and post-domains thus defines a transformation $\mathcal{P} \to (\_)^{\oplus}$. This transformation is not natural. Consider the following example: Let $A = \{1, 2, 3\}, B = \{x\}$ and $f(a) = x,\ a \in A$. The left hand side of the naturality condition evaluates to $\nu_B \circ F(f) = x$ while the right hand side evaluates to $G(f) \circ \nu_C = f(1) \oplus f(2) \oplus f(3) = 3x$.

For this reason a second trait called `Arrow[F[_],G[_]]` is defined that represents a morphism between these functors but does not imply naturality. The transformation operations are then defined as in Listing 39. The definitions are given as methods for

brevity but in the actual code they are implemented inside the `apply`-method of the corresponding `Arrow`-implementation.

```
def weightEN[A](d: Set[A]): Multiset[A] =
  Multiset(d.map(x => (x,1)).toMap)
def causalityPT[A](d: Multiset[A]): Set[A] =
  d.values.entrySet
```

Listing 39: Implementations of net class transformations

When performing a net class transformation the type parameter `A` is fixed to the type representing the places. The functions representing pre-domain and post domain are constructed via function composition as in Listing 40. The `transform`-method has three parameter lists. The first parameter list takes the two objects representing the net classes. The second parameter list takes the Petri net to be transformed and the `Arrow` instances describing the transformations. As the type members act as path-dependent-types (see Subsection 3.4.3) they are accessible in the following parameter lists as members of the `NetClas` instances. The third parameter list takes two implicit parameters that act as an evidence that places and transitions are represented as the same type in both net classes. These instances are provided in the Scala library inside the `Predef`-object whose members are automatically imported. It is also safe to instead depend on the type class instances representing an isomorphism presented in Listing 29 and convert all instances of places and transitions to the isomorphic representation used in the other net class.

```
def transform(dom: NetClass, cod: NetClass)(
  net: dom.Petrinet,
  tn: Arrow[dom.Net,cod.Net],
  tm: Arrow[dom.Marking,cod.Marking]
)(implicit sp: (dom.Place =:= cod.Place),
           st: (dom.Transition =:= cod.Transition)
) = new cod.Petrinet(
    places = net.places,
    transitions = net.transitions,
    pre = net.pre compose (tn(_:dom.Place)),
    post = net.post compose (tn(_:dom.Place)),
    marking = tm(net.marking)
)
```

Listing 40: Performing a net class transformation

The transformations arising from endomorphisms are modeled based on the representation of morphisms in Subsection 4.4.2. The signature of the method implementing class of transformations has to include the following parameters:

- the net class description as an instance `nc` of type `NetClass`,
- the Petri net to be transformed as an instance of `nc.PetriNet`,
- the morphism to be used, given as two functions `fp: nc.Place => Set[nc.Place]`, `ft: nc.Transition => Set[nc.Transition]`,
- the values describing inserted nodes `p0: Set[nc.Place]`, `t0: Set[nc.Transition]`,
- an initial marking `m0: nc.Marking[nc.Place]` describing the marking of inserted places and
- pre-domain as well as post-domain values for the inserted elements as functions `pre` and `post` of type `nc.Transition => nc.Net[nc.Place]`.

The operations on the types and kinds defined in the net class are accessible via a type class passed as an implicit value. This value `ops` is an instance of a trait `NetClassOperations` that is introduced to simplify the methods signature and contains the required type class instances as members.

Pre-domains and post-domains are given as functions $pre, post : T_i \to Net(P_i),\ i \in \{1, 2\}$ and transitions are mapped via the part $f_t : T_1 \to T_2$ of the morphism. Creating

a function representing pre-domain and post-domain requires a function $\overline{f}_t : T_2 \to T_1$ so that the pre-domain of the resulting net is represented as $pre_2 = Net(f_p) \circ pre_1 \circ \overline{f}_t$ and the post-domain accordingly. This situation is expressed in the following diagram:

