Dependent Types: Level Up Your Types

Marco Syfrig
University of Applied Sciences Rapperswil
Supervised by Prof. Peter Sommerlad
Seminar—FT2016

"It typechecks! Ship it!"
– Edwin Brady, designer of Idris [Bra]

Dependent types make types into first-class language constructs and ensure type safety depending on values and not only types. They help to get rid of all unit tests because dependent types require a proof that a function behaves as expected. This proof needs to be written by the developer himself and verifies that the function works correctly for all possible inputs. So a type, in the end, is a proof for the compiler and no unit tests are needed. Type checking may become undecidable since these types can depend on any value or expression.

This paper gives an overview what problem dependent types solve, kinding, dependent type theory and then a fairly big part about the differences to other systems. It then shows some current ongoing research and finally explains why dependent types may play a bigger role in the far future.

1 Introduction

The initial quote "It typechecks! Ship it!" from Edwin Brady [Bra]—designer of the dependently typed language Idris—may be exaggerating, but it highlights one of the main reasons of type checking. A developer wants to rule out as many errors as possible by automatically type checking his program. This is where dependent types come into play. They help a developer find even more errors during compile-time rather than only at run-time with run-time checks. Consider the Java method `first` in Listing 1 that returns the first element of a vector.

```
1 public static Integer
     first(Vector<Integer> vec) {
2 if(!vec.isEmpty()) {
3     return vec.get(0);
4   }
5   return null;
6 }
```

Listing 1. A check for the existence of an element is needed in a non-dependently typed language like Java

The check for `null` is omitted for readability. One crucial thing for the correctness is the check on *line 2* that the vector `vec` is not empty. Otherwise, a call like `first(new Vector<Integer>());` results in a run-time error and in the case of Java throws an `ArrayIndexOutOfBoundsException`. Notice also that the non-emptiness check has to be done at run-time which takes a little bit of time every time the method is called. The last burden is on the caller that has to check if the result is `null` and if so handle it. Another implementation could also throw and catch the above-mentioned exception or the caller could check for non-emptiness before calling. However, there is always at least one run-time check required and it has to be specified and clear whether the caller or callee does this check. Here is where dependent types step in and resolve this issue.

With a dependent type of vectors we can encode the requirement for non-emptiness into the type.

$$first : \Pi n : Nat.Vector(n+1) \rightarrow Int \qquad (1)$$

The preceding lambda expression (1) is taken from [Pie04] and explained in Section 3 in greater detail. It represents a function signature for a function that also returns the first element but with a dependent type. Passing an empty vector to *first* results in a compile-time error. Listing 2 shows an implementation in the dependently typed programming language Idris [idr]. The realization is exactly the same; just the type is named `Vect` in Idris.

```
1  import Data.Vect
2
3  -- Natural numbers definition
4  -- for better understanding
5  data Nat = Z | Suc Nat --zero | successor
6
7  first : Vect (Suc len) elem -> elem
8  first (x::xs) = x
9
10 firstNat : Nat
11 firstNat = first Vect 0 Nat
```

Listing 2. Typesafe function `first` to get the first element of a non-empty `Vect` in Idris

A `Vect` is created with the length and type it holds and `Vect 0 Nat` yields actually a different type than `Vect 1 Nat` since the length is different. `first` expects a vector that has been created with the `Suc Nat` data constructor which means it is bigger than `0`. The compiler checks the correctness of the code by looking at the function signature and the specified types there. If they do not match, for example for the application `first Vect 0 Nat` on *line 11*, the compiler throws an error about a type mismatch: `(input):1:7:When checking an application of function first: Type mismatch between 0 and Suc len`.

Another example of what can be done with dependent types is the function `isSingleton` in Listing 3. It computes a type that depends on the passed value. This can then be used in another function `sum` that calculates the sum of a single value or a list, depending on the passed type.

```
1  isSingleton : Bool -> Type
2  isSingleton True = Nat
3  isSingleton False = List Nat
4
5  sum : (single : Bool) -> isSingleton
       single -> Nat
6  sum True x = x
7  sum False [] = 0
8  sum False (x :: xs) = x + sum False xs
```

Listing 3. Function `isSingleton` in Idris that computes a type and returns it. The returned type is then used in `sum`.

The compiler checks for every possible input—which is really easy for just `True` and `False`—if a type is returned.

