This paper is part of a series of papers on the subject of type systems and is a result of a seminar work conducted at the University of Applied Sciences Rapperswil. The purpose of this paper is to give an introduction to the polymorphic lambda calculus. The first part intends to motivate the reader and provides a brief historical retrospection. The second part points out important technical aspects of the calculus, supported by examples in known programming languages such as Haskell, Go, C, C++, Java and C#. The last part points out some possible extensions to the calculus and provides conclusional thoughts about its usefulness in modern day language design.

1 Motivation
Most modern programming languages support some of the various types of polymorphism. Object oriented languages mostly allow subtyping and function overloading, whereas functional programming languages allow generic functions and data types. This second type of polymorphism, which allows generic programming, is called parametric polymorphism. It is also what separates the simply typed lambda calculus ($\lambda\rightarrow$) from the polymorphic lambda calculus ($\lambda2$).

$\lambda2$ is essentially a straightforward extension of $\lambda\rightarrow$, adding parametric polymorphism to the core of the calculus. The reader is therefore expected to be familiar with the concepts of $\lambda\rightarrow$ and also the untyped lambda calculus ($\lambda$). Introductions to both topics can be found in [Syf15] and [Bie15] for $\lambda\rightarrow$ and $\lambda$, respectively.

$\lambda2$ is also referred to as second-order lambda calculus or System F when looking at it in the context of proof theory.

2 Introduction
Jean-Yves Girard was the first to discover System F in 1972 by extending $\lambda\rightarrow$ to allow abstraction on types [GTL89]. Having his background in logic, he approached the subject in the context of proof theory [Pie02, p.341]. He was seeking an analogy between types and propositions based on the work of Curry and Howard [Rey90]. The original papers introducing System F are [Gir71] and [Gir72].

Only two years later, in 1974, John C. Reynolds discovered the same type system independently, calling it the polymorphic lambda calculus [Rey74]. As a computer scientist, he approached the subject from a programming language design perspective. His goal was to extend conventional typed programming languages to permit the definition of polymorphic procedures. This is stated in a paper published in 1990 [Rey90], where Reynolds also remarks:

It is extraordinary that essentially the same programming language was formulated independently by the two of us, especially since we were led to the language by entirely different motivations.
That this is not a coincidence is for example shown in [B+91], where Barendregt describes the positioning of $\lambda 2$ in the so called $\lambda$-cube (shown in Fig. 1), which builds up the theory of constructions ($\lambda C$) introduced by Coquand and Huet. The eight type systems building the cube are ordered by inclusion, hence the directed edges. A corresponding cube can be used to show how higher order predicate logic is built up.

The vertices of the cubes correspond to each other via the Curry-Howard isomorphism as shown in [Gri15]. $\lambda 2$ is therefore corresponding to second-order intuitionistic logic. The types in $\lambda 2$ are equivalent to propositions quantified at the second order, and universal abstraction and application correspond with quantifiers.

Note that this paper is only concerned with the aspect of computer science and every aspect of logic is completely omitted.

### 3 Extending $\lambda \rightarrow$ to Obtain Polymorphism

The simply typed lambda calculus is very expressive, especially when enriched with Church encodings as discussed in [Bie15]. However, the fact that every expression has exactly one type can lead to repetitive writing of essentially identical function bodies, when needed for different types. The following example shown in Fig. 2 is taken from [Pie02, p.342] and shows a doubling function, which takes a function as its first argument and applies it twice to the second argument:

1. doubleNat = $\lambda f: \text{Nat} \rightarrow \text{Nat}. \lambda x: \text{Nat}. f (f x)$;
2. doubleRcd = $\lambda f: \{ \text{bool} \} \rightarrow \{ \text{bool} \}. \lambda x: \{ \text{bool} \}. f (f x)$;

