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

Idris2 is a functional programming language that utilizes Quantitative Type Theory (QTT) to provide an advanced dependent type system. This system enhances both practical software development and formal proof verification by allowing developers to express complex specifications and ensure program correctness at compile time. This report explores the capabilities of Idris2 through a verification example, highlighting its unique features and advantages, and concludes with key insights into its potential applications and challenges.

1 Introduction

As software systems become increasingly complex and integral to our daily lives, the demand for programming languages that ensure reliability and correctness has grown significantly. Idris2 represents a major advancement in this area by offering a unique approach to program correctness through its type system, which is based on Quantitative Type Theory (QTT).

Idris2 is a pure functional programming language that treats types as first-class citizens, allowing them to be manipulated and computed like any other value. This capability fundamentally shifts the way we approach programming, enabling developers to work collaboratively with the compiler. The compiler acts as an assistant during development, helping to prove properties about the code and suggesting potential solutions for incomplete programs.

This type-driven development approach ensures program correctness before execution, making Idris2 particularly valuable for critical systems, mathematical proofs, and complex state-dependent applications such as concurrent systems [4]. For example, Idris2 can enforce safety in division operations, as demonstrated below:

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

example : Nat
example = safeDiv 12 5

example2 : Nat
example2 = safeDiv 12 0
```

In this example, the ‘safeDiv‘ function ensures that division by zero is prevented at compile time. The ‘example‘ variable compiles successfully because it divides 12 by 5, a valid operation. However, ‘example2‘ will result in a compile-time error because it attempts to divide by zero,

which violates the constraint enforced by the ‘GT d Z‘ condition. This demonstrates Idris2’s ability to catch potential runtime errors during compilation.

In the subsequent sections, this report explores several key aspects of Idris2. 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. Following this, we highlight Idris2’s interactive development environment, which supports type-driven development and interactive editing features. We also demonstrate a verification example using Idris2 to show its practical application in formal verification. Finally, we share insights on what Idris2 offers and the challenges we faced during the verification process.

2 Background

Programming languages have traditionally separated types and values, which means that certain errors, particularly those related to invalid values or incorrect data structures, are only caught at runtime. This limitation has posed 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 address 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 have 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, constructive logic 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 'Refl' 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, 'Refl' cannot prove 'plusReducesR' directly.

In Idris, an equality type that can be proved using 'Refl' alone is known as definitional equality. For cases like 'plusReducesR', where 'Refl' 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. This 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
```

The assertion '(xs = ys)' indicates that the two vectors are equal in both value and type. Therefore, 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, QTT 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) that denotes how many times a variable can be used within 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, dictating 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. Conversely, 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].

4.3 Linearity Example

In Idris 2, linearity (or multiplicity of 1) 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.

To illustrate this, we can consider a File System Protocol that allows us to open, close, and delete files. The protocol enforces linearity to ensure that files are managed correctly. The key rules are:

- 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
```

Next, we define 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
```

To ensure that files are created and used linearly, the ‘newFile‘ function 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."
```

An example protocol 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.

4.3.1 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.

For example, consider the following definition of a vector type:

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

In this definition, the ‘length‘ function retrieves the length of a vector, which is available at compile time but erased at runtime:

```
length : {0 n : Nat} -> Vec n a -> Nat
length {n} xs = n
```

The following example demonstrates a valid use of the ‘Vec‘ type, where the length information is available at compile time:

```
example : Vec 3 Integer
example = 1 :: 2 :: 3 :: Nil
```

However, if we try to define a function that sums the lengths of two vectors without making their lengths available at runtime, we encounter an error:

```
badSum : Vec m a -> Vec n a -> Nat
badSum xs ys = length xs + length ys
```

This results in an error because ‘m‘ and ‘n‘ are erased (multiplicity 0) and not accessible in this context.

To resolve this, we can define a function that makes ‘m‘ and ‘n‘ available at runtime:

