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 dependently-typed functional programming language designed to facilitate formal verification and program correctness through its expressive type system. By leveraging dependent types, Idris2 enables developers to encode precise specifications directly within the type system, catching potential errors at compile time. This report explores Idris2's capabilities through a verification example, examining its approach to theorem proving and interactive development. We discuss its strengths in formal verification, as well as challenges related to proof complexity and usability.

1 Introduction

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. By treating types as first-class entities, Idris2 enables developers to encode correctness properties in a way that can be checked at compile time, with the compiler acting as an assistant during development, reducing certain classes of runtime errors.

This type-driven development approach ensures program correctness before execution, making Idris2 particularly useful for critical systems, mathematical proofs, and applications with complex state dependencies [3]. 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 is designed to prevent division by zero by enforcing a compile-time constraint on its second argument. The constraint `GT d Z` ensures that `d` will be strictly greater than zero. The `{auto ok : ...}` syntax indicates an implicit argument with automatic proof search, a feature documented in the Idris2 reference manual [13]. While this is a simple example, what makes it powerful is not just the evaluation of constant expressions (which many compilers can do), but rather Idris2's ability to express logical constraints as part

of the type system. 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 at the invocation site. This capability extends beyond simple constants to arbitrary expressions where the constraint can be verified statically.

In the subsequent sections, this report explores several key aspects of Idris2. We begin with a brief overview of the language's background (Section 2) and its evolution from Idris 1 to Idris2. Next, we delve into the theoretical underpinnings of dependent types, focusing on propositions and equality (Section 3.2) in Idris2. We then discuss the features of Idris2 (Section 4), including Quantitative Type Theory (QTT) (Section 4.1), which formalizes resource usage through concepts like multiplicities (Section 4.2). Following this, we highlight Idris2's interactive development environment (Section 4.4) which facilitates type-driven development through features such as hole-driven programming, case splitting, and proof search. To illustrate its practical application in formal verification, we present a verification example (Section 5). Finally, we discuss the challenges encountered during the verification process and the broader implications of Idris2's approach to correctness (Section 5.3).

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 [10]. 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 [2]. 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 list 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 Idris2 [3].

Idris2 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. Idris2 improves type checking performance compared to its predecessor by optimizing evaluation strategies. Furthermore, the language enhances compile-time argument erasure, reducing the overhead of generated executable code. Additionally, Idris2 incorporates linear types as part of Quantitative Type Theory (QTT), providing a mechanism for better resource management and more precise control over side effects in functions [3].

3 Theorem Proving

3.1 Propositions

Before discussing theorem proving, we must first understand some fundamental concepts. A proposition is a statement that can be proven. Before proving it, we cannot formally determine whether it is true. A successful proof serves as evidence that establishes the proposition's validity. In functional programming, propositions correspond to types, and proofs correspond to values of those types [8].

Constructive logic, also known as intuitionistic logic, differs from classical logic by rejecting the Law of the Excluded Middle (LEM). In classical logic, LEM states that for any proposition P, either P is true or its negation is true ($P \vee \neg P$). Constructive logic requires explicit evidence to establish the truth of a proposition—if no proof exists, the truth value remains unknown. It's worth noting that LEM does hold for decidable propositions in intuitionistic logic, which are propositions where we can algorithmically determine whether they are true or false [5].

3.2 Equality

3.2.1 Definitional and Propositional Equality

The equality type (=) in Idris2 follows the Martin-Löf Identity Type [10].

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

This enforces definitional equality, meaning that two terms must reduce to the same normal form to be considered equal.

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

This proposition asserts that adding zero to any number `n` results in `n`, which holds definitionally true. To understand why this works, we need to look at how `plus` is defined in Idris2:

```
plus : Nat -> Nat -> Nat
plus Z y = y
plus (S k) y = S (plus k y)
```

When we evaluate `plus Z n`, it immediately reduces to `n` by the first clause of the definition, making `Refl` a valid proof.

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

The above definition does not work because `plus` is defined recursively on its first argument, and `plus n Z` does not immediately normalize to `n` in all cases. Since definitional equality in

Idris requires both sides to reduce to the same normal form automatically, `Refl` is not a proof of `plusReducesR`.

To prove this equality, we need propositional equality, which allows reasoning about equalities that do not hold by mere computation. We can prove this using induction on the natural number `n`:

```
plusReducesR : (n : Nat) -> plus n Z = n
plusReducesR Z = Refl
plusReducesR (S k) = rewrite plusReducesR k in Refl
```

In this proof, we first handle the base case where `n = Z`, which is trivial since `plus Z Z = Z` by definition. For the inductive case `n = S k`, we assume as our induction hypothesis that `plus k Z = k` holds. We then use the `rewrite` syntax to substitute this equality into our goal, transforming `plus (S k) Z = S k` into `S (plus k Z) = S k`, and finally into `S k = S k`, which is provable by `Refl`.

