Project 2
Type Systems: Theory and Practice

Ruben Bär
ruben.baer@gmail.com
February 16, 2011

Abstract

Type systems are the fundamental part of programming languages which ensure their formal correctness and prevent programs from execution errors. They provide languages with their level of expressiveness which is where many programming languages have shortcomings. However, many languages are now adapting new concepts like f-bounds etc. and some of them will be presented in this paper.

In the last decades different approaches have been taken to enhance the expressiveness of languages. But there are still ways to improve them and make programs more safe and reliable. For this purpose different type system extensions on diverse paradigms were invented. Although some of them have been shown to be undecidable or intractable this area is still under development and provides ideas for better systems.

Contents

1 Introduction

This papers originate within the second project module at the computer science course at the university of applied science in Berne[1]. The paper will give interested readers a practical and theoretical insight into type systems and programming languages to grasp the importance of type systems within this domain. We think that a good comprehension of type systems and programming language semantics is fundamental for writing good software and tackling difficult and large problems in computer science.

Besides syntactical differences, many programming languages differ in their expressiveness and flexibility. Especially object-oriented languages provide difficult concepts like subtyping, subclassing and multiple inheritance. And these differences that what distinguishes a good, well behaving programming languages.

This papers further aims at showing what difficulties exist in today's languages and why many programming languages includes limitations in their type systems and do not provide full expression power. Thus it gives understanding why the design of a good type system is not easy and has to be a principal part in designing languages, from the very beginning.

1.1 Structure

Giving a comprehensible introduction, the paper starts with an introduction about type systems and which differences exist. Of course, this part is not exhaustive but will give an insight into how complex the area is and which main directions exist. A more comprehensive introductory paper was written by Luca Cardelli and is be recommended to be read for further studies: [Car97].

Subsequently a chapter about type theory is presented. The very foundation, type theory, of type systems is the mathematical approach to them. Some knowledge within this section is necessary for the understanding of type rules in further chapters. The very beginning of type theory was to eliminate contradictions and paradoxes from set theory. Broadly speaking, type theory can be understood as an extension to naïve set theory. Herman Geuvers gave a lecture about type theory in 2008 with special focus on functional programming and reasoning on different levels [Geu09].

With this knowledge, we start presenting how type theory found it's way into computer science and programming languages with simply typed lambda-calculus. To give a feeling how types are constructed and what tasks a type systems has to perform, we step by step present how a simple core calculus is getting extended with more sophisticated type rules, gaining in expressiveness with special remarks on recursion, subtyping and polymorphism.

The topics mentioned above give a perfect foundation to investigate more complex language paradigms. In this paper we will focus on object-oriented programming languages and their difficulties in typing. Further, a short view on object-oriented calculi is given and how they are connected with the lambda-calculus.

Knowing the problems of object-oriented programming languages, the next chapter gives a hands-on to Scala and the features of its sophisticated type system. The small examples will

[1] http://www.ti.bfh.ch

show how some issues other languages have can be tackled and how elements of functional programming can fit well into object-oriented languages.

Finally, some special topics of the domain of type systems and type theory are given. A very short introduction shows that types and mathematics is strongly connected and even how types can have value parameters. Additionally a perspective of a generalisation of subtyping is given with type matching.

1.2 Prior Knowledge

For the full comprehension of this paper, some prior knowledge is inevitable. This paper requires a good knowledge of upper graduate students in computer science or related fields, as well as profound experience in programming in particular following the object-oriented, respectively functional paradigm including knowledge about their backgrounds and an understanding how according languages basically work. In addition, some knowledge of lambda-calculus in general is inevitable. This includes especially the notion, notation, recursion and simple evaluation of lambda expressions.

To support the reader's understanding we here give some references to introductory papers and books.

1.3 Introductory and Further References

Functional Languages

- Graham Hutton. *Programming in Haskell*. 2003

Lambda-Calculus

- Benjamin Pierce. *Types and programming languages*. MIT Press, Cambridge Mass., 2002.
- Richard Gabriel. The why of Y, 15-25, *SIGPLAN Lisp Pointers*, 1988.

Object-Calculi

- Anthony J.H. Simons. The theory of classification part 1: Perspectives on type compatibility. *Journal of Object Technology*, 1(1):55–61, May 2002. (column).
- Martín Abadi and Lucu Cardelli. *A theory of objects*. Springer, New York, 1996.

Object-Oriented Languages

- Kim Bruce. *Foundations of object-oriented languages : types and semantics*. MIT Press, 2002.

Types and Language Semantics

- Benjamin Pierce. *Types and programming languages*. MIT Press, Cambridge Mass., 2002.
- Michael Schinz and Philipp Haller. A scala tutorial for java programmers, November 2010.

Type Systems

- Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, page 2208–2236. CRC Press, 1997.

Type Theory

- Herman Geuvers. Introduction to type theory. *In Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, pages 1–56. Springer-Verlag, 2009.
- Simon Thompson. *Type Theory and Functional Programming*. March 1999.

Scala

- Martin Odersky. *Programming in Scala*. Artima, Mountain View Calif., 1st ed.,version 9 edition, 2008.
- Michael Schinz and Philipp Haller. *A scala tutorial for java programmers*, November 2010.

2 Type Systems

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute. [Pie02]

One essential part most programming languages consist of is the type system. The motivation behind type systems is preventing running programs from occurrences of execution errors, which is a non-trivial property to hold. When a program formulated in a specific language is prevented from execution errors, then it's called type sound.

A type system developed to provide type soundness for a specific programming language needs a precise formulation and notation to proof that the language is safe and sound. In addition to that a type system can be understood as a tool for reason about programs. This shows that type systems raised from the more fundamental fields logics and mathematics. There are type systems - very often with very powerful types like dependent types - used as proof checkers and theorem provers like Coq. The correspondence of types and proofs respectively propositions, in constructive mathematics was shown by Haskell Curry and William Alvin Howard. This relationship is known as Curry-Howard isomorphism or formulae-as-types correspondence. All this has lead to an own formal discipline in mathematics and computer science.

In many programming languages, especially in languages were not designed together with their type system or with a formal method, it was possible to show that the language is type unsound and, what's worse, that different implementations of the language uses slightly different type systems. Although such languages were checked by their type system they can generate execution errors where the programmer has to take care of.

An other reason for type systems is to give programmers a generic tool to validate their code. On one hand this validation process can be used to avoid errors at compile time on the other hand it can be used as supporting refactoring tool. Whenever a programmer wants to replace a data structure in a whole piece of code the typechecker will find the old and invalid use of the type.

2.1 Types

Programming languages are classified in typed and untyped languages. Untyped languages can be considered as a special case of typed languages which just one universal type for every variable and expression. In other words every variable can holds every value independent of its kind of value like boolean, number of function. In contrast a typed language restricts the range of values to its type. A typed language does not define that there are type annotations within the code. If the types are embedded in the syntax the languages is called explicitly typed otherwise implicitly typed and the type system has to infer the types. In languages like Haskell, the type annotation is for many parts optional an is inferred by the type system. Anyway type annotations helps to understand the code.

2.2 Execution Errors

Execution errors can have many different faces. On one hand they lead to immediate crashes, on the other hand they cause data corruption with crashes at later points of execution. Some of these errors are not preventable by the type system like dereferencing null references. This distinction divides execution errors into two groups: trapped errors and untrapped errors. As an example: dereferencing illegal pointers is called untrapped error, where division by zero is trapped. The main difference is that trapped errors cause an immediate error when the appropriate code is executed. Untrapped errors cause in arbitrary behaviour of the program. A language that does not cause any execution error is called safe [Car97].

2.3 Soundness and good behaviour

A language is sound if it is not possible that it causes an execution error and will never fail, which would be ideal; but programming language designs follow different ideas and aims. Although soundness is an optimal aim it is not possible every time. The designer has to decide both the level of expressiveness of the language and the level of safety properties used in the type system. For a given language the designer may define a set of execution errors as forbidden errors. Ideally the set of forbidden errors contains every untrapped error, to prevent unexpected behaviour and contains some of the trapped errors; and gives the opportunity to find easily the source of an error,. If a type system can ensure that no forbidden error occurs we say that the program behaves well. In contrast to good behaviour stands bad behaviour [Car97].

2.4 Properties

Luca Cardelli proposed several type system properties which should be shared among typed and safe languages [Car97].

Decidable The type checking algorithm should be decidable whether or not a program behaves well. As an example; with dependently typed languages type checking becomes undecidable.

Transparent The behaviour of a type system should be transparent to the user so she or he can predict whether the code is checkable or not.

2.5 Typed Languages

Strongly vs. Weakly Typed If a language is free of forbidden errors we say that it is strongly typed. All other execution errors that are not designated as forbidden may occur.

Whenever the type system does not prevent the occurrence of forbidden or untrapped errors we call the language weakly typed.

Statically vs. Dynamically Typed A language can be designed so that unsafe and ill behaving programs will never run. To prevent such a program from running the type checking has to be performed before it is started. Such languages are statically typed by a type checker.

