SEMINAR REPORT
Seminar: Programming Languages in Winter term 2024/2025

Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Idris2

Mehrdad Shahidi, Zahra Khodabakhishan
Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, Department of Computer Science

Note: *This report is a compilation of publications related to some topic as a result of a student seminar. It does not claim to introduce original work and all sources should be properly cited.*

Idris2 is a functional programming language that leverages Quantitative Type Theory (QTT) to offer an advanced dependent type system, making it well-suited for both practical software development and formal proof verification. This report provides an example problem verification using Idris2 to showcase its capabilities, and concludes by summarizing key insights.

1 Introduction

As software systems become increasingly complex and critical to our daily operations, programming languages continue to evolve to meet growing demands for reliability and correctness. Idris2 represents a significant advancement in this evolution by offering a unique approach to program correctness through its type system based on Quantitative Type Theory (QTT). As a pure functional programming language with first-class types, Idris2 allows types to be manipulated and computed just like any other value, fundamentally shifting how we approach programming. Unlike traditional programming languages that catch errors at runtime, Idris2 enables developers to work collaboratively with the compiler, which acts as an assistant during development, helping to prove properties about code and suggest potential solutions for incomplete programs. This type-driven development approach ensures program correctness before execution, making it particularly valuable for critical systems, mathematical proofs, and complex state-dependent applications like concurrent systems [4].
example of idris2 code:

```
safeDiv : (num : Nat) -> (d : Nat) -> {auto ok : GT d Z} -> Nat
safeDiv n d  = div n d

-- This works automatically because Idris can prove 0 < 5
example : Nat
example = safeDiv 12 5

-- This code failes to compile because Idris can't prove 0 < 0
example2 : Nat
example2 = safeDiv 12 0
```

In the subsequent sections, this report explores several key aspects of Idris 2 We begin with a brief overview of the language's background and its evolution from Idris 1 to Idris 2. Next, we delve into the theoretical underpinnings of dependent types, focusing on propositions, judgments, and

equality in Idris2. We then discuss the features of Idris2, including Quantitative Type Theory (QTT), multiplicities, and linearity, which enable more precise resource management and control over side effects. The report also highlights Idris2's interactive development environment, which supports type-driven development and interactive editing features. Then we prove a verification example using Idris2 to demonstrate its practical application in formal verification. Finally, we share insights on what Idris2 offers, and our challenges we faced during the verification process.

2 Background

Programming languages traditionally separate types and values, which means that certain kinds of errors, particularly those related to invalid values or incorrect data structures, are only caught at runtime. This limitation has been a significant challenge in software engineering, as it allows many types of errors—such as accessing an element from an empty list—to slip through until the program is executed. The goal of dependent types is to solve this problem by allowing types to depend on values, enabling the type system itself to enforce correctness at compile time.

The concept of dependent types is not new; it has roots in mathematical logic and type theory. However, its application in programming languages was limited for many years. Idris, first introduced by Edwin Brady in 2009, was a pioneering language that sought to make dependent types practical and usable in real-world programming. Idris was designed with the goal of enabling programmers to capture more precise specifications in the type system itself, making programs both more expressive and safer. Through its use of dependent types, Idris allowed for the formal verification of certain aspects of a program at compile time—such as proving that a vector is non-empty before calling the head function.

Despite its potential, Idris 1 had its limitations. The performance of type checking was not optimal for large programs, and the language's implementation and theoretical foundations were not as robust as they could be. These shortcomings were particularly evident when trying to use Idris for large-scale or more complex applications. The need for improvement in both usability and performance led to the development of Idris 2.

Idris 2 represents a complete redesign of the language, drawing on the lessons learned from Idris 1. One of the key challenges in the original Idris was the efficiency of type checking, especially when dealing with large programs. Idris 2 addressed this by significantly improving its type checking performance. Furthermore, the language introduced the ability to erase compile-time arguments more effectively, reducing the overhead in generating executable code. Additionally, Idris 2 incorporated linear types through Quantitative Type Theory, providing a mechanism for better resource management and more precise control over side-effects in functional programming.

These improvements made Idris 2 a more practical and scalable tool for developers looking to leverage dependent types.

3 Theorem Proving

3.1 Propositions and Judgments

Before delving into theorem proving, it is essential to understand the foundational framework of constructive logic. Also known as intuitionistic logic, it differs from classical logic by rejecting the Law of Excluded Middle—which asserts that every proposition is either true or false. Instead, constructive logic considers a proposition true only if it can be explicitly proven. Thus, an unproven proposition is not necessarily false; it may simply be unprovable.