Fig. 2. Doubling functions in $\lambda \rightarrow$

The same example can be made in C, where generic functions are not a feature of the language. One would have to implement the above doubling functions as shown in Listing 1:

```c
#include <stdio.h>
#include <stdlib.h>

// include standard library macro for bool
#include <stdbool.h>

int doubleNat(int (*f)(int), int x){
    return f(f(x));
}

bool doubleRcd(bool(*f)(bool), bool x){
    return f(f(x));
}

// example function for call with doubleNat
int incr(int x) {
    return ++x;
}

// example function for call with doubleRcd
bool neg(bool x) {
    return !x;
}

int main(void) {
    int i = doubleNat(incr,5);
    bool b = doubleRcd(neg,true);
    printf("int i: %d; bool b: %d",i,b);
    // int i: 7; bool b: 1
    return 0;
}
```

Listing 1. Doubling functions in C

Note that the type bool does not exist in C. However, in C99 a native type `bool` has been introduced that holds either 0 or 1. To use the keyword `bool` and the values `true` and `false`, one can include a standard library macro (line 6), which allows exactly that. This is legitimate, because the type bool can be introduced to $\lambda \rightarrow$ by extension. Pierce shows this in [Pie02, p.58ff] when introducing Church booleans to the untyped lambda calculus. One might also remark that macros can be used to define generic functions. This is not entirely true, since macros use pure text substitution and are not at all type safe. To
be able to call the doubling functions (lines 28, 29), two simple functions have been defined (lines 17, 22) matching the types of the first argument of the corresponding double function. It can be observed that both functions have the exact same bodies (line 8, 12).

Next to C there are other languages that do not support generic functions. In the language Go the above example looks as shown in Listing 2:

```go
package main

import "fmt"

func doubleNat(f func(int) int, x int) int {
    return f(f(x))
}

func doubleRcd(f func(bool) bool, x bool) bool {
    return f(f(x))
}

func main() {
    i := doubleNat(incr, 5)
    b := doubleRcd(neg, true)
    fmt.Printf("int i: %v; bool b: %t\n", i, b)
}
```

Listing 2. Doubling functions in Go

From a software engineering point of view this represents a violation of the abstraction principle and is usually resolved by introducing generics. A simple definition of the abstraction principle is given by Pierce in [Pie02, p.339]:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

The rest of this section aims to explain how the core of \( \lambda \rightarrow \) can be extended in such a way, that parametric polymorphism can be achieved. For better understanding, a brief definition of parametric polymorphism is given in the following subsection.

### 3.1 Parametric Polymorphism

Before starting to extend \( \lambda \rightarrow \) one needs to fully understand the meaning of parametric polymorphism. For that purpose, two different definitions are given in the following. The first definition is from Suter and is taken from his paper on polymorphism [Sut15]:

> In programming the abstraction principle aims to avoid duplicate code by abstracting out the varying parts. In some programming languages it is also possible to abstract out the types in order to write generic functions and datatypes. In type system parlance this is called parametric polymorphism. However, most programmers know it as Generics.

This definition gives a good idea of what is meant with parametric polymorphism. Note that the term generics is mostly used when talking about object oriented languages, where parametric polymorphism is the term used when talking about functional programming languages. In all following sections those two terms will be used synonymously. The second definition is from Pierce and is taken from [Pie02, p.340]:

> Parametric polymorphism allows a single piece of code to be typed generically, using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same.

To round off this subsection, a simple example is given in Listing 3, showing the use of generics in Java:

```java
public class Main {

    // generic print function
    private static <T> void print(T t) {
        System.out.println(t);
    }
}
```

Listing 3. Generic print function in Java
public static void main(String[] args) {
    print(100); // 100
    print("Para poly"); // Para poly
    print(true); // true
    print(new Object()); // java.lang.Object@15db9742
}

Listing 3. Generic print function in Java

Of course this example is not useful in practice, because the println() method in Java is already generic by itself. However, the example shows the syntax of how to write generic methods in general. Note that the example uses universal quantification over types, which is the most general form of parametric polymorphism. This means that no restrictions are made regarding the type that is passed to the generic parameter. There are other, more restrictive forms of parametric polymorphism, which are not relevant to the topic of this paper and are therefore not further mentioned.