In contrast to statically typed languages, we have the dynamically typed ones. The type checks are performed on runtime and have to prevent that no untrapped error can occur. Depending on the static checks it can be that a statically typed language also contains dynamic

checks like Java. This prevents the program from unsafe operations e.g. indexing an array out of bounds.

3 Introduction to Type Theory

Type Theory is the basis to study formally type systems. This topic is fundamental in mathematics, logic and computer science. In 1903 the mathematician Bertrand Russell found contradictions in the set theory [Irv95]. To eliminate these contradictions he introduced a theory of types. In this age type theory was a new field of logics which later found its way into computer science. Since the beginning of type theory many people improved this theory or found other approaches to it; e.g. with the simple theory of types, Alonzo Church introduced a simpler and more general theory which he translated into his simply typed lambda-calculus; or along with constructivism in mathematics Per Martin-Löf introduced an intuitionistic type theory [Mar84]. An other important milestone is the often mentioned Curry-Howard isomorphism. Although these details are interesting, this paper focuses on the usage of type theory in computer science, especially in programming languages.

3.1 Type Levels and Encoding

Programming languages exist on different levels, from machine code like assembler to high level languages like Java, Scala, Eiffel or Event-B. Depending on the level of the language and the used formal model behind it there are different definitions of what types are: from low level types which are very concrete but not very useful for reasoning up to abstract axiomatic types which can be used to proof theorems within and reason of a model.

3.1.1 Scheme

On the lowest level there is just a scheme for interpreting bit-strings. Since the bit string of the ASCII character 'A' is the same as the integer 65 it is maybe not as good for reasoning than the needs of machine memory would be.

3.1.2 Interface

One of the most used methods for programming languages are types defined on interface level. These types are defined by the interface compatibility which often is based on interface names and type hierarchy, e.g., subtyping. The concrete representation is uninterpreted by the model. Here type compatibility is syntactically defined which leads to a more precise model than the scheme based; ones, although it is still possible to define faulty expressions on this level.

3.1.3 Algebra

An other and more precise level is based on formal axioms. The object's behaviour is defined in detail with axioms that are building an algebra. A type *Ordinal* with algebras is characterised by:

The equations 3.1 define a type with two functions. The first one just returns the first ordinal. Succ(x) is a function with an ordinal as parameter and with an other ordinal as successor of x. The second line describes the behaviour of all ordinal types with a minimum of axioms [Sim02a].

$$\begin{aligned} Ordinal = & \exists ord.first :\rightarrow ord; succ : ord \rightarrow ord \\ & \forall x : Ordinal.(succ(x)! = first()) \wedge (succ(x)! = x) \end{aligned}$$

Figure 3.1: Axiomatically defined ordinal type

Typed languages with axiomatic definitions that are type sound, checked by a type system, never go wrong, i.e., typed expressions yield valid expressions only.

3.2 The Language of Type Theory

Before starting to investigate type systems in detail, this sections presents the language used in type theory, their theoretical foundation. A programming language of course needs more than just typing rules like syntax and evaluation rules corresponding to the typing rules.

First of all some basic assumptions about our universe of values are needed. Some primitive types like Integer, Character and Boolean can be assumed to exist. Naturally these type can be constructed from first principles. Other language concepts can be described and constructed with appropriate typing rules. The ordinal type from equation 3.1 gives an example how basic types can be defined. However, to make such detailed definitions we need a more powerful system than the simple type theory used in the simply typed lambda-calculus, since this does not automatically support existential quantifiers.

As starting point we assume the existence of set theory providing sets, elements, some set operations like $\in$, $\subset$ etc. and logic operations.

3.2.1 Rules of Inference

Typing rules are rules of inference in formal logic given in the following form:

$$\frac{Premise_1 \qquad Premise_2 \qquad \cdots \qquad Premise_n}{Conclusion}$$

An inference rule is a tool to perform formal proofs. If the set of premises is empty, then the rule describes an axiom which is true by definition. A widely known rule in logic is called modus ponens which can be written as following in rule notation:

$$\frac{A \rightarrow B \qquad A}{B} \text{ Modus-Ponens}$$

In type theory there are some operators which can be interpreted as set operators as well. $t : T$ is said if the term t is of the type T and this can be interpreted as $x \in T$ or the subtype relation is a subset relation in set theory. $B <: A$ says that the type B is a subtype of A which corresponds directly to the subset relationship $B \subseteq A$ [Sim02b].

3.3 Soundness = Progress + Preservation

As already seen in chapter 2, soundness is an important property for type systems. Sound programs will never get stuck, i.e. run into an execution error. To proof this property, two

theorems are needed which are called progress and preservation. Progress says that a well-typed program does not get stuck, i.e. that it will yield a value, or an other evaluation step is possible. Preservation guarantees that every evaluation step of a well typed expression yields a well typed expression as well. Both theorems together ensure that a program will not get stuck and will not loop forever. Details can be seen in [Pie02]. There can be found a proof for simply typed arithmetic expressions. For complex type systems this proof can be difficult, especially in object oriented programming languages. A soundness proof is often performed by reduction to simpler core calculi like the lambda-calculus for functional programming languages or the zeta-calculus for object oriented languages.

4 Typed Lambda Calculus

4.1 Simply Typed Lambda-Calculus

With the tools mentioned above the construction rules for the simply typed lambda-calculus can now be presented. The simply typed lambda-calculus is distinguished by two typing paradigms. As explained, languages with type annotation in their terms are called *explicitly typed*. These annotations are used to help the type checker and other users to understand the semantics of the code. Languages which omit type annotations are called *implicitly typed*. Here, the type systems have to implement a type-assignment system which infers or reconstructs the types in a given context. In this paper we are only focusing on explicitly typed languages.

There exists only one kind of type in a purely simply typed lambda calculus, which is the function type:

$$T ::= | T \to T$$

The simplest term defined in the lambda-calculus is a variable. It's typing rule is this:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

The premise says that the variable x with type T exists in the typing context[1] Γ. The context is a sequence of variable-type pairs for all known and typed variables. As long as no variable is bound in a context the latter is written as $\varnothing$ or is just omitted in the typing rule. A new variable can be bound to a context with its *comma*-operator e.g. $\Gamma, x : T$ extends the context with the typed variable x with type T.

Whenever a new variable is bound to a context, the variable has to be distinct to already bound variables. Since in lambda-calculus a variable has to be renamed if it is not free in a term, we can satisfy this condition by renaming the bound variable whenever necessary.

The next syntax rule we have to type is the abstraction, which is done with following rule:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{ T-Abs}$$

Here, the premise binds a new variable to the context and concludes that the term t_2 is of type T_2. This infers that we are now able to proof that a lambda abstraction over t_2 with the variable x of type T_1 is of the type $T_1 \to T_2$. This function type denotes a partial function from the domain T_1 to its codomain T_2.

Last but not least we need a typing rule for applications:

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ T-App}$$

[1] The context is sometimes called typing environment

As premises a function with a domain of type T_{11} and a variable of the same type are needed. Certainly T_{11} can be any valid type. The rule concludes that the application leads to the codomain of the function term t_1.

4.2 Extensions

There is an important observation someone can make with this definition. A purely simply typed lambda-calculus with only these rules and no base types like Integer, Boolean etc. has no well-typed term. The reason of this is obvious and needs no proof: it's simply because our set of valid expressions is empty and can not be derived from the empty context! This immediately leads to some extensions.

4.2.1 Boolean Type

A boolean type is the first extension for the purely simply typed lambda-calculus. With this we can adapt other basic types like Integer, Character, String and many more.

First we need to introduce an other typing syntax and a typing rule to show how an expression generates a derivation tree.

$$\begin{aligned} T ::= & |T \to T \\ & |Bool \end{aligned}$$

This rules introduces a boolean type:

$$\frac{}{\Gamma \vdash true : Bool}\ \text{T-True}$$

$$\frac{}{\Gamma \vdash false : Bool}\ \text{T-False}$$

With this hand full of rules a derivation tree for the expression $(\lambda x : Bool.x)true$ can be constructed, which is the identity function for boolean applied to the true value. This derivation shows that $x : Bool \in \varnothing$.

$$\frac{\dfrac{\dfrac{x : Bool \in x : Bool}{x : Bool \vdash x : Bool}\ \text{T-Var}}{\vdash \lambda x : Bool.x : Bool \to Bool}\ \text{T-Abs} \qquad \dfrac{}{\vdash true : Bool}\ \text{T-True}}{\vdash (\lambda x : Bool.x)true : Bool}\ \text{T-App}$$

4.2.2 Unit Type

The unit type is a useful base type even for languages supporting imperative constructs. It is a single constant type which is widely known as void in languages like C, C++ and Java. The rule is as simple as in the Boolean type, only that there is just one value within this type.

$$\frac{}{\Gamma \vdash unit : Unit}\ \text{T-Unit}$$

4.2.3 Product Type

In programming languages we have an immense use of composed data structures. The simplest way to build them are tuples, i.e., n-ary products which are ordered sequences of n-elements.