Constructive logic builds a database of judgments, where each judgment is a formally validated proof. This approach ensures the logical system's integrity by requiring concrete evidence for every proposition. For instance, proving a number is even involves explicitly providing a number that satisfies the condition, rather than merely asserting it. By avoiding assumptions like the Law of Excluded Middle, constructive logic maintains consistency and reliability, ensuring that only provable propositions are included in the system.

3.2 Equality

3.2.1 definitional and propositional equality

Equality in Idris is captured by the Equal type, defined as follows:

```
data (=) : a -> b -> Type where
Refl : x = x
```

This states that two values are equal if they are definitionally identical, with Reflexive serving as explicit evidence of their equality.

```
plusReducesL : (n : Nat) -> plus Z n = n
plusReducesL n = Refl
```

This proposition asserts that adding zero to any number n results in n, which is definitionally true.

```
plusReducesR : (n : Nat) -> plus n Z = n
plusReducesR n = Refl
```

This is not definitionally true, because the plus function is defined recursively on its first argument. While adding zero as the second argument reduces for specific cases, the reduction is not immediate or definitional. As a result, Reflexive cannot prove plusReducesR directly.

In Idris, an equality type that can be proved using Reflexive alone is known as definitional equality. For cases like plusReducesR, where Reflexive is insufficient, additional techniques, such as propositional equality or rewriting, may be required.

3.2.2 Heterogeneous Equality

Equality in Idris is heterogeneous, meaning that it allows for the possibility of proposing equalities between values of different types. It means suggesting that two values of different types can be considered equal under certain assumptions or relationships. It becomes essential when working with dependent types, where types themselves depend on values.

```
vect_eq_length : (xs : Vect n a) -> (ys : Vect m a) ->
(xs = ys) -> n = m
```

(xs = ys) asserts that the two vectors are equal in both value and type.So if xs = ys, then n and m must also be equal, because the length of the vector (n or m) is part of its type. This is why the function can safely return n = m.

3.2.3 Substitutive Property (Leibniz Equality)

Idris also supports reasoning about equality using the substitutive property, often referred to as Leibniz Equality. It formalizes the idea that if two values are equal, they are indistinguishable under any property. In Idris, this can be expressed as:

```
leibnizEq : (a : A) -> (b : A) -> Type
leibnizEq a b = (P : A -> Type) -> P a -> P b
```

4 Features of Idris2

4.1 Introduction to Quantitative Type Theory (QTT)

Quantitative Type Theory (QTT) extends traditional dependent type theory by adding explicit tracking of how variables are used in programs [1]. Developed by Conor McBride, it addresses inefficiencies in traditional type theory, where variables serve both as type constructors and computational entities. By explicitly modeling variable usage, QTT improves efficiency, especially in resource management like memory.

4.2 Multiplicities

In QTT, each variable binding is associated with a quantity (or multiplicity) which denotes the number of times a variable can be used in its scope: either zero, exactly once, or unrestricted [1].

```
0: The variable is used only at compile time and erased at runtime
1: The variable must be used exactly once at runtime (linear)
ω: The variable can be used any number of times (unrestricted)
```

Multiplicities in Idris 2 describe how often a variable must be used within the scope of its binding. A variable is considered "used" when it appears in the body of a definition, as opposed to a type declaration, and when it is passed as an argument with multiplicity 1 or ω. The multiplicities of a function's arguments are specified by its type, and they dictate how many

times the arguments can be used within the function's body. Variables with multiplicity ω are considered unrestricted: they can be passed to argument positions with multiplicities 0, 1, or ω. On the other hand, a function that accepts an argument with multiplicity 1 guarantees that the argument will not be shared within its body in the future, although it is not required to ensure that it has not been shared in the past [3].

4.2.1 Lineaitiy Example

In Idris 2, linearity or multiplicity of 1 as we saw, is a property of values in the type system that enforces the idea that a value must be used exactly once. This concept is rooted in linear logic and ensures resources are managed precisely, preventing accidental duplication or omission.

We have a File System Protocol that allows us to open a file, close it and delete . The protocol enforces linearity to ensure that files are managed correctly. The idea is to use linearity to enforce that:

- A file cannot be opened if it's already open.
- A file cannot be closed if it's already closed.
- A file cannot be deleted unless it's closed.