What is important at this point, is to think about the consequences of the absence of generic methods. Regarding the code given in Listing 3, one would have to write four different versions of the print method, each corresponding to one of the given argument types. Also, it is easily imaginable, that a print method for all sorts of other types might be needed in other contexts and as a consequence would have to be implemented separately as well.

The need for an extended type system or calculus arises from the problem stated above. The following subsection explains in detail how to extend the core of $\lambda\to$, so that parametric polymorphism can be achieved.

### 3.2 Quantification Over Types

The notation of the syntax and the evaluation and typing rules describing the calculus in this subsection are taken from [Pie02].

Before getting into the details of the following implementation, it is strongly recommended to remind oneself of the core mechanisms of $\lambda\to$, as these are considered given. An overview of the core syntax and the corresponding evaluation and typing rules can be found in [Pie02, p.103] or [Syf15].

The goal of the extension in this subsection is to allow functions to be instantiated with arbitrary types. This is why in addition to the abstraction and application of terms, the polymorphic lambda calculus introduces abstraction and application over types. This allows abstracting types out of terms and fill them in at a later point in time.

Type abstraction is written $\lambda X.t$, where the parameter $t$ is a type. To differentiate between term and type abstraction, the type variable is written as an uppercase $X$. Type application is written $t [T]$, in which the argument $T$ is a type expression.

Analogous to the reduction rule for term abstraction and application $E$-APPABS, a new reduction rule $E$-TAPPAPPS has to be added to the core of $\lambda\to$:

$$(\lambda X. t_{12}) [T_{2}] \rightarrow [X \rightarrow T_{2}]t_{12} \quad (E\text{-TAPPAPPS})$$

This rule is needed to evaluate type abstractions when confronted with a conforming type application. A second reduction rule $E$-TAPP has to be introduced, corresponding to rule $E$-APP1 of $\lambda\to$:

$$t_{1} \rightarrow t_{1}' \quad t_{1} [T_{2}] \rightarrow t_{1}' [T_{2}] \quad (E\text{-TAPP})$$

It is obvious that these newly introduced rules only make sense, when types can be generic. The next step is therefore to extend the syntax of $\lambda\to$ with a universal type, for being able to type these polymorphic abstractions. This is written $\forall X.t$.

Since types are now handled like terms, the binding of type variables to contexts has to be allowed as well. Corresponding to term variable binding, type variable binding is written $\Gamma, X$, again using an uppercase $X$.

The last step to make parametric polymorphism possible is to introduce typing rules for polymorphic abstraction $T$-TABS and application $T$-APP. These are shown below:
The rule T-TAPP shows the type that results when a generic function is applied to a type argument [Sut15]. Note that T-TABS has the side condition, that every type variable bound to the context has to be named differently. Variables that do not meet this condition may be renamed at will in order to satisfy it.

Combining all the additions mentioned above into the core of $\lambda \to$, it is now possible to specify generically typed functions. An overview of the complete core is given in [Pie02, p.343].

To round off this subsection, a simple example shall be given. The most common example is the generic identity function shown in Fig. 3:

\[ id = \lambda X. \lambda x:X. x; \]

Fig. 3. Generic identity function

The type of the given function is more interesting. Here the newly introduced universal type notation is used, which does nothing else than putting the universal quantifier before the type variable. Therefore, the type of the polymorphic identity function looks as shown in Fig. 4:

\[ \forall X. X \to X \]