Type systems should help build safer programs, have more expressive APIs, help a developer with hints and also the other direction, that type information should help the compiler to be more efficient. The following parts of the paper introduce a more formal definition of dependent types, explain the necessary knowledge to understand the expression in λ-Expression (1) and then some parts of Idris to better understand code examples as the ones shown above. Further, dependent types are compared to the more automated but less expressive refinement types and to C++ template programming which can achieve the same but has different goals. It will be explained that it is expected that dependent types will play a bigger role in the future as there is a lot of research and development currently, especially for Haskell. However, they will not play a more important role in everyday programming as long as such a type system cannot be merged with an existing one that does not require a developer to write his own proofs for every small property. The last section sums this up again.

2 Kinds

Types become first-class citizens in a dependent type system. However, we can start writing meaningless type expressions with the introduc-

tion of types as first class citizens and the abstraction and application rules for them. This section explains how this can be prevented with something called kinds. Just as we use the arrow type $\rightarrow$ for functions to denote the arity, we use kinds to denote the arity of a type. $Nat \rightarrow Nat$ has a different arity than $Nat \rightarrow Nat \rightarrow Nat$. We need that also for types. Readers not familiar with type systems and their formal notation are advised to gain a quick overview by first reading [Syf15, Chapter 2 Definitions].

The rules are not relevant for the understanding, but they can be found in [Pie02, Figure 29-1]. *(Bool Nat)* is a legal expression following these rules but meaningless. Applying *Nat* to *Bool* should not be possible. We achieve this by defining a system of kinds that restrict such expressions which will now be explained.

This system should classify type expressions by their arity so a proper type cannot be applied to a proper type anymore. The following listing is taken from [Pie02, Chapter 29] and shortened for simplification.

- $*$ Kind of proper types that take no arguments for instantiations like *Bool* and $Bool \rightarrow Bool$
- $* \Rightarrow *$ Kind of type operators. They are one arity higher. This means unary type constructors for constructing a list of a proper type for example.
- $* \Rightarrow (* \Rightarrow *)$ Kind of functions from proper types to type operators. For example, binary type constructors for two-argument operators.

We can now build rules with this kind system. It is only important to know that they exist but not how they look; they can also be found in [Pie02, Figure 29-1]. Having the kinding system and knowing that type expressions have different arities is enough to prevent meaningless expressions like $(Bool\ Nat)$. In the end, we have a new syntax to write $\Gamma \vdash T :: *$ to indicate that T must have exactly kind $*$ which means it must be a proper type. On the other side, these checks are used in the rules for building the type system. To summarize, kinds help to prevent one to build meaningless type expressions.

The whole effort was so we can write $Vector :: Nat \rightarrow *$ and the rules check the correctness of this expression. A kind is the type of a type constructor and in this case we state that *Vector* maps $k : Nat$ to a type [Pie04]. This is the crucial part: *Vector* is a type family and a parameter $k : Nat$ can be applied to it to yield a type. Normal types and functions have a single definition. Type families, on the other hand, have an interface that declares its kind and arity. We can now start building dependent types with this knowledge. The next section explains dependent types formally so we can then build practical examples in Idris in the section after.

3 Dependent Types

As said before, types become first class citizens in a dependent type system. Readers familiar with functional languages, where functions are first-class citizens, can draw the similarities. Types can be used everywhere other expressions can be used: a function can compute and return a type, types can be passed to a function, they can be computed, manipulated and assigned to a variable. An important distinction is, that *int* is a type and for example 1 is a value of type *int*. Non-dependently typed languages can already pass 1 as an argument and return it but not *int* itself. The conclusion for the compiler is that it does not matter if 0 or 1 is passed since they are both of type *int* and are equal at the level of types [Xi10]. They are interchangeable as far as type checking is concerned. Writing a function that expects a divisor must check for 0 to prevent division by zero errors. With languages supporting dependent types, we can create a type that depends on a value, that must not be zero, and then use this type in the function to permit only non-zero numbers. As we have seen before in Listing 2, we can achieve this by having type constructors for the type `Nat`. A type constructor is simply a constructor for creating new types based on existing ones. For example, constructing a list of a proper type. Listing 4 shows the two type constructors `Z` and `Suc`.

```
1 data Nat = Z | Suc Nat --zero | successor
```

Listing 4. Declaration of the type `Nat`. Excerpt from Listing 2

Upon application, they create a new type and

in the case of `Suc` this constructor depends on a value.

The other example we have already seen was the formal definition of `first` in λ-Expression (1). This expression first showed the Pi type (Π-type), also called dependent product type, without an explanation. This is now done in the following section.