A file can be in one of two states: Opened or Closed:

```
data FileState = Opened | Closed
```

We model the file's state in the type system:

```
data File : FileState -> Type where
  MkFile : (fileName : String) -> File st
```

the transitions between states:

```
openFile : (1 f : File Closed) -> File Opened
openFile (MkFile name) = MkFile name

closeFile : (1 f : File Opened) -> File Closed
closeFile (MkFile name) = MkFile name
```

Ensure that files are created and used linearly: newFile accepts a function that takes a Closed file, ensuring the file is used exactly once.

```
newFile : (1 p : (1 f : File Closed) -> IO ()) -> IO ()
newFile p = p (MkFile "example.txt")
```

Only files in the Closed state can be deleted :

```
deleteFile : (1 f : File Closed) -> IO ()
deleteFile _ = putStrLn "File deleted."
```

A protocol example can be seen below:

```
fileProg : IO ()
fileProg =
    newFile $ \f =>
        let f' = openFile f
            f'' = closeFile f' in
            deleteFile f''
```

This protocol demonstrates how linear types in Idris 2 enforce safe resource usage by modeling state transitions, preventing runtime errors, and ensuring predictable, reliable programs.

4.2.2 Erasure

Erasure in Idris2 determines which values are needed at runtime and which can be safely removed during compilation. Values marked with multiplicity 0 are used only during type checking and are erased from the final executable.

Here's a practical example:

```
data Vec : (n : Nat) -> Type -> Type where
Nil : Vec Z a
(::) : (x : a) -> (xs : Vec k a) -> Vec (S k) a

-- The length is available at compile time but erased at runtime
length : {0 n : Nat} -> Vec n a -> Nat
length {n} xs = n

-- This works because n is available at compile time
example : Vec 3 Integer
example = 1 :: 2 :: 3 :: Nil

-- This fails because m and n are erased (multiplicity 0)
badSum : Vec m a -> Vec n a -> Nat
badSum xs ys = length xs + length ys -- Error: m is not accessible

-- This works by making m, n available at runtime
goodSum : {m, n : _} -> Vec m a -> Vec n a -> Nat
goodSum xs ys = length xs + length ys
```

This example demonstrates how erasure helps optimize runtime performance while maintaining compile-time guarantees. The length information is checked during compilation but doesn't incur runtime overhead unless explicitly needed.

4.3 Interactive Development Environment

Idris2's type-driven development approach shines when paired with its editor integration. During development, the compiler acts as an assistant, guiding the programmer through the code and suggesting potential solutions for incomplete programs. This interactive workflow is facilitated by Idris2's hole-driven development, case splitting, and proof search features, which help programmers incrementally build correct programs by leveraging the type system. We used Neovim as our editor, which provides key bindings for common REPL commands, such as adding clause templates, case splitting, and showing type/context of symbols. These features enable a seamless development experience, allowing programmers to focus on writing correct code while the compiler handles the verification and proof process. The plugins is available on Github [7].

4.3.1 Core Workflow

Hole-driven development: Start with type signatures and placeholders

```
vzipWith : (a -> b -> c) -> Vect n a -> Vect n b -> Vect n c
vzipWith f xs ys = ?vzipWith_rhs
```

Then you can case split on the arguments to generate clauses:

```
-- Case split on xs
vzipWith f [] ys = ?vzipWith_rhs_1
vzipWith f (x :: xs) ys = ?vzipWith_rhs_2
--------------------------------
 0 c : Type
 0 b : Type
 0 a : Type
   ys : Vect 0 b
   f : a -> b -> c
 0 n : Nat
--------------------------------
vzipWit_0 : Vect 0 c
```

using the context and goal, we can fill the hole or use proof search to find the solutions if it is possible.

5 Verification Example

In this section, we tyring to prove correctness of a simple regular expression match function using Idris2. The function takes a regular expression and a string as input and returns a boolean indicating whether the string matches the regular expression. We define the regular expression type as using algebraic data types:

```
data Regex: (a: Type) -> Type where
Empty: Regex a
Epsilon: Regex a
Chr : a -> Regex a
Concat: Regex a -> Regex a -> Regex a
Alt:  Regex a -> Regex a -> Regex a
Star :  Regex a -> Regex a
```

Our regular expression can be defined for alphabets (values) of any type `a`. For simplicity, we will use `Char` in this example. The match function type signatures are as follows:

```
match : (r: Regex Char) -> (s: List Char) -> Bool
```

Our prove specification formula would be to show :

$$\forall r, s.s \in L(r) \Leftrightarrow \text{match } r \text{ s} = \text{True} \tag{1}$$

For this we need to capture the membership of a string in the language of a regular expression.

