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

RPTU 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 example problem verification using Idris2 to showcase its capabilities, examines its QTT-based approach, compares its features with Agda, and concludes by summarizing key insights and suggesting directions for future exploration.

add problem name here

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
```

The following sections provide an accessible overview of Idris2's key aspects. We begin with a brief introduction to Quantitative Type Theory (QTT) and compare Idris2 with both Haskell

and Agda, highlighting how these languages approach type systems differently. While Haskell represents traditional functional programming and Agda focuses heavily on theorem proving, Idris2 aims to bridge the gap between practical programming and formal verification. We then explore some distinctive features of Idris2 through simple examples, such as its interactive development environment and dependent types. To demonstrate these concepts in action, we present a straightforward verification example that shows how Idris2's type system can prevent common programming errors. The report concludes with a discussion of current limitations and practical considerations when using Idris2 in real-world scenarios.

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 Idris2 enforce safe resource usage by modeling state transitions, preventing runtime errors, and ensuring predictable, reliable programs.

4.3 Interface

we use interfaces (similar to type classes in Haskell) to define functions or operations that can work seamlessly across different types. This capability, known as overloading, is achieved by defining a common interface (a set of operations) and then providing specific implementations for each type that becomes an instance of the interface.

4.3.1 Implementing Show for Nat :

the Show interface is used to convert values into String.The key rule is uniqueness of the implementation , There should be only one implementation of am interface for specific type.

```
    Show Nat where
    show Z     = "Zero"
    show (S Z) = "One"
    show (S (S k)) = "Two"
    show (S (S (S k))) = "S(" ++ show (S (S k)) ++ ")"
```

Additionally, Implementations can have constraints. For instance, to implement Show for a vector (Vect n a), there must already be a Show implementation for the element type a.

```
    Show a => Show (Vect n a) where
```

This is a constraint. It means the implementation of Show for Vect n a (a vector of n elements of type a) is only valid if the element type a itself has an instance of Show.

```
    Show a => Show (Vect n a) where
    show xs = "[" ++ show' xs ++ "]" where
        show' : forall n . Vect n a -> String
        show' Nil        = ""
        show' (x :: Nil) = show x
        show' (x :: xs)  = show x ++ ", " ++ show' xs
```

4.3.2 Type classes in Idris vs Haskell

Type classes are a central feature of both Haskell and Idris 2. While they share many similarities, Idris 2's more expressive dependent type system enables additional flexibility and power.

In Idris 2, type classes leverage dependent types, allowing types to depend on values. This extends polymorphism beyond types (as in Haskell) to include relationships between types and values, enabling more expressive and precise constraints. Haskell lacks this capability due to its non-dependent type system.

4.4 Bidirectional Type Checking

4.4.1 Type Checking in Idris2 vs Haskell

Haskell uses a monotonic type checking system, primarily based on ** Hindley-Milner type inference** with extensions for generalized algebraic data types (GADTs) and type families.

Types are typically inferred using a unification algorithm, which attempts to match the types of expressions with the expected types. The Haskell compiler (GHC) uses a form of principal type inference for non-dependent types, meaning it attempts to deduce the most general type that fits the given expression, but it doesn't differentiate between checking and synthesizing as explicitly.

However, Idris 2 uses bidirectional type checking, which separates the roles of type synthesis (deducing types) and type checking (verifying types). This is crucial for dependent types and provides more flexibility and efficiency in type inference.

4.5 Interactive Development Environment

4.6 Dependent Types in Practice

5 Verification Example

5.1 Problem Description

5.2 Implementation and Proof

6 Practical Considerations

6.1 Current Limitations

6.2 Future Directions

7 Conclusion

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

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

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