The simplest tuple is a binary product $(A \times B)$ that can be interpreted as a Cartesian product with types A and B and with two projections $\pi_1 : (A \times B) \to A$ and $\pi_2 : (A \times B) \to B$ to extract an element of type A or B. An way to build larger data structures is to inductively build up binary products to an arbitrary size, e.g., $t := (A \times (B \times (C \times D)))$. Using this encoding may not be the most clever way. So how can we extract the 3^{rd} element of type C? We have to use the projects above like:

$$\begin{aligned} t' :=& \pi_2\ t : (B \times (C \times D)) \\ t'' :=& \pi_2\ t' : (C \times D) \\ t''' :=& \pi_1\ t'' : C \end{aligned}$$

Since no one would be happy with such a construct, we can generalize a binary product type to an ordered list of arbitrary elements with a fixed size. Now we have the opportunity to introduce a *dot* operator as syntactical sugar to access directly the n^{th} element with a correct type[2].

$$\frac{for\ each\ i \qquad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i^{i \in i..n}\} : \{T_i^{i \in 1..n}\}}\ \text{T-Tuple}$$

$$\frac{\Gamma \vdash t_1 : \{T_i^{i \in 1..n}\}}{\Gamma \vdash t_1.j : T_j}\ \text{T-Proj}$$

4.2.4 Record Type

One drawback of a tuple is that we have to know the exact index of our elements. It would be awkward if someone would like to use tuples for object encoding. We are familiar with specifying names to object members like fields and methods. Fortunately there is an easy generalisation of tuples called labeled records or simply records. Each field has a distinct label. For example, {name=jack, profession=pirate, age=35} is a record value of record type {name:String, profession:String, age:Nat}. Whit this, we can intuitively define the record rules:

$$\frac{for\ each\ i \qquad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in i..n}\} : \{l_i : T_i^{i \in 1..n}\}}\ \text{T-Record}$$

$$\frac{\Gamma \vdash t_1 : \{T_i^{i \in 1..n}\}}{\Gamma \vdash t_1.j : T_j}\ \text{T-Proj}$$

Here it is important to see that a different order of fields yield a different record type. The type {name:String, profession:String, age:Nat} is different from a permutation, e.g., {name:String, age:Nat, profession:String}. Since most programming languages treat both record types equivalent, we have to extend our model to include this flexibility as well. This includes subtyping which we describe in more detail in section 4.3.

[2]Here we use curly brackets like in record notation.

4.2.5 General Recursion

The simply typed lambda-calculus holds the strong normalisation property. This property guarantees that every sequence and program terminates in a normally formed term. Thus since the system has to terminate, it can obviously not be Turing complete, i.e., there are computable functions which we are not able to define in the context of the simply typed lambda-calculus.

However, it is not possible to define a fix point combinator within the language like it was possible in the untyped lambda-calculus. There are different possibilities to extend the simply typed lambda-calculus and perform general recursion which makes the language undecidable and Turing complete. One option is to introduce recursive data structures, which means that a value of such a type may contain other values of the same type. For example natural numbers defined by the Peano arithmetic. In Haskell, such a data type would be defined as data Nat = Zero — Succ Nat. An other option is a new primitive in the language itself which imitates such recursive behaviour.

Benjamin C. Pierce gives a good proof for the normalisation of the simply typed lambda calculus [Pie02, Chapter 12]. The main idea is that he inductively shows how the beta-reduction, better known as function application, preserves the type preserving property which we do not have in untyped lambda calculus. To introduce general recursion we have to 'fix' the beta reduction. A new evaluation rule with the *fix* keyword can be given as follow:

$$fix(\lambda x : T_1.t_2) \rightarrow [x \mapsto (fix(\lambda x : T_1.t_2))]t_2$$

Now the typing rule of the fix operator is obvious:

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash fix\ t_1 : T_1}\ \text{T-Fix}$$

The evaluation of *fix* has to be read so hat every occurrence of x in t_2 is replaced by $fix(\lambda x : T_1.t_2)$. With *fix* we can easily show that we have lost the normalisation property and are no more simply typed. The term $(\lambda x : T.x)$ is a simple diverge function under the application of *fix*. Passing this lambda expression to *fix* yields the same expression every time again. We refrain from giving a formal proof that such a typed lambda calculus is Turing complete. But it is sufficient to see that we can formulate terms that do not halt.

$$fix(\lambda x : T.x) \rightarrow [x \mapsto fix(\lambda x : T.x)]x \rightarrow fix(\lambda x : T.x)$$

Recursive Types

Next to general recursion there is an other important concept of recursiveness: recursive types. The importance of recursive types becomes evident with an example where the classical type constructors $\times$ and $\rightarrow$ reach their limits. Of course it is possible to tackle this problem by introducing them as a special construct in the language like we did with the *fix* operator. But what can be done after introducing all the list rules? Other extensions have to be included since other structures like queues, trees and graphs are needed as well. A language with all these language primitives would grow larger and larger which can not be our goal. The goal should be to have a slim language which is flexible and expressive for other programmers and has a slim core which can be proofed by us against some properties. The data structures we may want to use all have some properties in common. The next task now is to provide a sufficient mechanism to formalise such data structures within a framework.

A type of a list of numbers can be considered as a variant of the *Unit* type and a pair of a number and an other list. *nil* sufficiently tells what is needed to know about an empty list. *cons* is a pair of the current number and the subsequent list.

$$\mathbf{NatList} =< nil : Unit, cons : \{Nat, \mathbf{NatList}\} >$$

Particularly in this case the recursion is obvious. The equation is not a definition because the term on the left hand side is used on the right hand side as well, which is currently in the process of being defined. In type theory they use a new recursive operator that transforms a recursive equation into a proper definition:

$$NatList = \mu X. < nil : Unit, cons : \{Nat, X\} >$$

Now, NatList is defined as infinite type that satisfies the equation $X =< nil : Unit, cons : Nat, X >$. Unfortunately computing an infinite type would not halt and therefore it is not computable either. Fortunately however we do not have to expand the whole infinity in real life scenarios! However, a programming language has to provide rules that perform the μ recursion on type levels. Two rules are clearly sufficient for that. *unfold* rolls out one recursive step and *fold* rolls in one step.

$$unfold[\mu X.T] : \mu X.T \to [X \mapsto \mu X.T]T$$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash unfold[U]t_1 : [X \mapsto U]T_1} \text{ T-Unfold}$$

$$fold[\mu X.T] : [X \mapsto \mu X.T]T \to \mu X.T$$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash fold[U]t_1 : U} \text{ T-Fold}$$

Taking the example of the NatList above,

$$unfold[\mu X. < nil : Unit, cons : Nat, X >]$$

expands to

$$< nil : Unit, cons : \{Nat, \mu X. < nil : Unit, cons : \{Nat, X\} >\} > .$$

Performing the fold operation after an unfold operation yields the previous expression again.

$$unfold[S](fold[T]V_1) \to v_1$$

There exist other methods for recursive types. But if *fold* and *unfold* are concatenated and yield an operator that does not change anything, the recursion is called iso-recursion since both operators are isomorph.

Later, object-oriented programming languages are considered. It is noteworthy that objects are interpretable as records. However, these objects heavily use recursion for self references and accessing the own fields and methods provided by an object.

4.3 Subtyping

The type rules and type extensions we considered so far are all very rigid and do not allow us to pass values to functions with not the exact same type as specified or to return values with different types than specified. To extend a system with more flexibility a concept named subtyping was introduced. Subtyping is heavily used in object-oriented languages and is often, however falsely, referred to as their key concept. Establishing subtyping in programming languages has wide effects in their implementation and affects type rules already defined. Type checking becomes non-trivial with such features and proofing the soundness is not easy anymore. The difficulties to understand such code increases as well. All these issues are maybe the reason for the bad and inflexible type systems in many well known statically typed programming languages like Java, C#, Eiffel and many more.

Here we just introduce subtyping for functions and records. These two extensions are necessary for later chapters and are widely known by many developers. With the simple function application rule we have many expressions which behave well but are not typable, e.g., $(\lambda p : \{x : Nat, y : Nat\}.x \cdot y)\{x = 2, y = 3, z = 3\}$. Such expressions would end up in a type error since the lambda-expression expects the record type $\{x : Nat, y : Nat\}$, but the argument is $\{x : Nat, y : Nat, z : Nat\}$. There, the type $\{x : Nat, y : Nat, z : Nat\}$ can safely be used as parameter where $\{x : Nat, y : Nat\}$ is expected. This relation is called subtyping and is written as $\{x : Nat, y : Nat, z : Nat\} <: \{x : Nat, y : Nat\}$. Sometimes the subtype relation, where S is a subtype of T respectively T the supertype of S, $S <: T$ is referred to a subset relation [Sim02c], $S \subseteq T$ exactly like $x : X$, corresponds in set theory to $x \in X$.

As in set theory, subtyping is reflective and transitive.

$$\frac{}{S <: S}\ \text{T-Reflective}$$

$$\frac{S <: U \qquad U <: T}{S <: T}\ \text{T-Transitive}$$

In other words, subtyping is the safe substitution of expressions within a given context.