5.1 Language representation

from automata theory, we know that regular expressions is equivalent to formal languages. formal languages are usally represented in set theory as a set of strings. In type theory, we can represent languages as predicate on strings and using that we can show string membership in a language [5]. So we define Languages in Idris as follows:

```
Lang : (a: Type) -> Type
Lang a = List a -> Type

-- Empty language: no strings
public export
empty : Lang a
empty _ = Void

-- Universal language: all strings
public export
univ : Lang a
univ _ = Unit

-- Language contain empty string
public export
eps : Lang a
eps w = w = []

-- Single-token language
public export
tok : a -> Lang a
tok c w = w = [c]
```

```
-- Scalar multiplication
-- not sure when this is useful
public export
(:.:) : Type -> Lang a -> Lang a
(:.:) s l w = Pair s (l w)

public export
union: Lang a -> Lang a -> Lang a
union l1 l2 w = Either (l1 w) (l2 w)

public export
intersection: Lang a -> Lang a -> Lang a
intersection l1 l2 w = Pair (l1 w) (l2 w)

public export
exists : {a, b : Type} -> (p: (Pair a b) -> Type) -> Type
exists {a} {b} p =  DPair (a, b) p

public export
langConcat: {a: Type} -> Lang a -> Lang a -> Lang a
langConcat l1 l2 w =
  exists (\ (w1 , w2) => Pair (w = w1 ++ w2) (Pair (l1 w1) (l2 w2)))

public export
concat: {a: Type} -> List (List a )-> List a
concat = foldr (++) []

public export
langStar: {a: Type} -> Lang a -> Lang a
langStar l w  = DPair _ (\ws => Pair (w = concat ws ) (All l ws))
```

Now we can define the language of a regular expression as follows:

```
lang : {a: Type } -> Regex a -> Lang a
lang Empty = empty
lang Epsilon = eps
lang (Chr c) = tok c
lang (Concat x y) = langConcat (lang x) (lang y)
lang (Alt x y) = union (lang x) (lang y)
lang (Star x) = langStar (lang x)
```

and our correctness proof specification would be to prove the completeness and soundness of the match function:

```
  matchSoundness : (r: Regex Char) -> (s: List Char) -> match r s = True -> lang r s
  matchCompleteness: (r: Regex Char) -> (s: List Char) -> lang r s -> match r s = True
```

5.1.1 Implementation and Proof

Getting back to our match function, there are many way to implement it, we can use a simple recursive function to match the regular expression with the string. This implementation can make it is much easier to prove the correctness than other implementations of the function. For our example, we will use the following implementation:

```
match : Regex Char -> List Char -> Bool
match Empty str = False
match Epsilon [] = True
match Epsilon _ = False
match (Chr c) [] = False
match (Chr c) [x] = x == c
match (Chr c) (x1 :: x2 :: xs) = False
match (Concat r1 r2) str =
  matchConcatList r1 r2 (splits str) False
match (Alt r1 r2) str =
  match r1 str || match r2 str
match (Star r) [] =  True
match (Star r) (x::xs) =
  matchConcatList r (Star r) (appendFirst x (splits xs)) False
```

The match function is implemented using pattern matching on the regular expression and the input string. It recursively matches the regular expression with the string based on the regular expression's structure. The `matchConcatList` function is used to match the concatenation of two regular expressions with the string by splitting the string into all possible prefixes and suffixes and recursively matching the two regular expressions with the prefixes and suffixes. the implementation of `matchConcatList` and `splits` :

```
matchConcatList : (r1 : Regex Char) -> (r2 : Regex Char) ->
(s : List (List Char, List Char)) -> Bool -> Bool
matchConcatList r1 r2 [] acc = acc
matchConcatList r1 r2 ((s1, s2) :: xs) acc =
  (match r1 s1 \&\& (match r2 s2)) || (matchConcatList r1 r2 xs acc)

splits : List Char -> List (List Char, List Char)
splits [] = [([],[])]
splits (x :: xs) = ([], x :: xs) :: appendFirst x (splits xs)
```