```
goodSum : {m, n : _} -> Vec m a -> Vec n a -> Nat
goodSum xs ys = length xs + length ys
```

This example illustrates 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.

4.4 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 are available on GitHub [7].

4.4.1 Core Workflow

The core workflow in Idris2 begins with hole-driven development, where programmers start with type signatures and placeholders (holes) in their code. For example, consider the following function signature for ‘vzipWith‘, which combines two vectors:

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

In this case, ‘?vzipWith_rhs‘ represents a hole that we need to fill in with the implementation of the function.

Next, we can perform case splitting on the first argument, ‘xs‘, to generate clauses for different scenarios. This allows us to handle the base case (when ‘xs‘ is empty) and the recursive case (when ‘xs‘ has elements):

```
vzipWith f [] ys = ?vzipWith_rhs_1
vzipWith f (x :: xs) ys = ?vzipWith_rhs_2
```

At this point, the context and goal are displayed to the programmer. The context shows the types of the variables involved, while the goal indicates what needs to be proven or implemented. For example, the context might look like this:

```
0 c : Type
0 b : Type
0 a : Type
  ys : Vect 0 b
  f : a -> b -> c
0 n : Nat
```

```
------------------------------
vzipWith_rhs_1 : Vect 0 c
```

Here, the context provides information about the types of ‘f‘, ‘xs‘, and ‘ys‘, as well as the expected type of the result. The goal is to fill in the hole or use proof search to find the solutions if possible. This interactive process allows for rapid development and verification of code, ensuring that the implementation adheres to the specified types and logic.

5 Verification Example

In this section, we aim to prove the correctness of a simple regular expression match function using Idris2. The function takes a regular expression and a string as input, returning a boolean indicating whether the string matches the regular expression.

We define the regular expression type using algebraic data types as follows:

```
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 type signature for the match function is as follows:

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

Our proof specification formula aims to show:

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

To achieve this, we need to capture the membership of a string in the language defined by a regular expression.

5.1 Language Representation

From automata theory, we know that regular expressions are equivalent to formal languages, which are typically represented in set theory as a set of strings. In type theory, we can represent languages as predicates on strings, allowing us to demonstrate string membership in a language [5].

We define languages in Idris as follows:

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

```
empty : Lang a
empty _ = Void

univ : Lang a
univ _ = Unit

eps : Lang a
eps w = w = []

tok : a -> Lang a
tok c w = w = [c]

(:.:) : Type -> Lang a -> Lang a
(:.:) s l w = Pair s (l w)

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

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

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

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)))

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

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)
```

Our correctness proof specification involves proving both 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

To implement the match function, we can use a simple recursive approach that matches the regular expression against the input string. This implementation is straightforward and facilitates proving correctness more easily than other potential implementations. The implementation is as follows:

```
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 uses pattern matching on the regular expression and the input string, recursively matching based on the structure of the regular expression. The `matchConcatList` function is employed to match the concatenation of two regular expressions with the string by splitting the string into all possible prefixes and suffixes, then recursively matching the two regular expressions with these prefixes and suffixes.

The implementations of `matchConcatList` and `splits` are as follows:

```
matchConcatList : (r1 : Regex Char) -> (r2 : Regex Char) ->
(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)
```

Next, we proceed with the proof, starting with 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 with the first case, since we define the Empty language as a predicate that yields the bottom type (Void), we can use the `absurd` function (ex-falso-quadlibet) to prove this case. For the Epsilon case, Idris2 provides enough information to prove it simply by rewriting the `prf` in our goal:

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

matchCompleteness Epsilon s prf = rewrite prf in Refl
```

For the `Chr` case, we need to further case split on the string `s` to align with the cases defined in the `match` function. For the cases where the string is `[]` or `(x1::x2::xs)`, we can use the `impossible` syntax to indicate that these cases are not possible. The singleton case `[y]` is the only case we need to prove, and we require a lemma stating that the equality of a singleton list implies the equality of its elements:

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

We can use the lemma to prove the goal:

```
singeltonEq : {x, y : Char} -> Prelude.Basics.(::) x [] = y :: [] -> x == y = True

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

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

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