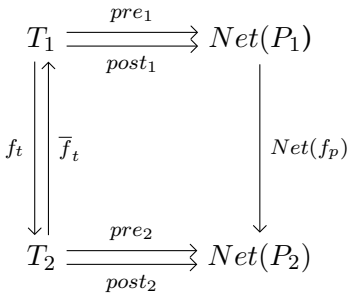


Figure 5.1.: Construction of pre-domain and post-domain

Exploiting the fact that the operations are defined on finite sets it is possible to generate a function representing $\overline{f}_t$ by generating pairs (v, t) with $t \in T_1$, $v \in \tilde{f}_t(t)$. A generic method implementing this will be called `invert`. It takes a parameter of type `Set[A]` and a function `A => Set[B]` as an argument and returns a function `B => Set[A]`.

```
def endo(nc: NetClass)(
  // parameters omitted – see description
)(implicit ops: NetClassOperations[nc.type]): nc.PetriNet = {
  import nc._, ops._

  val ftInv = invert(net.transitions, ft)

  def mapConditions(pDom: Transition => Net[Place],
                    pCod: Transition => Net[Place]) =
    { t: Transition =>
       netMonoid.append(netMonad.bind(includeNet(ftInv(t)), net.pre), pCod(t))
  }

  nc.PetriNet(
    places = p0 ++ net.places.flatMap(fp),
    transitions = t0 ++ net.transitions.flatMap(ft),
    pre = mapConditions(net.pre, pre0),
    post = mapConditions(net.post, post0),
    marking = markMonoid.append(
      bindMark.bind(net.marking, (p: Place) => includeMark(fp(p))),
      m0)
  )
}
```

Listing 41: Implementation of morphism-based transformations

Listing 41 shows the implementation in detail. It is important to note that the type parameter to the `NetClassOperations` trait is parameterized with the path-dependent type of the net class instance. This is due to the fact that the type members describing the net class are part of the instance and thus only accessible via the path. The function `mapConditions` is used to compute the functions representing pre-domain and post-domain of the resulting net. This is implemented by mapping the transition $t \in T_2$ to a set of transitions $v \subseteq T_1$. The set is transformed to an instance of the data type representing net structure using an instance of `Arrow[Set,Net]` and the pre-domain of the original net is computed and flattened. The monoid type class instance for the type `Net[Place]` is then used to add the pre-domain defined in the transformation to the resulting structure. Places and transitions are mapped by applying the functions f_p and f_t respectively to each element via the method `flatMap`. This method implements the monadic bind operation described in Section 5.1 and thus returns the union of all sets

that result from application of the function passed as an argument to the elements of the set. The marking of the resulting Petri net is also computed using the monadic bind operation. As the implementation of monadic binding depends on the kind representing the data structure, an instance of the type class `Bind[Marking]` is used to compute the transformed marking. The result of this operation is a marking that does not include places deleted in the transformation.

5.4. Syntax Layer

For now we have implemented several operations for our data types as type classes. Using the defined operations is currently still cumbersome as all operations are defined as methods on the type class instances. In this section we will introduce a pattern for making the use of these operation more natural in Scala.

Subsection 3.4.9 introduced the concept of implicit conversions and provided an example how the enrich-my-library pattern can be used to add operations to arbitrary data types.

Example 5.4.1. Making the operations defined on ordinary types (as opposed to kinds) available as members on instances is done by introducing a class `Syntax[A]` that wraps an instance of type `A`. The operations are defined in traits that are then composed to the final wrapper class as shown in Listing 42. These traits can be located inside the source code files of the respective type classes that implement the operations. The trait `Wrapped` carries the type of the wrapped instance and the reference to the wrapped instance. A call to the method `⊕` on an integer value triggers the implicit conversion to `Syntax[Int]`.

```
trait Wrapped {
  type Self
  val value: Self
}

trait MonoidSyntax { self: Wrapped =>
  def ⊕(that: Self)(implicit m: Monoid[Self]) =
    m.append(value, that)
}

trait PureSyntax { self: Wrapped =>
  def pure[M[_]](implicit p: Pure[M]) =
    p.pure(value)
}

class Syntax[A](val value: A) extends Wrapped
                              with PureSyntax
                              with MonoidSyntax {
  type Self = A
}

implicit def any2syntax[A](a: A) = new Syntax(a)

// use the monoid instance for integers via MonoidSyntax
val twenty = 10 ⊕ 10

// Wrap String in a List[String]
val foos = "foo".pure[List]
```