Fig. 4. Type of the generic identity function

Using the reduction rule E-TAPPABS with (for example) the type $\text{Nat}$, written $id \ [\text{Nat}]$, the type variable $X$ is replaced by the concrete type $\text{Nat}$, which results in $[X \to \text{Nat}] \ (\lambda x:X. x)$. Simplified, the result looks as expected and is shown in Fig. 5:

\[ \lambda x : \text{Nat}. x \]

Fig. 5. Identity function over natural numbers

Note that $\text{Nat}$ is not by default included in the core of the calculus, but is a Church extension of the typed lambda calculus for representing the natural numbers. As per definition, the expected type for the identity function over the natural numbers is obtained as shown in Fig. 6:

\[ \text{Nat} \to \text{Nat} \]

Fig. 6. Type of the identity function over natural numbers

The extension of $\lambda \to$ with the above rules form a more expressive, but also more restrictive calculus called the polymorphic lambda calculus or $\lambda 2$, which is exactly what Girard and Reynolds discovered. The following subsection shows some examples of how modern programming languages implement parts or even the whole core functionality of $\lambda 2$.

3.3 $\lambda 2$ in Modern Languages

Parametric polymorphism is part of many modern programming languages. A first example was already given in Listing 3, where a generic method for printing on the console in Java was shown. Another example is the definition of the generic doubling function as seen in Fig. 2 in the functional programming language Haskell. This is shown in Listing 4:

```
1 2 -- defined in a .hs file
3 4 -- generic doubling function
5 double :: (a -> a) -> a -> a
6 double f x = f (f x)
7 8 -- example function for call with int
9 incr :: Int -> Int
10 incr x = x + 1
11 12 -- example function for call with bool
13 neg :: Bool -> Bool
14 neg b = not b
15 16 -- use in terminal or command line
17 18 -- call double with incr
19 double incr 5 -- 7
```
Listing 4. Doubling function in Haskell

The code only defines one function with name `double` (line 5) and assigns a generic type \((\text{a} \rightarrow \text{a}) \rightarrow \text{a} \rightarrow \text{a}\) to it. The exemplary functions `incr` and `neg` are also defined as in Listing 1 (lines 9, 13). As one can see, the function `double` can be called with `incr` and an Int typed argument as well as with `neg` and a Bool typed argument (lines 19, 22, 23).

Analogous to the generic identity function shown in Fig. 3, the doubling function can now be expressed generically in \(\lambda 2\). The definition of the double function is given in Fig. 7:

\[
\text{double} = \lambda X. \lambda f:X \rightarrow X. \lambda a:X. f (f a);
\]

Fig. 7. Generic double function

The type of the double function is shown in Fig. 8 and is not surprising at all. It corresponds directly with the type annotation that was set for the `double` function (line 5) of Listing 4:

\[\forall X. (X \rightarrow X) \rightarrow X \rightarrow X\]

Fig. 8. Type of the generic double function

Using the reduction rule E-TAPPABS given in subsection 3.2, arbitrary types can be applied to replacing the universal type variable \(X\).

To make clear that not only functional programming languages implement concepts of \(\lambda 2\), more examples shall be given. It is easily possible to implement the same example in C++ using templates as shown in Listing 5:

As in Haskell, the calls with type mismatches are recognized during compile time. If all types match during compile time, the C++ compiler instantiates the `doubleFunction` separately for every call with different types of arguments [Sut15]. In Listing 5, `doubleFunction` gets instantiated twice.

Since Java 8, the generic doubling function can also be implemented in Java, using the `Function` interface. This allows to write code that resembles the syntax usually used to declare higher order methods. The code then looks as shown in Listing 6:

Listing 5. Doubling function in C++

Listing 6. Doubling function in Java

As in Haskell, the calls with type mismatches are recognized during compile time. If all types match during compile time, the C++ compiler instantiates the `doubleFunction` separately for every call with different types of arguments [Sut15]. In Listing 5, `doubleFunction` gets instantiated twice.