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 and Bob

Atkey, QTT addresses a fundamental challenge in type theory: managing the distinction between compile-time type information and runtime computational data.

In traditional dependent type theory, there is often no clear separation between values used for computation and values used only for type checking. This can lead to inefficiencies where information that is only needed during type checking is retained at runtime. QTT solves this problem by introducing a formal system to track variable usage, allowing the compiler to determine which values can be safely erased after type checking and which must be preserved for runtime computation.

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)

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 Idris2, 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 [3].

To illustrate this, we can consider a file system protocol that allows us to open, close, and delete files. The protocol uses 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 ()
```

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

If we deviate from the correct sequence—for example, by failing to delete the file—Idris rejects the program at compile time. Consider the following incorrect example:

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

This will result in a type error, as the linear type system detects that `f"` was not used exactly once.

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.

Consider this vector type definition:

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

In Idris2, dependent type parameters like `n` in `Vec n a` have a multiplicity of zero by default, meaning they're erased at runtime. If we want to access the length parameter in a function, we must explicitly include it in the type signature:

```
length1 : Vec n a -> Nat
length1 xs = ?goal
```

The context for the above function would be:

```
0 a : Type
0 n : Nat
xs : Vec n a
------------------------------
goal : Nat
```

This results in an error because `n` has multiplicity 0, making it inaccessible at runtime. By explicitly binding `n` in the type signature, we change its multiplicity:

```
length2 : {n : Nat} -> Vec n a -> Nat
length2 {n} xs = n
```

The context for this function would be:

```
0 a : Type
n : Nat
xs : Vec n a
------------------------------
goal : Nat
```

Now, `n` has unrestricted multiplicity, meaning it can be used at runtime.

We can also explicitly set a parameter's multiplicity to zero by placing `0` before the parameter name. For example, `{0 n : Nat}` would make `n` have multiplicity zero even when explicitly bound, ensuring it's erased at runtime while still being accessible during type checking.

This design ensures that type-level information doesn't incur runtime overhead unless explicitly needed, optimizing performance while maintaining compile-time guarantees.

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 primary editor for Idris2 development, utilizing the idris2-nvim plugin [9]. This plugin provides key bindings for common REPL interactions, including: 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.

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. At this stage of development, the context is as follows:

```
 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.

5 Verification Example

In this section, we aim to prove the completeness of a simple regular expression match function using Idris2. While a full correctness proof would include both soundness and completeness, we focus specifically on proving completeness, showing that if a string is in the language denoted by a regular expression, then our matching algorithm will correctly identify it. The function takes a regular expression and a string as input, returning a boolean indicating whether the string matches the regular expression (5.1).

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 expressions are parametrized by a type `a` which represents the type of elements in our alphabet. In our implementation, we instantiate `a` with `Char`, allowing `match` to use decidable equality, which `Char` provides.

There are two ways to define the semantics of a regular expression: denotational and operational. We formalize both approaches and then aim to prove they are equivalent (we only prove completeness here).

5.1 Operational Semantics

The operational semantics of regular expressions is defined by a `match` function that determines whether a string is accepted by a regular expression. Our implementation is as follows:

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

Further implementation for `matchConcatList` and `splits` can be found in our public repository [12].

5.2 Denotational Semantics

We follow the approach by Conal Elliott in "Symbolic and automatic differentiation of languages" of representing languages as predicates on strings [6].

We define a language as:

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

This means a language over alphabet type `a` is a predicate that, given a list of elements of type `a`, produces a type. When this type is inhabited, the string belongs to the language; when it's uninhabited (like `Void`), the string is not in the language.

The fundamental language operations are defined below, each precisely capturing a particular way of constructing languages:

Empty language ($\emptyset$): Contains no strings at all. This is represented using `Void`, which is uninhabited, ensuring no string belongs to this language.

```
empty : Lang a
empty _ = Void
```

Universal language (Σ^*): Contains all possible strings over the alphabet. Represented using `Unit`, which is always inhabited, so every string belongs to this language.

```
univ : Lang a
univ _ = Unit
```

Epsilon language (ϵ): Contains only the empty string. A string belongs to this language if and only if it equals the empty list.

```
eps : Lang a
eps w = w = []
```

Single-token language (c): Contains only the one-character string [c]. A string belongs to this language if and only if it's a singleton list containing the specified character.

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

Union ($P \cup Q$): Contains all strings that belong to either of two languages. A string belongs to the union if it belongs to either constituent language, represented by the sum type `Either`.