Proceeding with the proof, we can start by proving the completeness of the match function. The completeness proof states that if a string is in the language of a regular expression, then the match function should return True. We can prove this by pattern matching on the regular expression:

```
matchCompleteness : (r: Regex Char) -> (s: List Char) ->
 lang r s -> match r s = True
matchCompleteness Empty s prf = absurd prf
```

```
matchCompleteness Epsilon s prf = ?matchCompleteness_rhs_1
matchCompleteness (Chr c) s prf = ?matchCompleteness_rhs_2
matchCompleteness (Concat r1 r2) s prf = ?matchCompleteness_rhs_3
matchCompleteness (Alt r1 r2) s prf = ?matchCompleteness_rhs_4
matchCompleteness (Star r) s prf = ?matchCompleteness_rhs_5
```

Starting from first case, as we define the Empty language as predicate that give us bottom type (Void), we can use absurd(ex-falso-quadlibet) to prove the case. for the Epsilon case, Idris2 give us enough information to prove just by simply rewrinting the `prf` in our goal.

```
s : List Char
  prf : s = []
------------------------------
matchCompleteness_rhs_1: match Epsilon s = True

-- using the context and goal
matchCompleteness Epsilon s prf = rewrite prf in Refl
```

for the `Chr` case, we need to further case split on the string `s` to have same cases as defined in the `match` function. for the cases where the string is `[]` or `(x1::x2::xs)`, we can use the `impossible` syntax to show that these cases are impossible (can be used instead of `absurd`). The singelton case `[y]` is the only case that we need to prove, as you see in the code below we needed a lemma to that equality of singelton list implies the equality of the elements.

```
matchCompletness (Chr x) [] prf impossible
matchCompletness (Chr x) (x1 :: x2 :: xs) prf impossible
matchCompletness (Chr x) ([y]) prf =  ?goal
------------------------------
x : Char
y : Char
prf : [y] = [x]
------------------------------
goal : intToBool (prim__eq_Char y x) = True

-- use the lemma to prove the goal
singeltonEq :{x, y : Char} ->  Prelude.Basics.(::) x [] = y :: [] -> x == y = True

matchCompletness (Chr x) ([y]) prf = rewrite singeltonEq prf in Refl
```

for `Alt` case we can split the proof into two cases (since disjunction has two constructors) and prove each case separately. in both case we use the recursive call to prove the goal, except for the Right case we need to use the `orTrueTrue` lemma to prove the goal.

```
matchCompletness (Alt x y) s (Left z) = let rec = matchCompletness x s z in
  rewrite rec in Refl
matchCompletness (Alt x y) s (Right z) = let rec = matchCompletness y s z in
  rewrite rec in rewrite orTrueTrue (match x s) in Refl
```

Further cases require more complex proofs which involves splitting the string into all possible prefixes and suffixes and recursively matching the two regular expressions with the prefixes and suffixes. Getting to the `Concat` case, we can furhter case split on `prf` to get the prefixes and suffixes and more proofs in the depenpendent pair and independent pair(conjunction) to prove the goal (`**` denotes the dependent pair and `(,)` denotes the independent pair). adding recursive calls in to let bindings to give us more information to prove the goal. by looking at the context and goal we can see we need to using some lemmas to show that z and w are in the splits of `s` and having proof that their are matching the regular expressions can prove the goal.

```
matchCompletness (Concat x y) s (((z, w) ** (v, (t, u)))) =
let rec1 = matchCompletness x z t
    rec2 = matchCompletness y w u
    temp = trans (cong (&& (match y w)) rec1 ) rec2
    in ?goal

------------------------------

 z : List Char
 w : List Char
 s : List Char
 v : s = z ++ w
 x : Regex Char
 t : lang x z
 y : Regex Char
 u : lang y w
 rec1 : match x z = True
 rec2 : match y w = True
 temp : match x z && Delay (match y w) = True
------------------------------
goal : matchConcatList x y (splits s) False = True
```

the lemmas we used in the proof are as follows:
(you can find the implementation of the lemmas in the Github repository [9])

```
splitElem : {s : List Char} -> (z, w : List Char) ->
  (p : s = z ++ w) -> Elem (z, w) (splits s)

splitsMatch: {s : List (List Char , List Char)} -> Elem (s1, s2) s ->
  match r1 s1 && match r2 s2 = True -> matchConcatList r1 r2 s False = True
```