3.1 Pi Type (Dependent Functions)

We can express this behavior of functions having a return type depending on the argument formally with the Pi type. We pick the type family *Vector* up again. The following examples are taken from [Pie04]. With the kinding assertion

$$Vector :: Nat \to * \qquad \text{(K-VECTOR)}$$

we introduce this type family and on application, *Vector n* yields a type dependent on *n*. For this subsection we just use a vector that holds *Int*s for simplicity. Otherwise, we would have another type argument. The compiler can access the value of *n* to check various things whenever this *Vector* is used. We now need a way to instantiate vectors to be able to use them. So we build a function *init* for that as seen in λ-Expression (INIT).

$$init : \Pi n : Nat.Int \to Vector\ n \qquad \text{(INIT)}$$

With *init* $5\ 1$ we get a vector of type *Vector* 5—a type in the type family *Vector*—with 5 elements set to the value 1. Notice that the 5 belongs to the type, like `int[5]` in Java or C++. But the value is actually used and checked at compile-time and not only at run-time. As a consequence, *init*'s return type depends on the input and to express that we used the Pi type. $\Pi x : S.T$ means that we bind the function's argument x so that we can replace all free occurrences of x in T. This is in contrast to the simply-typed λ calculus where the right hand side cannot be a type and here the result type with a Pi type can vary with different arguments. We now have the type T that depends on the value of x respectively *Vector* that depends on the value of n.

The final example is the one from the Introduction section that is repeated here:

$$first : \Pi n : Nat.Vector(n+1) \to Int \qquad \text{(FIRST)}$$

The application $first\ 0\ (init\ 0\ 1)$ results in an error. $(init\ 0\ 1)$ yields type *Vector* 0 which does not match the expected type $Vector(0+1)$. And since n is of type N the value zero is the lowest value, thus $n+1$ is always greater than zero and a non-empty vector is not possible to pass.

The Pi type is similar to $\forall X.T$ in System F that is explained in [Amr16], but the abstraction is with a term rather than a type. We now go over to build real programs with the Pi type in Idris.

4 Idris

Idris is a functional programming language with dependent types. It is a fairly recent language. The development started in 2011 and the language is still actively developed. It is compiled to C but also has backends in JavaScript, Java, LLVM or even PHP. The language's syntax is close to Haskell's, thus the basic syntax is not explained here and Haskell knowledge is preferable. [Gri15, Haskell Prerequisites] gives a good quick overview. The main difference is that Idris only uses a single colon `:` while Haskell uses a double colon `::` for type declarations.

4.1 Example of Vector Addition

Idris allows us to write a function that appends two vectors and checks at the same time the length of the resulting vectors. The following code is taken from the data type `Vect` from the Idris library [BCAS]. We first show the definition of a vector.

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

Listing 5. Idris's `Data.Vect` definition from [BCAS]

Listing 5 shows the definition of the `Vect` type family. `Nil` is used to create empty vectors and `::` to create non-empty vectors. With *line 3* we explicitly state that, when appending an element to another `Vect` with `::`, the resulting `Vect`'s length is exactly one longer. The input is a vector of length `k` (from `Vect k a`) and the resulting vector has length `S k` which is the successor of `k`. This is the verification a developer gets from Idris. It must hold for all possible values of `k`, so all natural numbers.

We can then go on by defining the before mentioned function `++` for vectors where we not only append one element but all elements from another vector. This is shown in Listing 6. On *line 2* we say the function's return type is `Vect (m + n) a` where `m` and `n` are the lengths of the input vectors.

```
1 ||| Append two vectors
2 (++) : (xs : Vect m a) -> (ys : Vect n
      a) -> Vect (m + n) a
3 (++) []       ys = ys
4 (++) (x::xs) ys = x :: xs ++ ys
```

Listing 6. Idris's `Data.Vect.++` definition from [BCAS]

Idris checks if we actually return a vector of length `m + n` in every case for every input. Listing 7 shows a counter example that does not compile because of an error in the definition of `myapp`.

```
1 myapp : (xs : Vect m a) -> (ys : Vect n
      a) -> Vect (m + n) a
2 myapp Nil       ys = ys
3 myapp (x :: xs) ys = x :: myapp xs xs --
      error
```

Listing 7. Own implementation of vector append with a type error on *line 3*

Idris' type checker will complain that there is a type mismatch since we do not actually return a vector that has always length `(m+n)`. It might in some cases, but Idris requires it for all of them.

```
Type mismatch between
 Vect (k + k) a (Type of myapp xs xs)
and
 Vect (plus k m) a (Expected type)
Specifically:
 Type mismatch between
   plus k k
 and
   plus k m
```

Notice that we did not write an explicit proof for this function. Proofs are just mentioned shortly and shown in an example. Listing 8 shows two proofs for a `Vect`. That an element is found in a vector and that an empty `Vect` does not contain anything. A big part of the `Data.Vect` file [BCAS] are actually just proofs so the compiler can ensure the correctness of a program.

```
1  ||| A proof that some element is found
       in a vector
2  data Elem : a -> Vect k a -> Type where
3       Here : Elem x (x::xs)
4       There : (later : Elem x xs) -> Elem
       x (y::xs)
5
6  ||| Nothing can be in an empty Vect
7  noEmptyElem : {x : a} -> Elem x [] ->
       Void
8  noEmptyElem Here impossible
9
10 Uninhabited (Elem x []) where
11   uninhabited = noEmptyElem
```

Listing 8. Proofs in Idris about `Vect` taken from the standard library [BCAS]. Proofs are like normal program code.