Since Java 8, the generic doubling function can also be implemented in Java, using the `Function` interface. This allows to write code that resembles the syntax usually used to declare higher order methods. The code then looks as shown in Listing 6:

Listing 6. Doubling function in Java
public static void main(String[] args) {
    int i = doubleFunction(incr, 5);
    boolean b = doubleFunction(neg, true);
    System.out.println("int i: "+ i + " ; boolean b: " + b);
    // int i: 7; boolean b: true
}
}

Listing 6. Doubling function in Java

What stands out in this example, is that methods cannot be implemented in the usual way (line 13, 21), if the usage in a higher order method is required. Instead, the interface `Function` has to be implemented which wraps the needed method in a object, which then can be passed as an argument to other methods. Higher order methods are not per se a feature of Java, although they are part of the core of λ2. In fact, it is not uncommon that programming languages only implement parts of the core of λ2 or extend it to implement additional functionality like subtyping, for example.

In addition, it is worth mentioning that, as in C++, generics in Java are typechecked at compile time. The main difference is, that in Java all generic type information is lost during compile time, because all the information gets erased. The advantage of this approach is, that only one version of a generic class or function is compiled and then used for all types the generic parameter is instantiated with [GJS+14].

A last example shown in Listing 7 shows how generic methods can be implemented in C#. Note that C#, unlike Java, supports higher order functions the same way as C++ and Haskell do.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Main
{
    class Program
    {
        // generic doubling method
        static T doubleFunction<T>(
            Func<T,T> f, T x)
        {
            return f(f(x));
        }
        // example method for call with int
        static int incr(int x)
        {
            return x + 1;
        }
        // example method for call with bool
        static bool neg(bool b)
        {
            return !b;
        }
        static void Main(string[] args)
        {
            int i = doubleFunction(incr,5);
            bool b = doubleFunction(
                neg, true);
            Console.WriteLine("int i: {0}; bool b: {1}",
                i, b);
            // int i: 7; bool b: True
        }
    }
}
```

Listing 7. Doubling function in C#

Finally, one might have noticed, that none of the examples given in this subsection show generic data types. An example of how a polymorphic list can defined is given in [Pie02, p.345ff]. Other generic data types can be defined accordingly.

4 Properties of λ2

The extensions introduced in subsection 3.2 result in a higher expressiveness of the calculus. The question that arises is whether the basic properties proven for λ→ in [Pie02, p.95ff] are still valid in λ2. The following subsections show, how types can defined to extend the core of the calculus and how this affects the preservation, progress and normalization properties.

4.1 Defining Types

The first observation is, that altough the typing rules for λ2 differ from the ones of λ→, it is still possible to define all usual data types like integers, booleans, lists, etc. by using only the core of the calculus. In [GTL89], Girard shows how to define boolean and integer types, as well as generic lists and binary trees. The Church encodings for booleans, numerals and lists along with their most common functions are described in [Pie02, p.347ff].

A simple example shall be given, which shows
how Church numerals can be defined within $\lambda_2$. The example is taken from [Pie02, p.348f] and further referencing is omitted. Given is the definition of Church numerals as encoded for the untyped lambda calculus as shown in Fig. 9:

\[
\begin{align*}
c_0 &= \lambda s. \lambda z. z; \\
c_1 &= \lambda s. \lambda z. s z; \\
c_2 &= \lambda s. \lambda z. s (s z); \\
c_3 &= \lambda s. \lambda z. s (s (s z)); \\
&\quad \ldots
\end{align*}
\]

Fig. 9. Untyped Church numerals

To embed the above encoding in $\lambda_2$, it is necessary to provide suitable type annotations to each term. If the type of argument $z$ is $T$, then $s$ should have type $T \rightarrow T$ and the return value of $s$ should also be of type $T$. Therefore, the resulting type of each numeral must be as shown in Fig. 10:

\[
\text{CNat} = \forall X. (X \rightarrow X) \rightarrow X \rightarrow X;
\]

Fig. 10. Type of Church numerals in $\lambda_2$

After adding the corresponding type annotations to the arguments $s$ and $z$, the type $\text{CNat}$ is applied to the initially untyped numerals. The resulting typed numeral encodings are shown in Fig. 11:

\[
\begin{align*}
c_0 &= \lambda X. \lambda s:X \rightarrow X. \lambda z:X. z; \\
c_1 &= \lambda X. \lambda s:X \rightarrow X. \lambda z:X. s z; \\
c_2 &= \lambda X. \lambda s:X \rightarrow X. \lambda z:X. s (s z); \\
c_3 &= \lambda X. \lambda s:X \rightarrow X. \lambda z:X. s (s (s z)); \\
&\quad \ldots
\end{align*}
\]

Fig. 11. Typed Church numerals

Extending the core of $\lambda_2$ with a Church boolean encoding or other types is just as simple and straightforward as the above example. Overall, it can be stated that every expressible type in $\lambda_\to$ can also be defined in the polymorphic lambda calculus. How this affects the properties of the core is described in the following subsection.

4.2 Preservation, Progress and Normalization

Although it might seem surprising at first, it is relatively easy to show that the preservation of types and the progress property stay valid for $\lambda_2$. The proofs for these properties are straightforward extensions of the ones shown in [Pie02, p.104ff] for $\lambda_\to$.

However, the proof for the normalization of terms in $\lambda_2$ is far more complicated. Luckily, Girard managed to achieve exactly this. He showed, that every expression of the polymporphic lambda calculus possesses a normal form, which states that every such expression describes a terminating computation [Rey90].

5 Type Reconstruction

So far, every example given in this paper compiles and is executed without runtime errors. At first sight, this does not seem to come as a surprise, because the examples are fairly simple. When thinking about it a little longer, one question has to be raised and this is best shown on the generic doubling function defined in Listing 4. Listing 8 shows the extracted function:

\begin{verbatim}
1  -- generic doubling function
2  double :: (a -> a) -> a -> a
3  double f x = f (f x)
\end{verbatim}

Listing 8. Implicitly typed doubling function

Note that the function heading only contains unknown, yet to be determined type variables, but completely omits type parameters. When calling the function $\text{double}$ with the $\text{incr}$ function and
5 as its arguments, type applications are also omitted. This is shown in Listing 9:

```cpp
-- call double with incr
double incr 5 -- 7
```

Listing 9. Call of doubling function

Since C++14, it is also allowed to write lambda functions with parameters declared using the `auto` type specifier. The corresponding code example is shown in Listing 10:

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

// generic lambda function
auto doubleFunction = [](auto f, auto x) {
    return f(f(x));
};

// example function for call with int
int incr(int x) {
    return ++x;
}

// example function for call with bool
bool neg(bool x) {
    return !x;
}

int main() {
    int i = doubleFunction(incr,5);
    bool b = doubleFunction(neg, true);
    cout << "int i: " << i << "; bool b: " << std::boolalpha << b;
    // int i: 7; bool b: True
}
```

Listing 10. Generic lambda function in C++

What is the impact of using this so called implicit form of parametric polymorphism? Cardelli pointed this out in a paper published in 1987 [Car87]:

> Implicit polymorphism can be considered as an abbreviated form of explicit polymorphism, where the type parameters and applications have been omitted and must be rediscovered by the language processor. Omitting type parameters leaves some type-denoting identifiers unbound; and these are precisely the type variables.

The above examples show, that in most languages, some type information can be omitted and left for the compiler to reconstruct. In some languages it is even possible to completely omit type annotations. How is it possible to hand over the responsibility of finding a suitable type from the programmer to the compiler? The rest of this subsection gives a brief answer to this question.

The problem of finding a type for a term within a given type system is called type inference. Is every term in a given untyped, the problem is called type reconstruction and is even harder to solve than type inference [Car96].