4.3.1 Covariance vs. Contravariance

Subtyping is one formal approach for safe substitution. To ensure safe substitution in some specific contexts, there are some natural restrictions to subtyping called covariance, contravariance and invariance. These terms stem from category theory [Pie91]. We understand the term variance as a conversion of types, e.g., the conversion from a supertype to a subtype and vice versa.

Within a type system an expression in a type rule is:

invariante if the type has to match exactly. A type system without subtyping is invariant.

covariant if the type changes in the same direction as the subtype relation, i.e., a covariant expression E within T can change in S to an other expression F iff $S <: T$ and $F <: E$.

contravariant if the type changes in the other direction as the subtype relation, i.e., it is the opposite of covariant, if $S <: T$ then $E <: F$.

This knowledge can now be used to examine the subtyping of arrow types respectively functions and ask the question, what a subtype of a function is? Which subtype relations between $A_1 \to A_2$ and $B_1 \to B_2$ are allowed?

As seen in the section 4.1, to change the parameter is allowed in a subtype direction and not in the supertype direction. With this it is obvious that a subtype function is allowed to change the parameters only in the contravariant way!

Considering the return type, we have a slightly different situation, e.g., with function concatenation we can pass the return value of the function g to the function h; $h \circ g$. Again, as we have seen in the introduction, a passed value does not need to have the exact type of the parameter. To pass a subtype of the parameter's type is a safe substitution. This gives us the definition of the variance of the return type: covariance.

Using both definitions above, the following subtype rule for functions comes into effect:

$$\frac{A2 <: B2 \qquad B_1 <: A_2}{B_1 \to B_2 <: A_1 \to A_2}\ \text{T-SubFunction}$$

4.3.2 Record Subtyping

Subtype relations for records are an other important aspect of subtyping since they correspond to subtyping of objects in object-oriented languages. In record subtyping we have two directions where we have to check the subtype relation. On one hand we have a relation in the width and on the other hand we have a depth relation which considers the elements of the record.

Starting with the width rule we consider the records $S = \{k_1 : S_1 \dots k_m : S_m\}$ and $T = \{l_1 : T_1 \dots l_n : T_n\}$. It should be intuitive that for getting some fields is a safe substitution in contrast to adding new fields.

$$\frac{}{\{l_i \dots T_i^{i \in 1..n+k}\} <: \{l_i \dots T_i^{i \in 1..n}\}}\ \text{T-SubRecordWidth}$$

It is maybe not directly intuitive that a record with more fields is a proper subset of an other record. Here we have to think that a record with more fields describes something with more details and is less general the its supertype. Taking a real life example; we have a record describing human beings with *name* and *age*. If we add the field *gender*, we get a record which is more specific and has less valid values than a record for every human.

The depth subtype relation considers the individual fields within a record. As we have seen in section 4.3.1 a function can substitute other functions if the new function is a subtype of the old one. Using this fact, we can define an additional record rule:

$$\frac{\text{for each } i \qquad s_i <: T_i}{\{l_i \dots T_i^{i \in 1..n+k}\} <: \{l_i \dots T_i^{i \in 1..n}\}}\ \text{T-SubRecordDepth}$$

4.4 Polymorphism

Polymorphism derives from greek and describes something with many different shapes or forms. However, polymorphism has different meanings depending on the context under discussion. Ad-hoc polymorphism is an other name of function and method overloading, subtype polymorphism in object oriented programming is a concept allowing variables to refer to instances of other types iff the instance has a common supertype with the variable type and is usually implemented by providing a subtyping infrastructure. Here we use the term *polymorphism* for parametric polymorphism, formulated independently by John C. Reynolds and later Jean-Yves Girard [Rey74, GF71], [AS96] as a extension to the lambda calculus which is called System-F. It

is noteworthy that the polymorphic lambda-calculus is sometimes called second-order lambda-calculus because it corresponds to a second-order intuitionistic logic vie the Curry-Howard isomorphism [Gro95]. Among object-oriented programmers they call this polymorphism generics since they often use polymorphism for subtype polymorphism.

4.4.1 System F

In section 4.2.5 about recursive types we have seen how to implement a list for natural numbers. Do create a list for characters it is necessary to copy the code and replace every type occurrence of Nat with Char which yields a CharList. Doing the same for boolean would create again a new copy of the code and so forth. That this is unsatisfactory is obvious. To remove such unnecessary code duplications the System F defines a straightforward extension to the lambda-calculus by adding type abstractions and type applications which is the same as term abstraction and applications just on type level. System F allows here to use every type as type parameter without any restrictions, it just have to make sense in a given context. So types introduced by System F are universal types. In contrast to them there exists existential types too. This existential type are encodable by universal type like in quantified logic. Because of the difficulty of existential types we omit them in this report.

First, we start with a very small example of a parametric identity function; $id = \lambda X.\lambda x : X.x$. Here, id is a function at type level since X in argument expecting a type and not a value. id is now a type constructor which has to bi instantiated with a type, e.g., $id[Nat]$. This evaluates to a function $\lambda x : Nat.x$ of type $Nat \to Nat$. Since every type is valid as parameter the type of id is $\forall X.X \to X$. This helps to distinguish polymorphic functions from instantiated functions. The typing rules for polymorphic abstraction and application is similar to the rules of lambda abstraction and application.

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2}\ \text{T-TAbstraction}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_2}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}}\ \text{T-TApplication}$$

The tools are now read to refine the NatList to a polymorphic List.

$$List\ T = \lambda T.\mu X. < nil : Unit, cons : \{T, X\} >: \forall T.(T \to R \to R) \to R \to R$$

To produce a NatList based of this polymorphic list is easily done by instantiate List with Nat; List[Nat] which yields the NatList again;

$$NatList = \mu X. < nil : Unit, cons : \{Nat, X\} >$$

Here has to be remarked that we are able to define type erasure function. These make it possible to convert the polymorphic lambda-calculus to a simply typed lambda-calculus and even an untyped lambda calculus. This fact was used by the generics in Java. To preserve binary backward compatibility of Java 1.5 to legacy code. Java Bytecode contains no generic codes.

Type Constructors

For understanding f-bounds it is essential to understand type constructors, i.e., type operators. Type constructors are functions like *id* in section 4.4.1. A n-ary type operator is a function taking zero or more type parameters and returning an other type. Type constructors are basically a typed lambda-calculus on an other level. This calculus has only the basic type *star* which is called a kind. One may argue that a function of type $Nat \to Nat$ is a type operator as well because passing a value changes the type to the basic type Nat. But the essence of type operators are that they are taking kinds, the type of types, and constructing functions that accepts values.

4.4.2 F-Bounds

One essential drawback of the pure System F is that every type is allowed in the type application and so no one can use operations on variables of the introduced type through the polymorphic abstraction, e.g., a tree node may just want allow records which contains a *children* field.

$$TreeNode = \forall N.\lambda n : N.\{orig = n, children = map\ TreeNode\ n.children\}$$

Such code would just produce a type error since the type checked can not know that n contains a *children* field. One way to escape this misery is to combine subtyping with the universal quantification which is called Function-Bound or short F-Bound. The extension for System F is simple System $F_{<:}$ and is called "f-sub" and is straightforward as well but we have to modify many of our rules.

However, type operators are functions producing types. This fact can be used to bound a type argument to a subtype relation. Recalling that a subtype relation is something similar to a subset relation in set theory we may express the same principle in first-order logic; $\forall x \in P : P \subseteq \mathbb{N}(Q(x))$.

This can be translated to our problem stated above. The type parameter N is now enforced to be a subtype of $\{children : List[TreeNode]\}$.

$$TreeNode = \forall N <: \{children : List[TreeNode[N]]\}.\lambda n : N.\{orig = n, children =$$

$$map\ TreeNode[N]\ n.children\}$$

System-$F_{<:}$ Type Rules

Figure 4.1 shows new and all affected rules of the basic lambda calculus rules for abstraction, application and variable. The main difference is that we have to introduce the context in every subtyping rule. Without this information the type checker would not be able to check the bounds since they are context sensitive.

The rules presented in this chapter are more or less straightforward and do not need explicit explanation. However, one rule is more sophisticated then the others and thus needs some extra explanation. Rule T-All may be understood as follows. There are two types; $T = \forall X <: T_1.T_2$ and $U = \forall X <: S_1.S_2$ with a subtype relation of $S <: T$, which is deducible by its premises. This allows contravariant changes in the bounded quantification. Whereas the first premise is self-explanatory, the second premise can be explained saying that $S_2 <: T_2$ if $X <: T_1$ and X is free in S_2 and T_2. This gives S a stronger domain and T a stronger codomain.