4.2 Totality

Type checking becomes undecidable when arbitrary values or expressions can be used for types to depend on. It would require checking if two types are equal which is as hard as checking if two programs return the same result. Hence, Idris allows non-total (partial) functions. A total function is one that terminates for all possible inputs or guarantees to produce some output before making a recursive call [Com].

Idris aims to be a general purpose programming language and not only a theorem prover. This is the second reason why partial functions are allowed. A developer should not always be bothered by the compiler and many programs are in an unfinished state and the developer knows that there are errors. This means, Idris does not

require a proof for every function like dependent types do. The example in Listing 9 shows a total function where a call results in a compile-time error.

```
1  total
2  safeHead : (l : List a) -> {auto ok :
       NonEmpty l} -> a
3  safeHead [] {ok=IsNonEmpty} impossible
4  safeHead (x::xs) {ok=p} = x
5
6  main : IO ()
7  main = do
8    -- compile error, can't find
9    -- value of type NonEmpty[]
10   let x : Integer = safeHead []
11   print x
```

Listing 9. Complete function in Idris that results in a compile-time error when called with an empty list. The `total` keyword indicates a total function.

Listing 10 shows the partial function that will only produce an error at run-time. Marking it with the `total` keyword would result in a compile-time error as the compiler notices the missing case for an empty list.

```
1  unsafeHead : List a -> a
2  unsafeHead (x::xs) = x
3
4  main : IO ()
5  main = do
6    let x : Integer = unsafeHead []
7    print x -- runtime error
```

Listing 10. Partial function in Idris that is not reduced when type checking because it is not well defined for all inputs and the function is not labeled with `total`

The difference and allowance of such functions may help developers have an easier beginning when starting with Idris. It may also help the popularity of the language since a proof is not necessary for each function. One, however, loses an important property, namely that the program will never fail.

5 Differences

Readers familiar with other programming languages might have had thought about similar concepts while reading this paper. C++ offers all mentioned features here also in the form of templates that we can also use during compile-time. Refinement types also allow developers to catch errors for empty vectors or null values with refined types as one can with dependent types. Also, some type dependent calculations can be done with polymorphism. This section discusses all these three topics and for each shows what can be achieved and where they differ.

5.1 C++ Templates

C++'s template language is Turing complete. This means you can do a lot of stuff during compilation like the calculation of π or checking passed arguments that they meet a requirement. To take it away, one can do everything with C++ that Idris with dependent types can with static knowledge at compile-time. Idris can additionally measure things at run-time and guarantee that this measurement is correct. We first show an example that can also be achieved with templates and then we show why C++ is not a dependently typed language. Some of the statements in this subsection are taken from [red], a discussion the author of this paper started.

5.1.1 Achieving the Same

For example, we can use C++'s built-in templates like `std::integral_constant` to represent a value of a specific type at compile time. With its help and `std::array`—which has a fixed length—we can rebuild the dependent function `init` shown in λ-Expression (INIT).

```
1  template <typename T, T V>
2  using i_con =
       std::integral_constant<T, V>;
3
4  template<size_t n, int i>
5  constexpr std::array<int, n>
       init(i_con<size_t, n>,
       i_con<int, i>) {
6    return  std::array<int,n> {/*
       fill array with i */};
7  }
8
```

```
9  int main() {
10   std::array<int,3> foo =
       init(i_con<size_t,3>{},
       i_con<int, 42>{});
11 }
```

Listing 11. Create an `std::array` with a length based on a value at compile-time in C++

Listing 11 shows this `init` function in C++ that creates an `std::array` with a size depending on the passed value. Everything is done at compile-time and assigning it instead to `std::array<int,4>` on *line 10* would result in a compile error. Passing it to a function that expects an array of length 4 would also result in a compile error. We could even compute types and return them conveniently with the Boost.Hana library [Dio].

C++ templates have a lot of static computational power. Its type system, however, is relatively weak. A type is less expressive and less powerful than a dependent type.