Listing 42: Syntax layer for types

Example 5.4.2. The enrich-my-library pattern can also be applied to types defined via a type constructor. The base trait `WrappedKind` has two type members, one higher kinded member referencing the type constructor and one type member that captures the type that was applied to the kind for constructing the type of the wrapped value. All traits in Listing 43 defining the operations follow the same pattern as in Example 5.4.1 but now have the type constructor available. The method `lengths` provides an example usage. A context bound is used to express the dependency for a functor instance for kind `M[_]` and inside the `map`-method is invoked on the parameter. As nothing is known about the type `M[A]` this will trigger the implicit conversion to `KindSyntax[M,A]`.

```
trait WrappedKind {
  type M[X]
  type A
  val value: M[A]
}

trait FunctorSyntax { self: WrappedKind =>
  def map[B](f: A => B)(implicit t: Functor[M]) =
    t.fmap(value, f)
}

trait MonadSyntax { self: WrappedKind =>
  def flatMap[B](f: A => M[B])(implicit m: Monad[M]) =
    m.bind(value, f)
}

class KindSyntax[F[_],T](val value: T)
  extends WrappedKind
     with MonadSyntax {
  type M[X] = F[X]
  type A = T
}

implicit def any2kindSyntax[M[_],A](ma: M[A]) =
  new KindSyntax[M,A](ma)

// Usage:
def lengths[M[_] : Functor](xs: M[String]) =
  xs.map(x => x.length)
```

Listing 43: Syntax layer for kinds

6. Conclusion and Prospects

6.1. Discussion

The goals stated in Subsection 1.1 have been met as shown subsequently:

1. In this thesis we showed how Scala's language features can be used to model categorical concepts. Especially when working with generic data structures the concepts are easily transferable between the categorical model describing abstract structures and the categorical model for describing functional programming languages. The separation of data types and operations provided by type classes together with the higher abstraction level over types and type constructors enables a modular design in which the behaviour of the program depends on the types used. Furthermore the behaviour is fully determined by the types as the operations are defined on types rather than instances. Referential transparency together with the grouping of type classes and the inability to modify behaviour of a single object by overriding the behaviour make the code more maintainable. The ability to also encode axioms in the type system proved helpful. Different operations with the same signature can not be misused when the implementor choses the right interface after evaluating whether the respective axioms hold for the provided implementation.

2. We also showed that it is possible to use the common foundation supplied by category theory to create formal abstractions that can be used to easily transfer concepts that exist in the problem domain into the solution domain. The formal model given by Abstract Petri Nets can be used as a foundation for a program describing different classes of Petri nets. The resulting framework is highly modular in that it can be easily extended to support various low-level net classes by providing the relevant type class instances that model the underlying mathematical concepts. These concepts are also generic enough for the resulting implementations to be of use in other fields.

3. Both transformations based on functors and transformations based on morphisms have been implemented. The definition of a functor between net classes can be easily given by implementing two generic methods that perform the actual mapping between net structure and marking of both nets. The transformations based on morphisms are implemented using only type classes that are used for defining the general structure and basic operations on the net and thus should be easily transferable to new low-level net classes. As this morphism-based transformation is very general it can be used as a foundation to implement the more advanced net transformations on top of it.

Some disadvantages are also to be mentioned. Operations depending on many type classes carry a long type signature. A solution to this is the grouping of type classes to yield another instance that publishes its members. On the other hand this makes the resulting library harder to use as the type class instances are hidden inside other objects.

6.2. Applicability to High-Level Nets

The presented software design is capable of representing low-level net classes. This subsection will present the concepts that can be used to extend the model to also support high-level petri nets and will illustrate the difficulties in extending the model.

In Subsection 4.3.1 the notion of typed algebraic high-level networks was introduced. There are numerous possible implementations but as one goal of this work was to maintain static type safety the approach taken in this subsection will try to leverage Scala's type system for a possible implementation.