Such a system became undecidable since the rule T-All does not terminate for all inputs because we are able to define type relations which diverge. Therefore many systems do not provide a full System-$F_{<:}$ but makes some restriction within the type system.

$$\frac{}{\Gamma \vdash S <: Top} \text{ T-Top}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

$$\frac{}{\Gamma \vdash S <: S} \text{ T-Reflective}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \text{ T-Abstraction}$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T} \text{ T-Transitive}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_1 \to T_2} \text{ T-Application}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \text{ T-Arrow}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1.t_2 : \forall X <: T_1.T_2} \text{ T-TAbstraction}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11}.T_2 \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}} \text{ T-TApplication}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1.S_2 <: \forall X <: T_1.T_2} \text{ T-All}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \text{ T-TSubtype}$$

Figure 4.1: System $F_{<:}$ Type Rules

5 Object-Oriented Languages

The object-oriented principles were introduced in programming languages in the late 60s with Simula 67. It's noteworthy that Simula 67 was already shipped with an automatic garbage collector. The first object-oriented language which is similar to today's languages is Smalltalk. Its basis are objects and messages sent between two objects. This concept is now adopted by many other languages. Since this time many other object-oriented programming languages were created, all with their own specialities. These languages are dealt with as a major improvement over older procedural languages. Concerning statically typed languages we can see that many and very popular languages have a poor type system. Either they are too restrictive and lead to a massive use of checked or unchecked type casts, or the are too weak and allow code which can lead to execution errors. Since many statically typed programming languages have a too restrictive type system they all provide mechanisms to escape from the type system. It's so that these languages lack in expressiveness. Object-oriented features are, with the massive use of inheritance and subtyping, heavy to describe in a flexible and safe system.

5.1 Object-Oriented Calculi

Although object-oriented languages are one of the most successful concepts in programming languages no consistent and comprehensive theoretical background existed so far. A popular approach has been presented by Abadi & Cardelli [AC96b]. They defined many constructs found in object-oriented programming languages in a small core calculus.

5.1.1 Object Calculus

In the Theory of Objects Abadi & Cardelli constructed a minimal calculus describing the features of object-oriented programming languages. We can consider this calculus as the object-oriented counterpart to the lambda calculus which is the basis of most functional programming languages. Corresponding to the lambda calculus there exists an untyped object calculi as well as first, second and higher ordered caluli. Abadi & Cardelli called this calculus the untyped ζ-calculus fully defined by four kinds of terms.

a, b	$::=$	terms
	x	variable
	$[l_i = \zeta(x_i)b_i^{i \in 1..n}]$	object (l_i distinct)
	$a.l$	method invocation
	$a.l \Leftarrow \zeta(x)b$	method update

The close relationship between the λ-calculus and the ζ-calculus is obvious. In their theory an object is simply a collection without an explicit order of methods. Each method has at least a bound variable for *self* or *this* and a body that produces a result. There are two distinct operations on objects; method invocation and method update. Like in the lambda calculus we are able to encode booleans, natural numbers etc. within this basic calculus. But not only data types are encodable like that. It's possible to describe functions as objects as well. This makes it possible to formulate the lambda calculus within this theory. Since this reduction is possible we can easily prove that this theory is Turing complete. However, this theory fast

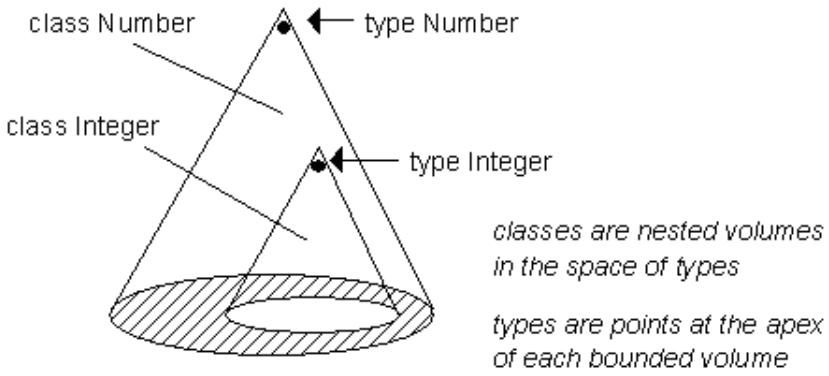

Figure 5.1: Classes as volumes containing types as points [Sim03]

leads to a complex and difficult calculus which requires a good knowledge of all the details to understand the type constructs.

5.1.2 A Theory of Classification

As mentioned above, the lambda calculus can be formulated within the ζ-calculus. But the other way is possible too. A slightly different approach was presented by Anthony Simons' Theory of Classification [Sim02a]. In the Journal of Object Technology he introduced in 20 parts a theory for objects using the lambda calculus. The articles are especially written for non theoreticians and this uses a very intuitive way in describing his theory.

In contrast to The Theory of Objects he does not declare methods as first-order objects and therefore keeps the concepts of objects and methods separated. However, the calculus is more readable and easier for interested people without a deep theoretical background.

The objects are encoded by recursive records with the μ notation.

$$aPoint = \mu self.\{x \mapsto 2, y \mapsto 3, equals \mapsto \lambda p.(p.x = self.x \wedge p.y = self.y\}$$

For the notion of inheritance Anthony Simons simply introduced a special union operator with override, where the union set prefers the methods of the subclass.

Unlike many object oriented languages subtyping is here not automatically created by subclassing and this gives his metalanguage a better expressiveness compared to languages like Java etc.

He illustrated the relationship between classes and types as well which shows figure 5.1. The exact types are the points on the top of the cone since the type is the least specification a class has to implement. The volume of the whole cone are all types within this class of types. A subtype, e.g., Integer, of number is a point again with its one cone under the code of number. This relationship is essential within this theory and also the reason of its name.

5.2 Object Based vs. Class Based

The encoding structure of objects in object-oriented programming languages might be given in following different approaches. On one hand there are class-based languages introducing a special language construct, classes, which are kinds of extensible templates only used for the instantiation of objects at runtime. On the other hand there are languages without such a construct and are called object-based languages. As an example, JavaScript is an object-based language; in particular it is a prototype-based language which is a form of object-based. Such

```
1  function Mamal() { // Define new mamal object
2    this.voice = "Mamal";
3  }
4
5  Mamal.prototype.say = function() {
6    return this.voice;
7  }
8
9  Cow.prototype = new Mamal; // Defining Subclass (No subtype!)
10 Cow.prototype.constructor = Cow;
11 function Cow() {
12   this.voice = "Muh";
13   this.name = "Cow";
14 }
15
16 Cow.prototype.isAm = function() {
17   return this.name;
18 }
19
20 cow = new Cow();
21 document.write((cow instanceof Mamal) + (cow instanceof Cow)); // False, True
22
```

Figure 5.2: JavaScript Subclassing

languages allow the programmer to dynamically update instance variables as well as methods. If the language is statically typed the variables and methods overridden on runtime have to be type compatible with those original. However, if we want to instantiate a new object we have to clone a prototype or an instance of an object.

An other important variation of these two concepts is inheritance. If objects are directly included in the language we are not able to inherit classes since this construct is missing. To mime such constructs of inheritance these languages provide other similar features which allow the programmer to extend objects or to delegate operations to other objects.

The mechanism of extension is fairly easy. An object only needs to be declared that it extends an available prototype an then we may update available fields or add new features to the prototype. With this the new object prototype clones the original fields and creates an object completely decoupled from its parent. How this decoupling looks like is illustrated in figure 5.2. The defined subclass is no instance of its super class.

The method of delegation works similarly as the delegation in the GoF Patterns [Gam95]. A new object prototype delegates the original methods to the parent object it depends on. If we now update or change the parent object we also change the new object and vice versa.

A more detailed presentation these concepts and how they will fit in to a calculus can be found in [AC96b, Chapter 4].

5.3 Subtyping vs. Subclassing

Subtyping and subclassing are two other concepts which often lead to confusion in particular the nomenclature is sometimes used very incoherent. But they are two different approaches which should not be mixed up although there are languages where they are strongly coupled.

As we have seen in section 5.2, JavaScript provides subclassing by cloning prototypes but no subtyping. Once a prototype is cloned the connection between the original and the clone prototype gets lost and both can be used independently. JavaScript uses duck typing to mime such behaviour.The name duck typing refers to the duck test by James Whitcomb Riley:

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

As long as the necessary field exists the type of an object is valid in the used context. If an object is considered as a duck and contains all attributes of a duck it is used as a duck.

In contrast to JavaScript there are many languages connecting these principles strongly together. To name only a few examples; Java, C++, C#, Delphi and Smalltalk. Every sub class relation between two classes are automatic subtypes in the same direction. There exists no possibility to decouple these concepts which restricts the expressiveness of these languages enormously. All subclasses have to obey the covariant, contravariant and invariant restrictions if provided. That this concept is not expressive enough for particular situations is shown by the Circle-Eclipse problem [Mar96]. To solve such problems a decoupling of type and class hierarchy would be helpful.

We have already seen in section 4.3 what subtyping in the context of functions and records is and in section 5.1.2 that recursive records can be used as object representation. The concept of object subtyping should thus be clear. Readers interested in further information are welcomed to take a look in the following literature [Gun94].

5.4 Typing Problems

In studying programming languages it is inevitable to know the current problems of many object-oriented languages, in order to know how to do it better and more expressive.

In general, object-oriented languages are difficult to type check. Subtyping, as example, is easy as long as we just add new fields. Changing types of methods or instance variables can destroy type correctness. Today many class based programming languages ensure that a subclass relation on class level yields a subtype relation on object level. This property forces that changes in classes are inflexible and leaks on expressiveness. If we allow to change a method type we have to ensure that the old method holds the function subtype relation which is only correct in a contravariance implementation. Additionally many programming languages do not provide all subtype possibilities. However, if one would like to override the equals method with its own type as parameter, this would still be impossible since parameters have to be contravariant to be type safe.