When analyzing the untyped lambda calculus, type reconstruction is solvable within \( \lambda \to \). This means that it is possible to reconstruct all possible types in \( \lambda \to \), when type annotations are completely omitted. Note that solvable means that either the types are found or the reconstruction fails when faced with invalid terms. The first algorithm solving this problem is called the Hindley-Milner Algorithm, which is used in ML [Mil78]. This sounds promising, since so far every property holding in \( \lambda \to \) remained valid when looking at \( \lambda \). Unfortunately, this is not the case when trying to solve type reconstruction for the untyped lambda calculus within \( \lambda \). This was formally stated in a theorem by Wells in 1994 in [Wel94]. Pierce even states, that various forms of partial type reconstruction also lack decidability regarding \( \lambda \) [Pie02, p.354]. But how is it still possible, that a lot of languages allow implicit polymorphism or even to omit any form of type annotations?

In practice, algorithms for type reconstruction within \( \lambda \) have been implemented. Theoretically, these algorithms are undecidable. However, the cases where they actually diverge are caused by ill typed programs and are very unlikely. This makes the algorithms practically applicable [Car96].

6 Relevancy

Every expression formed within the polymorphic lambda calculus describes a terminating computation. Intuitively, one can imagine that there must be some computable functions that cannot be expressed by the calculus. Reynolds describes exactly this in [Rey90], but then comments:

> Yet the polymorphic typed lambda calculus is just such a language, in which one
can express almost anything that one might actually want to compute. Indeed, Girard has shown that every function from natural numbers to natural numbers that can be proved total by using second-order arithmetic can be expressed in the calculus.

Later in the same paper, Reynolds even summarizes:

The polymorphic typed lambda calculus is far more than an extension of the simply typed lambda calculus that permits polymorphism. It is a language that guarantees the termination of all programs, while providing a surprising degree of expressiveness for computations over a rich variety of data types.

But λ2 has its limitations. One can argue that, only because a function might be expressed within λ2, it does not mean that this is the best way of expressing it. The calculus also forbids lazy calculations on infinite data structures, which are for example possible in Haskell.

It is certainly possible to extend the core of λ2. By doing this, it is possible to achieve features like lazy computation, subtyping, ad hoc polymorphism and so on. However, one has to be aware that in most cases, extending λ2 can only be achieved with the cost, that the guarantee of termination is lost. In practice, the pure form of the polymorphic lambda calculus is not expressive enough to define meaningful programming languages.

7 Conclusion

The intention of this paper was to introduce the reader to the concepts of the polymorphic lambda calculus. To achieve that, it was shown how the core of the simply typed lambda calculus can be extended. A universal type has been introduced with which it is possible to achieve genericity. Additional rules for type abstraction and application have been introduced to make the universal type compatible with the other rules given in the calculus. A simple example was given to demonstrate the functionality of λ2.

By analyzing code examples, it was shown that the calculus is not only relevant for studying purposes, but plays an essential role when designing the type systems of almost every modern programming language.

The paper also gave insight of how the pure calculus can be extended by additional types. This was demonstrated by showing the necessary steps to introduce Church numerals and their corresponding functions.

A brief section about type reconstruction was given to make clear that type annotations cannot be omitted without consequences, because type reconstruction of the untyped lambda calculus is not solvable within λ2.

The last section of the paper described the limitations of the polymorphic lambda calculus and its relevancy in a practical point of view. Most programming languages therefore implement only specific parts of the core functionality or an extended calculus for which the normalization property does not hold anymore.

Considering all these different aspects of λ2, the summarized conclusion of this paper can be stated as follows:

The polymorphic lambda calculus is a highly useful instrument to study and to lecture parametric polymorphism and its application in the design of programming languages. The fact that almost none of the modern languages implement the pure calculus suggests that λ2 without any form of extension is almost irrelevant in a practical context.

References


[Car96] Luca Cardelli. Type systems. 1996.


