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

Idris, first introduced by Edwin Brady in 2009, emerged as a programming language designed to explore dependent types in a practical programming context. The language drew inspiration from both Haskell's practical functional programming approach and Agda's powerful type system. While the original Idris (now known as Idris1) successfully demonstrated the potential of dependent types in practical programming, it also revealed certain limitations in its implementation and theoretical foundations [2].

These insights led to the development of Idris2, a complete redesign of the language implemented in Idris1 itself. The new version introduced several significant improvements, including:

- Better type checking performance
- Improved erasure of compile-time-only arguments
- Linear types through Quantitative Type Theory
- A more robust implementation based on experience with Idris1

A key example demonstrating Idris2's practical application of dependent types is the classic vector length safety:

```
data Vect : Nat -> Type -> Type where
    Nil  : Vect Z a
    (::) : a -> Vect k a -> Vect (S k) a

-- Safe head function that can only be called on non-empty vectors
head : Vect (S n) a -> a
head (x :: _) = x

-- This compiles
example1 : Integer
example1 = head [1,2,3]

-- This fails to compile as the vector might be empty
example2 : Maybe Integer
example2 = head []
```

2.1 Quantitative Type Theory Foundations

Quantitative Type Theory (QTT) extends traditional dependent type theory by adding explicit tracking of how variables are used in programs [1]. In Idris2, QTT serves two main purposes:

- Providing clear type-level guarantees about which values are required at runtime
- Enabling precise tracking of resource usage through linear types

2.1.1 Core Concepts

QTT in Idris2 uses three fundamental quantities that specify how variables can be used:

- 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)

For example, here's how quantities work in practice:

```
-- The 0 quantity means 'n' is erased at runtime
length : {0 n : Nat} -> Vect n a -> Nat
length [] = 0
length (x :: xs) = 1 + length xs

-- The 1 quantity ensures the file handle is used exactly once
writeAndClose : (1 handle : File) -> IO ()
writeAndClose handle = do
    writeFile handle "Hello"
    closeFile handle  -- Must close the file exactly once
```

2.1.2 Benefits of QTT

The integration of QTT in Idris2 provides several key advantages:

- **Erasure Control**: Clear type-level specifications of which values are needed at runtime
- **Resource Safety**: Linear types ensure resources are used exactly once
- **Performance**: Better control over runtime behavior through erasure
- **Protocol Safety**: Ability to enforce correct usage of protocols and state machines

3 Comparing Idris2 with Haskell and Agda

While Idris2, Haskell, and Agda are all functional programming languages with strong type systems, they each serve different purposes and offer distinct approaches to type theory and practical programming. This section examines their key differences and similarities across various aspects.

3.1 Type System Characteristics

- **Haskell**: Features a Hindley-Milner type system[1] with type classes, offering strong static typing without dependent types. Type-level programming is possible through extensions but is not a core feature.
- **Agda**: Implements a dependent type system based on Martin-Löf Type Theory[2], primarily focused on theorem proving and formal verification. Its type system is more expressive than Haskell's but requires more manual proof work.
- **Idris2**: Combines practical programming with dependent types through QTT, offering a middle ground between Haskell's practicality and Agda's theorem-proving capabilities. Its type system supports both runtime and compile-time type checking with explicit erasure control.

3.2 Practical Programming Aspects

Consider the following example implementing a safe list indexing function in all three languages:

```haskell
-- Haskell: Runtime safety through Maybe type
(!?) :: [a] -> Int -> Maybe a
xs !? n | n < 0     = Nothing
        | otherwise = case drop n xs of
                        []     -> Nothing
                        (x:_)  -> Just x
```

```agda
-- Agda: Compile-time safety through dependent types
_!!_ :  {A : Set} {n}  Vec A n  Fin n  A
(x  xs) !! zero  = x
(x  xs) !! suc i = xs !! i
```

```idris
-- Idris2: Practical dependent types with erasure
index : (i : Fin n) -> Vect n a -> a
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

Key differences in practical usage include:

- **Development Experience**:
  - Haskell offers the most mature ecosystem and tooling
  - Agda focuses on interactive theorem proving
  - Idris2 provides interactive type-driven development with practical programming features
- **Performance Considerations**:

[1] Named after Roger Hindley and Robin Milner for their foundational work on type inference [5, 7].

[2] Martin-Löf's type theory [6] introduced dependent types and laid the foundation for modern proof assistants like Agda.

  - Haskell generally offers the best runtime performance
  - Agda's focus is on correctness rather than performance
  - Idris2's QTT enables better performance through erasure of compile-time-only values

3.3 Theorem Proving Capabilities

The languages differ significantly in their approach to formal verification:

- **Haskell**:
  - Limited built-in theorem proving capabilities
  - Relies on external tools like LiquidHaskell for formal verification
  - Properties often verified through testing rather than proofs
- **Agda**:
  - First-class support for theorem proving
  - Extensive library for mathematical proofs
  - Requires explicit proof terms
- **Idris2**:
  - Balanced approach to theorem proving
  - Automated proof search capabilities
  - Integration of proofs with practical programming

3.4 Syntax and Learning Curve

The syntax and learning curve vary significantly:

- **Haskell**: Most accessible syntax, familiar to many functional programmers
- **Agda**: Steeper learning curve, requires understanding of type theory
- **Idris2**: Haskell-like syntax with additional complexity from dependent types

An example showing similar functions across the three languages:

```haskell
-- Haskell: Simple pattern matching
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```agda
-- Agda: Explicit type universe and termination checking
reverse :  {A : Set}  List A  List A
reverse []       = []
reverse (x  xs) = reverse xs ++ (x  [])
```

```idris
-- Idris2: Dependent types with erasure
reverse : {0 a : Type} -> List a -> List a
reverse [] = []
reverse (x :: xs) = reverse xs ++ [x]
```

3.5 Use Case Recommendations

Based on these comparisons, each language is best suited for different scenarios:

- **Choose Haskell** for:
  - Production-ready software development
  - Large-scale applications
  - When a mature ecosystem is crucial
- **Choose Agda** for:
  - Mathematical proofs and formalization
  - Academic research in type theory
  - When maximum proof power is needed
- **Choose Idris2** for:
  - Combining practical programming with formal verification
  - Projects requiring precise resource management
  - When dependent types need to coexist with practical features

4 Key Features of Idris2

4.1 Interactive Development Environment

4.2 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.