5.4.1 Self-References

Ii is clear that object-oriented programming languages have to use self references and automatically a recursive type. Without those they were not able to access their own fields like instance variables or methods, without this feature. However, the most objects would be senseless. For the self references we often use a special variable called *this* or *self*. However, what type should *self* have? In Java, every method is implicitly declared as virtual and is resolved over a virtual table. In figure 5.4.1 we see a simple example of Java code. The output of this program executing the main method would be a "B". So the *this* in line 3 is from the runtime type B and not A.

```
1 public class A {
2   public void print() {
```

```
3       System.out.println(this.name());
4     }
5     public String name() {
6       return "A";
7     }
8     public static void main(String[] args) {
9       B b = new B();
10      b.print();
11    }
12  }
13
14  public class B {
15    public String name() {
16      return "B";
17    }
18  }
```

Listing 5.1: Call of a Java virtual method

5.4.2 Binary Methods

An other problem arising with today's object-oriented programming languages are binary methods. They are called binary since they have exactly two parameters. One of them is *self* and the other is a user defined second parameter, e.g., equals, greaterThen, lessThen, add etc. are all binary methods. For the first class, such methods are no problems. Considering the methods, *equals* of the object O would have the type $O \times O \rightarrow Bool$. But what about its subclass $S <: O$? Since method subtyping has contravariant parameters we are not allowed to define the equal method as $S \times S \rightarrow Bool$. Now, if we want to override this method we are forced to use runtime type checks and runtime type conversions. As users of this objects, we can now not be sure that all these methods would be meaningful. This is not only a problem in such trivial cases as mentioned above. For tree and other data structures it would be helpful if we could specify the expected type and not only the highest used definition.

5.4.3 Return Type

An other problem of object-oriented programming languages and their type hierarchies are the contravariant return type of methods. A situation shown in listing 5.2 respectively 5.3 is desired in many situations. Unfortunately such constructs are not permitted and a type system has to refuse them because the do not build a subtype relation.

However, this examples shows an other problem as well. A variable that changes the type in subclasses is not allowed. They have to be invariant! This makes type casts inevitable and allows the program to fail at runtime execution which is undesirable.

```
1  public class Node {
2    private Command cmd;
3    public Command getCommand() { ... }
4  }
5
6  public class GUINode extends Node {
7    private GUICommand cmd;
```

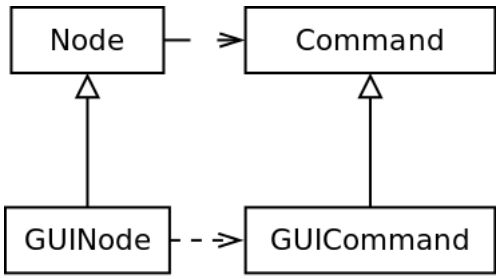

Figure 5.3: Type hierarchy for nodes and their commands

```
8
9     // Method refinement
10    @Override
11    public GUICommand getCommand() { ... }
12  }
```

Listing 5.2: Node classes with covariant return types

5.5 Bounded Parametric Polymorphism

F-bounded parametric polymorphism is a straightforward extension for object-oriented languages which is similar to f-bounded polymorphism in lambda calculus. Implementations are often called *generics* in Java or C#, respectively *templates* in C++. Concrete seen they differ a lot in their implementation and in their semantics since C++ templates support metaprogramming and are Turing complete[1]. Anyway, here we can just take a view onto generics in Java and how they behave in different situations. The theoretical background can be found in section 4.4 at in [Pie02, IPW99].

5.5.1 Generics in Java

Since 1995 Martin Odersky from the Swiss Federal Institute of Technology in Lausanne (EPFL) and Philip Wadler, of Bell Labs, Lucent Technologies started with a research project called Pizza[2] that implements an extension to Java with parametric polymorphism aka generics, function pointers aka first-class functions and class cases and pattern matching aka algebraic types [OW97, BOSW98]. Based on this knowledge Odersky and Wadler started a second project called GJ (Generic Java)[3] which found its way into the Java 5.0 standard. To ensure backward compatibility they implemented the generics with a type erasure that maps GJ code to standard Java code by erasing generic type informations as we already mentioned for the typed lambda calculus.

The listing 5.3 shows a comparison of legacy Java code without generics and new Java code with generics and how the legacy code needs to use unsafe type conversions.

```
1  public class Foo {
2    public int inc(Integer i) {
3      return i + 1;
4    }
```

[1] see http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html
[2] see http://pizzacompiler.sourceforge.net/
[3] see http://lampwww.epfl.ch/gj/

```
5
6     public void foo() {
7       List ints = new ArrayList();
8       ints.add(1);
9       ints.add(2);
10      for (Object i : ints) {
11        // Explicit and unsafe type conversion!
12        inc((Integer)i);
13      }
14    }
15
16    public void bar() {
17      List<Integer> ints = new ArrayList<Integer>();
18      ints.add(1);
19      ints.add(2);
20      for (Integer i : ints) {
21        inc(i);
22      }
23    }
24  }
```

Listing 5.3: Simple generic Java example

Wildcards

But how about the type compatibility of generics? There are still some drawbacks that have to be resolved, which we will see then considering the following two type declarations: `List<Integer>` and `List<String>`. These two types are obviously not type compatible even though both are lists. If someone would want to invoke operations on these two lists which are completely independent from the type parameter like `size()`, we may declare the list holding `Objects`. But `List<Object>` is still not type compatible with the other list since they are not contravariant compatible.

For tackling this problem Java provides a type wildcard "?" which stands for arbitrary types. This leads to the following subtype relation `List<Integer> <: List<?>`. Since return types are covariant this can be used to write functions like in listing 5.4. However, because it is not type safe any more to add new elements to such lists we need some other constructs to get covariant and contravariant changes.

```
1  public void printObjects(List<?> lst) {
2    for (Object o : lst) {
3      System.out.println(o);
4    }
5  }
6
7  public int getSize(List<?> lst) {
8    return lst.size();
9  }
```

Listing 5.4: Function with type wildcard

F-Bounds

To cover covariant and contravariant changes in generic types, we have to introduce f-bounds. This section covers only generics on class level. Java and many other languages would allow generic methods which are independent from their class. Anyway, the semantics of these generic methods is similar and does not much differ from the class level, which is why the section will not need to go into them. For more details of Java generics see [4] [Naf07].

As seen in 4.3.1 and 4.4.2 f-bound is a boundary for type parameters specified by a type operator. For this Java introduced two keywords to specify whether the boundary is an upper or a lower bound. Depending on which is used it semantically is a specification of contra respectively covariant changes.

T extends Bound<T> To specify a polymorphic parameter `T` with an upper bound, the keyword `extends` can be used by the type declaration. The use of `extends` is shown in the listing 5.5. This defines the f-bound $\forall T <: Expandable[T].TreeNode[T]$ and makes the use of the parameter covariant.

T super Bound<T> To specify a polymorphic parameter `T` with a lower bound, the keyword `super` can be used by the type declaration. This defines the f-bound $\forall Expandable[T] <: T.TreeNode[T]$ and makes the use of the parameter contravariant.

```
1  public class TreeNode<T extends Expandable<T>> {
2    public T orig;
3
4    public List<T> getChildren() {
5      return this.orig.expandChildren();
6    }
7  }
8
9  public interface Expandable<T> {
10   public T getObject();
11   public List<T> expandChildren();
12 }
```

Listing 5.5: F-Bounded TreeNode in Java

[4] see http://download.oracle.com/javase/tutorial/java/generics/index.html

6 Scala Programming Language

Scala is a modern multi paradigm, statically typed programming language developed mainly at the École Polytechnique Fédérale de Lausenne (EPFL) by Martin Odersky since 2001. The main paradigms are object oriented and functional. Scala can be executed on the JVM or the .NET VM.

The name of the language derives from "*sca*lable *la*nguage" and with this emphasises the idea behind the language. It is a compact language that only contains minimal operations and control constructs. All other operators and control constructs are implemented within the framework and are not part of the language. This allows the programmer to build an own domain specific language within Scala itself. Further scala makes heavy use of type inference and allows the programmer to keep the program as small as possible and without having to overload the code with unnecessary declarations. This language has the goal to be flexible and expressive as well as to provide general purpose.

Here we dedicate a whole chapter this language. However, we do not want do give an introduction to the language [Ode08, Wam09, SH10]. For readers familiar with object-oriented languages the examples should be easily understandable. Although the language is rather new there already are good books and tutorials worth wile to read. Although scala makes heavy use of type inference and gives the programmer the opportunity to remove many unnecessary constructs, the code in this section is very explicit since the reader should not be confused by it.