```
    rewrite rec in rewrite orTrueTrue (match x s) in Refl
```

Further cases require more complex proofs, which involve splitting the string into all possible prefixes and suffixes and recursively matching the two regular expressions with these prefixes and suffixes.

For the `Concat` case, we can further case split on `prf` to obtain the prefixes and suffixes, leading to more proofs involving dependent pairs and independent pairs (conjunction) to prove the goal. The dependent pair is denoted by `**` and the independent pair by `(,)`. We add recursive calls into let bindings to provide more information for proving the goal. By examining the context and goal, we can see that we need to use some lemmas to show that z and w are in the splits of `s` and that having proof of their matching with the regular expressions can help us prove the goal:

```
matchCompleteness (Concat x y) s (((z, w) ** (v, (t, u)))) =
    let rec1 = matchCompleteness x z t
        rec2 = matchCompleteness 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 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 these lemmas, we can prove the goal:

```
matchCompleteness (Concat x y) s (((z, w) ** (v, (t, u)))) =
    let rec1 = matchCompleteness x z t
```

```
        rec2 = matchCompleteness y w u
        temp = trans (cong (&& (match y w)) rec1) rec2
        lem = sym (splitsMatch (splitElem z w v) temp) in rewrite lem
    in Refl
```

For the `Star` case, we prove it as follows:

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

matchCompleteness (Star r) (y :: ys) (([] ** (z, ws))) = absurd z
matchCompleteness (Star r) (y :: ys) (((x :: xs) ** (z, ws))) = impossible
matchCompleteness (Star r) (y :: ys) ((xs ** (z, ws))) = impossible
```

We add the extra cases that are impossible since Idris2 requires us to prove all cases. However, as noted, these cases overlap with those we have already proven. This is a limitation in Idris2, as it seems unable to infer that cases where `xs` and `ws` have different sizes are not possible, thus requiring us to prove them all.

We did not further prove the soundness of the match function, but we assume it would be similar to the completeness proof, albeit 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 enforce safe state transitions. This feature not only enhances the reliability of software but also allows developers to reason about resource management at a higher level of abstraction.

The language's compile-time guarantees effectively eliminate entire categories of runtime errors, particularly in resource-sensitive operations and stateful systems. This is particularly beneficial in critical applications where safety and correctness are paramount, such as in financial systems, medical devices, and other safety-critical software.

However, despite its strengths, Idris2 does have limitations that can impact the proof verification process. For instance, the presence of overlapping patterns in case analysis can lead to proofs that are more exhaustive than necessary, complicating the verification of complex data structures or recursive functions. Additionally, there are instances where Idris2 struggles to simplify the context and goals during proof construction, necessitating the manual addition of lemmas to facilitate the proof process. This can make the verification experience less intuitive and more cumbersome for developers.

Furthermore, while Idris2's REPL provides a solid foundation for interactive development,

there is room for improvement. Enhancements such as auto-completion, history navigation, and more informative error messages could significantly enrich the development experience, making it more user-friendly and efficient. The integration with editors, while functional, could also be made more seamless to align with the expectations set by other modern programming languages.

In conclusion, Idris2's innovative approach to dependent types and formal verification positions it as a powerful tool for developing reliable and correct software. Its expressive type system, combined with an interactive development environment and support for theorem proving, offers a promising path toward safer and more robust software systems. As the language continues to evolve, addressing its current limitations will be crucial in broadening its adoption and enhancing its usability for a wider range of programming tasks.

Acknowledgements

We would like to express our gratitude to the developers and contributors of Idris2 for their innovative work on this functional programming language. Their efforts have made it possible to explore advanced concepts in type theory and formal verification.

We also extend our heartfelt thanks to our supervisor, Cass Alexandru, who provided invaluable guidance and support throughout this project. Their insights and encouragement were instrumental in shaping our work.

Additionally, we acknowledge the use of AI assistance in improving our report and for autocompleting some parts of our code. This support has significantly enhanced the clarity and quality of our work.

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.