The signature $SIG = (S, OP)$ consists of a set of sorts S and a set OP of constant and operational symbols. In this approach the sorts will be represented as Scala types and the operational symbols are expressed as values and pure methods and functions. By using algebraic data types as introduced in section 3.4.6 to represent the type part of the signature it is possible to restrict the data types of the net.

As places of high-level petri nets are typed we will introduce an additional higher kinded type called `TypedPlace`, a unary type constructor with a covariant type parameter. Furthermore it is not desirable to define the algebra of the net class over all types available in the Scala type system. For representing the hierarchy of valid data types the trait representing a net class has to include type information that determines which types are eligible for being used as type parameters. Thus the trait will contain types

`Top` and `Bot` representing the top type and bottom type of the hierarchy. The type parameter of the `TypedPlace` type will be bounded by these types.

```
trait HLNetClass {
  type Top
  type Bot <: Top

  type TypedPlace[+X >: Bot <: Top]

  type Place = TypedPlace[Top]
  // other code omitted
}
```

Example 6.2.1. Given the algebraic high-level net from Example 4.3.1 the sorts of the associated net class can be given as follows:

```
type Top = Sort

type Bot = Nat with Process with ProcessType

sealed trait Sort

case class Nat(x: Int) extends Sort

case class Process(i: Int, t: ProcessType) extends Sort

abstract class ProcessType extends Sort

case object RS extends ProcessType

case object WR extends ProcessType
```

Listing 44: Sorts of Reader-writer net

The trait `Sort` is the top type for all valid types in the algebra, thus the places of the net are typed as `TypedPlace[Sort]`. This example also contains two values `RS` and `WR` that have associated singleton types. In this context it is not desirable to assign these types to a place as this would make the place effectively a constant. Furthermore assigning an element type of `Nothing` to a place has to be avoided (see 3.6). For these reasons the

bottom type `Bot` is assigned as given in the listing. `Bot` is uninhabited as it defines a compound type of class types.

The variables' types are encoded as a trait `Var[+X >: Top <: Bot]` with the type parameter bounded as in the case of the places. Variables are defined as singleton objects extending this trait with the type type parameter representing the variables data type. This encoding of variables allows the type safe construction of the pre-domains of the transitions by requiring place and variable to have their respective type parameters represent the same type.

The problem with the model becomes obvious when trying to offer a type save way of constructing the pre-domains and post-domains of a transition. For satisfying the consistent assignment agreement it is necessary to ensure that variables referenced in the terms of the post-domains are actually included inside the pre-domain. Information about the variables available would have to be encoded in the types of the transitions. This encoding is possible either in by tracking the types involved in the construction in a similar way as it is done in heterogeneous lists (Kiselyov et al., 2004) or by tracking the types of the variables using phantom types (Leijen and Meijer, 1999). The fact that an expression can occur in the terms of multiple incoming arcs makes an implementation that accounts for that in the type system even more complex. The complexity of an implementation that incorporates this would result in a library that is cumbersome to use because of the type operations involved.

The upcoming Scala release 2.10 will add compiler macros[1] to the language. These macros are implemented as ordinary Scala methods that can transform the abstract syntax tree of a program during the compilation phase and even provide new types based on the structure of the program. Inspecting how macros can be used to track the types and variables associated with a transition of a high-level net could lead to interesting results.

The model as is can be extended to implement the semantics of algebraic high-level nets without tracking the types and variables. One possible solution that provides at least type safety during construction of the pre-domain could involve representing the terms as expression trees. The enabling of transitions would then include an interpretation phase that evaluates the expression dynamically. The variables involved would also need to be tracked as values and not as types. The existence of a variable in the pre-domain can not be statically determined at compile time and as such involves additional checks at runtime.

[1] see `http://scalamacros.org/`

A. Appendix

A.1. Sets and Classes

The concept of classes was introduced to be able to describe large collections of sets. Every set is also a class and hence there exists the term *proper classes* to explicitly state that the class in question is not a set.