Our interesting to this language mainly is its highly sophisticated type system which contains constructs that are new to many programmers. This innovative type systems is designated to be consistent, complete and eliminates many drawbacks from other languages like Java, C++, Eiffel, C#, Smalltalk and many more. So, we herby present three features of the type system that many developers have missed in other object-oriented languages;

- Abstract types
- Explicitly typed self types
- Algebraic data type aka type cases

6.1 Abstract Types

Most object-oriented languages provide features to declare classes and methods as abstract. An abstract class, respectively method, is a place holder that can already be referenced but without full functionality, i.e., such classes can not be explicitly instantiated but has to be refined and finished in a subclass. But commonly known languages do not yet provide the possibility to define abstract type parameters that have to be refined in subclasses. And exactly this Scala has implemented and Scala also helps to eliminate some problems found in binary methods described in section 5.4.2.

In listing 6.6 the use of abstract types is demonstrated. The declaration within the `Node` class of the `type Cmd <: Command` is an abstract type. Every occurrence of the type `Cmd` within the class has to be specialised by the subclasses. Further, the type system enforces that the specialisation is a subtype of `Command`. The benefit of this construct is that constructs like `val`

`n:Node = new GUINode(); n.execute(new ConsoleCommand());` are detected by the static type system and produce a type mismatch by the compiler, since `execute` would expect a ConsoleCommand object.

```
1  abstract class Command { ... }
2  abstract class Node {
3    type Cmd <: Command;
4    def execute(cmd: Cmd);
5  }
6
7  class GUICommand extends Command { ... }
8  class GUINode extends Node {
9    type Cmd = GUICommand;
10   override def execute(cmd: GUICommand) { ... }
11 }
12
13 class ConsoleCommand extends Command { ... }
14 class ConsoleNode extends Node {
15   type Cmd = ConsoleCommand;
16   override def execute(cmd: ConsoleCommand) { ... }
17 }
```

Listing 6.6: Scala code: Abstract type declaration

The type error is maybe a bit strange at the first glance, but Scala is expecting a type that is indexed by an object reference. Such types are called path-dependent, path meaning referencing an object.

```
error: type mismatch;
 found   : ConsoleCommand
 required: n.Cmd
       n execute(new ConsoleCommand)
```

6.1.1 Path-Dependent Type

So how can we fix the call of `execute` in such a way as to not run into a type error? From the error message we can conclude that code like `val n:Node = new GUINode(); n.execute(new GUICommand());` would not help since the argument requires the type n.Cmd. Unfortunately instantiation of abstract type variables is not possible in scala because we can not know which parameters the constructor expects. This issue can be tackled with a factory method like in figure 6.7.

```
1  abstract class Command { ... }
2  abstract class Node {
3    type Cmd <: Command;
4    def execute(cmd: Cmd);
5    def createCommand(): Cmd
6  }
7
8  class GUICommand extends Command { ... }
9  class GUINode extends Node {
```

```
10    type Cmd = GUICommand;
11    override def execute(cmd: GUICommand) { ... }
12    override def createCommand(): Cmd = { return new GUICommand; }
13  }
14
15  class ConsoleCommand extends Command { ... }
16  class ConsoleNode extends Node {
17    type Cmd = ConsoleCommand;
18    override def execute(cmd: ConsoleCommand) { ... }
19    override def createCommand(): Cmd = { return new ConsoleCommand; }
20  }
```

Listing 6.7: Updated scala code: Abstract type declaration

This factory method is now used to create an instance of a Command that can be passed to `execute`, e.g., `val n:Node = new GUINode(); n.execute(n.createCommand());`. Further, if we have two values with the same reference like `val m:Node = m;` scala treads `m.Cmd` and `n.Cmd` as two different types no matter whether both variables are referencing the same object or not since the type system is static and can not use runtime information.

This semantics of path-dependent type includes inner classes as well. As opposed to Java, an inner class depends on the outer object and not on the outer class. As a result, scala can not instantiate an inner class without using a reference of the outer class.

6.2 Explicitly typed self Types

An other type feature in scala is the explicitly typed self type. Normally, the special variables *self*, respectively *this*, are implicitly typed with the object type itself. But in some cases, the type of this is not finally specified at the moment of declaration. To clarify the motivation behind this strategy the following example in listing 6.8, is given.

```
1   abstract class Tree {
2     type Node <: TreeNode;
3
4     val root: Node = newNode();
5
6     abstract class TreeNode {
7       def expand(): Node;
8     }
9
10    def newNode(): Node;
11  }
12
13  abstract class BinTree extends Tree {
14    def newNode(parent: Node): Node;
15
16    abstract class BinTreeNode extends TreeNode {
17      self: Node =>
18
19      var left: Node;
20      var right: Node;
21
```

```
22      override def expand(): Node = {
23        left = newNode(self);
24        right = newNode(self);
25        return self;
26      }
27    }
28  }
```

Listing 6.8: Abstract tree with explicitly typed self

Line number 17 in listing 6.8 shows how the self type of self in the abstract class `BinTreeNode` is explicitly set to the abstract type `Node`. This is necessary since the method `newNode` expects as parameter a `Node` type. At this point we can not know what type `Node` will explicitly have at compile type. The self type can not enforce the compile to ensure that `BinTreeNode` is a proper subtype of `Node`.

6.3 Algebraic Type

Many modern functional programming languages know a feature called pattern matching. In Haskell, pattern matching is realised with algebraic types which are sometimes called variant types. Listing 6.9 shows the type constructor `BinTree` with two type constructors `Branch` and `Leaf`. Further, the function visit accepts data of the type `BinTree` and its body tries to match a specific type with a type constructor. Depending on which constructor got matched a different block of code will be evaluated. If it matches on a leaf, only the value will be returned, otherwise, a tree branch is matched that evaluates a function that takes the left and right side of the branch as argument.

```
1  data BinTree f a = Branch f (BinTree a) (BinTree a) | Leaf a
2
3  visit :: BinTree f a -> a
4  visit (Leaf a) = a
5  visit (Branch f l r) = f (visit l) (visit r)
```

Listing 6.9: A binary tree in haskell with functions on the branches

Scala provides a similar mechanism for algebraic types with case classes and pattern matching which is shown in the following example in listing 6.10, which is a fully runnable code.

```
1  abstract class BinTree[A] {  }
2  case class Branch[A](f: (A, A) => A, l: BinTree[A], r: BinTree[A]) extends BinTree[A] { }
3  case class Leaf[A](a: A) extends BinTree[A] { }
4
5  class TreeTest {
6    // tree visitor
7    def visit[A](tree: BinTree[A]): A = {
8      tree match {
9        case Branch(f, l, r) =>
10         return f(visit(l), visit(r));
11       case Leaf(a) =>
12         return a;
13     }
```

```
14    }
15
16    // setup evaluation tree
17    def setup(): BinTree[Int] = {
18      return new Branch[Int]((_+_), // injecting + as anonymous function
19        new Branch[Int]((_*_), // injecting * as anonymous function
20          new Leaf[Int](3),
21          new Leaf[Int](4)),
22        new Leaf[Int](5));
23    }
24
25    // run test
26    def test() {
27      println(visit(setup())); // => prints 17
28    }
29  }
30
31  object Launcher {
32    // main method
33    def main(args: Array[String]) {
34      val testObject = new TreeTest();
35      testObject.test();
36    }
37  }
```

Listing 6.10: A binary tree in haskell with functions on the branches

7 Special Topics

As seen in the earlier chapters, there are many difficulties which are not expressible within many static type systems since they are too restrictive and are may the reason why a lot of programmers tend to use dynamically typed languages. Scala is maybe an attempt to face this trend by providing a flexible as well as safe type system. Nonetheless there are many other ideas to extend the expressiveness and safety of languages. This chapter gives a short overview of special topics influencing today's type systems and language semantics. The first section introduces the notion of dependent types, and a further a section is depicted to type matching which is an alternative to subtyping.

7.1 Dependent Types

This paper so far presented three families of abstractions. The simplest form are terms that are indexed by other terms, i.e., are parametrized by terms. Parametric polymorphism are terms indexed by types and type operators are types indexed by types.

$\lambda x : T.t$	terms index by terms
$\lambda X.t$	terms index by types
$\lambda X.T$	types index by types

It is thus clear that one kind of abstraction is missing: types can be indexed by terms as well. Types abstracted by terms are called dependent types and are especially used in intuitionistic type theory [NPS90].

Dependent types offer more precision for typing terms. The name *dependent types* comes from the notion that the resulting types of methods and functions are depending on the values of the arguments. This precision enables us to reason about programs in a more sophisticated way. An example will yield a better understanding. We are considering the matrix multiplication of two matrices A and B. The multiplication is only defined if A has the dimension $l \times m$ and $m \times n$. Using non-dependently typed type systems we could not enforce that the dimensions of two matrices are compatible with each other and it would have to be checked at runtime. Such behaviour is obviously undesirable. Dependent types shall fill this lack of safety within type systems. A matrix multiplication function *mul* would maybe types like

$$mul : Matrix[l, m] \to Matrix[m, n] \to Matrix[l, n]$$

If these parameters of the first two matrices would not match with the correct dimensions the type system could reject it with a type error. Consequently boundary checking of arrays becomes unnecessary. In C it is obligatory to specify the arrays size at the beginning and they are not dynamically defined. These informations at compile time could be used in dependent types to eliminate unnecessary boundary checks.