By rewriting the goal using lemmas we can prove the goal.

```
matchCompletness (Concat x y) s (((z, w) ** (v, (t, u)))) =
let rec1 = matchCompletness x z t
    rec2 = matchCompletness y w u
    temp = trans (cong (&& (match y w)) rec1 ) rec2
    lem = sym (splitsMatch (splitElem z w v) temp) in rewrite lem
```

```
      in Refl
```

The `Star` case, we proof it as follows:

```
  matchCompletness (Star r) (y :: ys) (((x::xs) ** (z, w::ws))) =
  let rec = matchCompletness (Star r) (foldr (++) [] xs) (xs ** (Refl, ws) )
      rec1 = matchCompletness r x w
      There (splitPrf) = splitElem x (foldr (++) [] xs) z
      temp = trans (cong (&& (match (Star r) (foldr (++) [] xs))) rec1) rec
      spp = splitsMatch splitPrf temp
      in spp
matchCompletness (Star r) (y :: ys) (([] ** (z, ws))) = absurd z
matchCompletness (Star r) (y :: ys) (((x :: xs) ** (z, []))) impossible
matchCompletness (Star r) (y :: ys) ((xs ** (z, ws))) impossible
```

We add the extra cases that are impossible since idris2 wanted us to prove all the cases but as you can see the case has overlapping patterns with case that we already proved. This something I could not find a way to avoid in Idris2. My hypothesis that Idris2 can't figure out the cases that `xs` and `ws` has different sizes are not possible, so it asked me to prove them all.

We didn't further prove the soundness of the match function, but I assume it would be similar to the completeness proof but in the opposite direction.

6 Conclusion

This exploration of Idris2 demonstrates its unique capabilities in bridging formal verification and practical programming through Quantitative Type Theory (QTT). By introducing multiplicities (0, 1, ω), Idris2 enables precise control over resource usage patterns—as exemplified in our file system protocol where linear types enforced safe state transitions. The language's compile-time guarantees eliminate entire categories of runtime errors, particularly in resource-sensitive operations and stateful systems.

Regarding the proof verfication, Idris2 lacks some features that could make the proof process more efficient. For example, the overlapping patterns in the cases can make the proof to be more exhaustive than it should be. This can be challenging when dealing with complex data structures or recursive functions. Additionally, we experienced some cases where Idris2 couldn't further simplfy the goal and context when we have case clauses and we had to manually add the lemmas to prove the goal. This can make the proof process more cumbersome and less intuitive.

Idris2's REPL also could be improved by adding more interactive features, such as auto-completion, history navigation, and better error messages, to enhance the development experience. The editor integration is not as seamless as some other languages, and it make the development process more challenging.

Despite these challenges, Idris2's unique approach to dependent types and formal verification makes it a powerful tool for developing reliable and correct software. Its expressive type system, interactive development environment, and support for theorem proving offer a promising path towards safer and more robust software systems.

References

[1] Robert Atkey (2018): *Syntax and semantics of quantitative type theory.* In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 56–65.

[2] Edwin Brady (2013): *Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of functional programming* 23(5), pp. 552–593.

[3] Edwin Brady (2021): *Idris 2: Quantitative type theory in practice.* arXiv preprint arXiv:2104.00480.

[4] Edwin Brady (2023): *Idris 2: Quantitative Types in Action.* YouTube. Available at `https://www.youtube.com/watch?v=0uA-tKR6Ah4`.

[5] Conal Elliott (2021): *Symbolic and automatic differentiation of languages. Proc. ACM Program. Lang.* 5(ICFP), doi:10.1145/3473583. Available at `https://doi.org/10.1145/3473583`.

[6] Roger Hindley (1969): *The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society* 146, pp. 29–60.

[7] Giuseppe Lomurno: *Idris2-nvim.* Available at `https://github.com/idris-community/idris2-nvim`. GitHub repository.

[8] Per Martin-Löf & Giovanni Sambin (1984): *Intuitionistic type theory.* 9, Bibliopolis Naples.

[9] Zahra Khodabakhshian Mehrdad Shahidi: *code base for the seminar.* Available at `https://github.com/cyberkatze/idris-seminar`. GitHub repository.

[10] Robin Milner (1978): *A theory of type polymorphism in programming. Journal of computer and system sciences* 17(3), pp. 348–375.