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

Intersection ($P \cap Q$): Contains strings that belong to both languages. A string belongs to the intersection if it belongs to both constituent languages, represented by the product type `Pair`.

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

Concatenation ($P * Q$): Contains strings that can be split into two parts where the first part belongs to the first language and the second part belongs to the second language. We use dependent pairs and existential quantification to express this operation.

```
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)))
```

Kleene star (P^*): Contains strings that can be decomposed into a list of substrings, each belonging to the input language. This includes the empty string (corresponding to an empty list of substrings).

```
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))
```

The semantics of regular expressions is then given by the `lang` function, which maps each regular expression constructor to its corresponding language operation:

```
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)
```

5.3 Proof of Completeness

Having defined both operational semantics (the `match` function in Section 5.1) and denotational semantics (the `lang` function), we can now prove the completeness property.

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

We begin 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 to prove this case. For the `Epsilon` case, Idris2 provides intermediate information to prove it by rewriting the equality proof (`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
```

Idris2 can reach propositional equality (`x = y`) from the equality of singleton lists but not the boolean equality (`==`). Using the `believe_me` function we can prove (`x == y = True`). As noted in the Idris2 documentation, `believe_me` is an unsafe cast that allows users to bypass the typechecker when they know something to be true even though it cannot be proven [14]. It should only be used when we have a proof that Idris2 cannot find.

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

matchCompleteness (Chr x) ([y]) prf = rewrite singletonEq 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
```

The `orTrueTrue` lemma, which is built into the Idris2 standard libraries, has the following type signature:

```
orTrueTrue : (x : Bool) -> x || Delay True = True
```

This lemma simply formalizes that for any boolean value `x`, the expression `x || True` always evaluates to `True`. The `Delay` keyword defers evaluation of its argument until needed.

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 prove intermediate subgoals. By examining the context and goal, we can see that we need to use some lemmas to show that `pre` and `suf` 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 (((pre, suf) ** (v, (t, u)))) =
    let rec1 = matchCompleteness x pre t
        rec2 = matchCompleteness y suf u
        temp = trans (cong (&& (match y suf)) rec1) rec2
    in ?goal

------------------------------
pre : List Char
suf : List Char
s : List Char
v : s = pre ++ suf
x : Regex Char
t : lang x pre
y : Regex Char
u : lang y suf
rec1 : match x pre = True
rec2 : match y suf = True
temp : match x pre && Delay (match y suf) = True
```

```
------------------------------
goal : matchConcatList x y (splits s) = True
```

The lemmas used in the proof are as follows:

```
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 = True
```

By rewriting the goal using these lemmas, we can prove the goal:

```
matchCompletness (Concat x y) s (((pre, suf) ** (v, (t, u)))) =
let rec1 = matchCompletness x pre t
    rec2 = matchCompletness y suf u
    temp = trans (cong (&& (match y suf)) rec1 ) rec2
    lem = sym (splitsMatch (splitElem pre suf 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.

6 Conclusion

This exploration of Idris2 demonstrates its capabilities in formal verification through dependent types. Our verification example with regular expressions illustrates how Idris2's type system enables us to encode and prove properties about programs. By expressing the semantics of regular expressions both denotationally and operationally, we were able to formally prove the completeness property of our matching algorithm.

Idris2's compile-time guarantees effectively help eliminate certain categories of runtime errors through type checking. The interactive development environment, with features like hole-driven

development and case splitting, provides valuable assistance when constructing proofs. These tools were particularly helpful in our verification example when dealing with complex pattern matching and equality proofs.

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.

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] Michael Dummett (2000): *Elements of Intuitionism*, 2 edition. *Oxford Logic Guides* 39, Oxford University Press.

[6] 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`.

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

[8] William A Howard (1980): *The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44, pp. 479–490.

[9] Giuseppe Lomurno: *Idris2-nvim: Idris2 language server for Neovim.* Available at `https://github.com/idris-community/idris2-nvim`.

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

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

[12] Mehrdad Shahidi & Zahra Khodabakhshian (2024): *Source code for implementations and proofs.* Available at `https://github.com/cyberkatze/idris-seminar`.

[13] The Idris Community (2023): *Custom Backend Cookbook - BelieveMe.* Available at `https://idris2.readthedocs.io/en/latest/tutorial/miscellany.html#auto-implicit-arguments`.

[14] The Idris Community (2023): *Custom Backend Cookbook - BelieveMe.* Available at `https://idris2.readthedocs.io/en/latest/backends/backend-cookbook.html#believeme`.

[15] Philip Wadler (2015): *Propositions as Types. Communications of the ACM* 58(12), pp. 75–84, doi:10.1145/2699407.