The next typing rule shows a partial definition for a dependently typed list as well as how the type changes if a new element is inserted with *cons*.

$$\frac{\Gamma \vdash X}{\Gamma \vdash nil : List\ 0\ X}\ \text{T-EmptyList}$$

$$\frac{\Gamma \vdash X \qquad \Gamma \vdash n : Nat \qquad \Gamma \vdash x : X \qquad \Gamma \vdash xs : List\ n\ X}{\Gamma \vdash cons\ x\ xs : List\ (1+n)\ X}\ \text{T-ConsList}$$

7.1.1 Dependently Typed Languages

Some research today is concerned to build practical languages with dependent types. Cayenne, Epigram or Dependent ML are only two of a handful of such languages. But not only in programming languages dependent types are used. Automatic theorem proofers are using constructive logic and type theory as foundation. Further details can be found in section 7.1.3. All of these languages are functional to keep type checking reasonable. Anton Setzer provides an approach for object-oriented programming with dependent types [Set06]. Since object-oriented programming languages are per se difficult to type he used the idea of interactive programs, which means that the program is changing within execution. This interaction is similar to monads in functional programming. This makes the reasoning about such programs more difficult.

A comprehensive introduction is given by [McK06] or [Pie05, Chapter 2]. But all of them require good knowledge of type systems for fully understanding. A very simplified ML-dialect implemented in Haskell is shown in [Lea07]. The types are only allowed to depend on numbers and not on more sophisticated values.

7.1.2 Undecidability

As already mentioned in other sections and declared in section 2.4, the decidability of type systems is one of the most important properties to ensure. This shows one of the major drawbacks of dependently typed programming languages. Of course, it is desirable to have a language that is Turing complete. However, extending such languages with dependent types makes them intrinsically undecidable because type checking is the process of checking whether two types are equal or not and dependent types are indexed by terms. Such type systems are undecidable if terms are not strongly normalising, i.e., are not Turing complete [Pie05].

To make a type system decidable again, a language design can follow a totalitarian approach and remove general recursion in a language like Coq and Lego have done.

7.1.3 Curry-Howard Isomorphism

In constructive respectively intuitionistic logic an object is only proven if this object could be constructed. This holds that in this system basic axioms like the law of excluded middle and double negation are missing and are thus not provable within the system. The excluded middle is not provable in this logics since someone might argue that a middle exists and was just not yet constructed. This idea yields an interesting comprehension of types. If we construct an object within programming we can consider a type as its proof since the object has been constructed. Further a function from type A to B is an implication, since we have a mapping from one object to an other, i.e., if the proof A is true and we can map all its values to B the assumption is true. But if a map from A to B fails we know that the proposition was wrong and this assumption has the same semantics as a logical implication.

Formula	Types
Universal quantification	Dependent product type
Existential quantification	Dependent sum type
Implication	Function type
Conjunction	Product type
Disjunction	Sum type
True formula	Unit type
False formula	Bottom type

Table 7.1: Logic and type correspondence

But not only zero-order logic corresponds to type, even higher-order logic is isomorph to types. Quantification is formulated with dependent types. The correspondance between logic and types is stated in table 7.1.

This idea of the logic-type correspondence has been formulated by the logician William Alvin Howard and the mathematician Haskell Curry [How80] and is widely known as Curry-Howard isomorphism, proofs-as-programs correspondence or originally as formulæ-as-types correspondence.

Based on this logic system, the mathematician Martin Per-Löf formalised an intuitionistic type theory in [Mar84]. This theory as now being generalised to the *Logical Framework* LF.

7.2 Type Matching

Since the very beginning of object-oriented languages subtyping has been a key features which is one of the best known concepts and was a natural extension that works directly with inheritance and subsumtion. Parametric subtyping as a form of second-order subtyping is a straightforward extension to first-order subtyping as we have seen. But we have also seen that binary methods are a big problem with subtyping. To tackle such problems a concept called *matching* has been introduced. This matching got interpreted by Martín Abadi and Luca Cardelli as third-order subtyping [AC96a]. Matching does not support subsumtion but adds support for inheritance of binary methods and consequentally gives a different expression power which was missed in subtyping.

The notation in type theory for matching is similar to subtyping. Instead of $A <: B$ matching is written as $A <\# B$ and is read as A matches B. $\forall(X <\# A)B(X)$ consequently is a match-bounded quantification. But what does matching do what subtyping doesn't? In a subtyping environment we can ensure that if $n : A$ and $A <: B$ that $n : B$ as well. This rule of subsumtion is not given by matching. The matching $A <\# B$ does only guarantee that n provides the interface of B. However, transitivity is given as well. The loss of subsumtion has now an other pleasant side effect. An inherited binary method's parameter is allowed to have a contravariant change [AC96a].

8 Short Reflections

This paper has shown that today's general-purpose programming languages such as Java, C#, C++ etc., include type systems which are insufficient. Programmers then therefore need to have a possibility to escape this type system, performing unsafe operations; we saw this in the implementation of binary methods. As seen, statically typed programming languages give some certainty that a set of errors will not occur at runtime. Due to this they however suffer in expressiveness. In order to overcome this trade-off situation, type systems and according programming languages need to be developed which can both ensure safety as well as provide sufficiently expressiveness. With Scala a step has been made into such a direction, which is why we considered it useful to present this new and innovative language with its sophisticated type system.

The functional paradigma and the object-oriented paradigma are genuinely distinct; they require the programmer to think in completely different ways and are each formulated in their own specific core calculi. It has therefore been very interesting to see that these different core calculi, corresponding to those two diverse paradigms, can be translated into each other.

As a further interesting aspect going beyond the specific use of type systems in computer science we have seen that type systems are strongly related to mathematics. There we can use them for formal reasoning in logics as well as in automatic provers.

Bibliography

[AC96a] Martín Abadi and Luca Cardelli. On subtyping and matching. *ACM Trans. Program. Lang. Syst.*, 18:401–423, July 1996.

[AC96b] Martín Abadi and Lucu Cardelli. *A theory of objects*. Springer, New York, 1996.

[AS96] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, July 1996.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proc. OPPSLA'98*, October 1998.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, page 2208–2236. CRC Press, 1997.

[Gam95] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading Mass., 1995.

[Geu09] Herman Geuvers. Introduction to type theory. In *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, pages 1–56. Springer-Verlag, 2009.

[GF71] JY Girard and JE Fenstad. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[Gro95] Ph De Groote. *The Curry-Howard isomorphism*. Academica, Louvain-la-Neuve, 1995.

[Gun94] Carl Gunter. *Theoretical aspects of object-oriented programming : types, semantics, and language design*. MIT Press, Cambridge Mass., 1994.

[How80] William A Howard. The formulae-as-types notion of construction. page 479–490, London, 1980. Academic Press.

[IPW99] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and GJ. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 23:132—146, 1999.

[Irv95] A. D Irvine. Russell's paradox. http://plato.stanford.edu/entries/russell-paradox/, December 1995.

[Lea07] Andres Löh and et al. Simply easy! an implementation of a dependently typed lambda calculus. 2007.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Naples, 1984.

[Mar96] RC Martin. The liskov substitution principle. 1996.

[McK06] James McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

[Naf07] Maurice Naftalin. *Java generics and collections*. O'Reilly, Beijing ;;Sebastopol CA, 2007.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction*. Clarendon Press, 1990.

[Ode08] Martin Odersky. *Programming in Scala*. Artima, Mountain View Calif., 1st ed., version 9 edition, 2008.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. *IN PROC. 24TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, pages 146—159, 1997.

[Pie91] Benjamin Pierce. *Basic category theory for computer scientists*. MIT Press, Cambridge Mass., 1991.

[Pie02] Benjamin Pierce. *Types and programming languages*. MIT Press, Cambridge Mass., 2002.

[Pie05] Benjamin Pierce. *Advanced topics in types and programming languages*. MIT Press, Cambridge Mass., 2005.

[Rey74] John C Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, page 408–423, London, UK, 1974. Springer-Verlag.

[Set06] Anton Setzer. Object-Oriented programming in dependent type theory. 2006.

[SH10] Michael Schinz and Philipp Haller. A scala tutorial for java programmers, November 2010.

[Sim02a] Anthony J.H. Simons. The theory of classification part 1: Perspectives on type compatibility. *Journal of Object Technology*, 1(1):55–61, May 2002. (column).

[Sim02b] Anthony J.H. Simons. The theory of classification, part 2: The Scratch-Built typechecker. *Journal of Object Technology*, 1(2):47–54, July 2002. (column).

[Sim02c] Anthony J.H. Simons. The theory of classification, part 4: Object types and subtyping. *Journal of Object Technology*, 1(5):27–35, November 2002. (column).

[Sim03] Anthony J.H. Simons. The theory of classification, part 8: Classification and inheritance. *Journal of Object Technology*, 2(4):55–64, July 2003. (column).

[Wam09] Dean Wampler. *Programming Scala*. O'Reilly, Sebastopol CA, 2009.