A.2. Semigroups, Monoids and Groups

A *semigroup* $(S, \cdot)$ consists of a set S together with a binary operation $\cdot : S \times S \longrightarrow S$ satisfying the following axioms:

- *closure*: $\forall a, b \in S : (a \cdot b) \in S$
- *associativity*: $\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$

A *monoid* is a semigroup with unit, i.e. a triple $(S, \cdot, e)$ consisting of a set S, an associative binary operation $\cdot : S \times S \longrightarrow S$ and the unital element $e \in S$. In addition to the axioms of a semigroup the monoid satisfies the axiom for the identity element:

- *identity element*: $\exists e \in S : \forall a \in S : e \cdot a = a = a \cdot e$

If a monoid also satisfies the invertibility property

- *invertibility*: $\forall a \in S \exists b \in S : a \cdot b = e$

it is called a group.

Semigroups, groups and monoids with a commutative operation are called abelian semigroups, abelian groups or commutative monoid respectively.

A.3. Grothendieck Group

The Grothendieck group construction can be used to construct an abelian group from an abelian semigroup.

The Grothendieck group of an abelian semigroup $S = (A, +)$ is $K(S) = S \times S/\sim$ where $\sim$ is the equivalence relation:

$$(s,t) \sim (u,v) : \exists r \in S : s + v + r = t + u + r$$

- identity: (s, s)
- addition: $(s,t) + (u,v) = (s+t, u+v)$
- inverse: $-(s,t) = (t,s)$

When S is not only a commutative semigroup but also a commutative monoid with unital element e we can easily construct an element of the Grothendieck group from an element $s \in S$ by creating a pair (s, e).

A.4. Free Commutative Monoid

Given a set P the *free commutative monoid* generated over P is given by $(P^{\oplus}, \lambda, \oplus)$ with λ denoting the unit of the monoid and $\oplus$ the binary operation of the monoid. For all $u, v, w \in P^{\oplus}$ the following properties hold:

- λ acts as a left and right identity:

$$v \oplus \lambda = v = \lambda \oplus v$$

- $\oplus$ is associative:

$$u \oplus (v \oplus w) = (u \oplus v) \oplus w$$

- $\oplus$ is commutative:

$$v \oplus w = w \oplus v$$

Another common way to represent the free commutative monoid is the sum notation:

$$\sum_{p \in P} \lambda_p p$$

where $\lambda_p \in \mathbb{N}$ denotes the coefficient for element p.

Bibliography

Jiří Adamék, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories*. Dover Publications, Mineola, New York, 2009. URL http://katmat.math.uni-bremen.de/acc/.

Steve Awodey. *Category Theory*. Oxford University Press, 2010.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21:359–411, 1989.

Kurt Jensen. Coloured petri nets: A high level language for system design and analysis. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer Berlin / Heidelberg, 1991. ISBN 978-3-540-53863-9.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: 10.1145/1017472.1017488. URL http://doi.acm.org/10.1145/1017472.1017488.

Daan Leijen and Erik Meijer. Domain-specific embedded compilers. Conference on Domain-Specific Languages, page 109–122. USENIX, 1999.

Martin Odersky Lukas Rytz. Relative effect declarations for lightweight effect-polymorphism. 2012. URL http://infoscience.epfl.ch/record/175546/files/rel-eff_1.pdf.

Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. *SIGPLAN Not.*, 43(10):423–438, October 2008. ISSN 0362-1340. doi: 10.1145/1449955.1449798. URL http://dx.doi.org/10.1145/1449955.1449798.

Martin Odersky. Scala language specification, 2011. URL http://www.scala-lang.org/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf.

Julia Padberg. *Abstract Petri Nets: Uniform Approach and Rule-Based Refinement*. Shaker Verlag, 1996.

Ahmad Ramin and Tavagoli Kolagari. Transformation of open and algebraic high-level petri net classes. Technical Report 2002-24, Technische Universität Berlin, 2002.

Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143169. URL `http://doi.acm.org/10.1145/143165.143169`.

Index

U

V

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. August 2012 Moritz Uhlig