5.1.2 What Is Actually the Difference Now?

So, where is the difference? Does this mean that C++ is dependently typed? No, it does not. Even though we can achieve the same at compile-time as with a language with a dependent type system like Idris, C++ is different. The before mentioned point about run-time behavior is something C++ does not have. A C++ compiler just checks all explicitly passed values to a function while Idris checks for all possible inputs and that the implementation is correct for all those possibilities. Even those that will just arise at run-time since a dependent type captures a concept of what should hold. Every function belonging to a type will then amount to proving that it meets the type's specifications. We have a proof that it holds and not just a check for passed values as with C++. A first-class type embodies this proof and all properties with it and we can pass it around without manually checking it. The Idris compiler does this for us and if a property does not hold, the program does not compile.

So what can you not do with C++ because of that? Implementing an insertion sort that is proven to be correct on all possible inputs at compile-time. This is possible in Idris as shown in [Fos]. We could verify it for all checked values. Well, one could write a proof assistant with templates, but that would be an overhead. This highlights the difference the best. C++ could do a lot that dependent types do, but the dependent type system is specifically built for providing proof that an implementation is correct while C++'s type system is not. But one could help himself by writing a lot of template code. Dependent types are like a new programming paradigm that brings convenience and simplicity. Like non-functional languages that can do and simulate the same things as functional languages, but a functional language might be easier to use and better suited for a specific task.

To summarize, C++ can do a lot of static checking and constructing but the value needs to be known at compile-time. This boundary does not exist with a dependently typed language. A dependent type system knows limitations and properties a type holds and can check it implicitly without the necessity of explicit checks as they are necessary in C++. This makes dependent types more convenient to program with for some problems. C++ just allows building something with the template system that simulates some dependent type features but it is not built into the language. Types as first-class citizens is an unknown concept in C++.

5.2 Refinement Types

In short, dependent types are more powerful than refinement types. Refinement types allow to—as the name says—*refine* existing types. A type consists of all the values of a given type which satisfies a given predicate.

$$\{v : B \mid p\} \qquad \text{(Refinement)}$$

In (Refinement) , v has a base type B—like *Int* or *Bool*—and p is a predicate that must hold true for all values of the type [Rei16]. For example, $\{n : int | n > 42\}$ inhabits all integer values that are bigger than 42. One can achieve the same result for trivial cases as with dependent types, but

there are two big differences between these two systems.

First, refinement types are limited to decidable logics that is why they can offer much more automation and type inference [Jha]. Refinement type systems only allow verification conditions that can be efficiently validated by a Satisfiability modulo theories (SMT) Solver [VSJ+14]. This automates the solving of the constraints to check whether a program is well- or ill-formed. Dependent types, on the other hand, need proof terms as can be seen in an insertion-sort implementation in Idris [Fos]. This implementation contains a lot of these proof terms, that actually look like program code, to verify that the insertion sort actually gives back a sorted list and contains the same elements that were passed as inputs. The implementation in LiquidHaskell with refinement types is much shorter [Jha].

Second, refinement types just have their predicate they depend upon. For `{n : int | n > 42}` the type just contains all values that are bigger than 42. With dependent types, you can write anything in a type that you can normally write in an expression. We can compute a type as we saw in the introduction with the `isSingleton` function that is again shown in Listing 12.

```
1 isSingleton : Bool -> Type
2 isSingleton True = Nat
3 isSingleton False = List Nat
```

Listing 12. Function `isSingleton` in Idris that computes a type and returns it. This cannot be done with refinement types.

5.3 Polymorphism

Some dependent type features can be modeled with polymorphism. Polymorphism allows us to write functions that work for different types—we write *terms that abstract over types*. For dependent types, we write types that depend on terms—*types abstract over terms*. This can also be seen in the λ-cube in Figure 1 by Barendregt [Bar91]. Polymorphism is represented by λ2 and dependent types by λΠ on a different axis of the cube. This means they have almost nothing in common.

Polymorphism allows a function to behave differently on different types. One can use function overloading so the function behaves differently for different types and we even get a different return type depending on the passed type as shown in Listing 13.

```
1 public int foo(char c) {
2   return (int) c;
3 }
4
5 public String foo(int i) {
6   return Integer.toString(i);
7 }
```

Listing 13. Ad hoc polymorphism in Java. We can differentiate between two types and depending on the type we return a different type. These are however two completely different functions.

We could also have the implementation behave differently for different types by subtying. This is however everything we can achieve with polymorphism. We cannot compute types and return types that depend on a value. Notice that a function that returns a `Vector<int>` with a specific length does not compute a type. Either the type is passed explicitely or it is static and not depending on any passed argument type and we return an instance of this type and not the type itself.

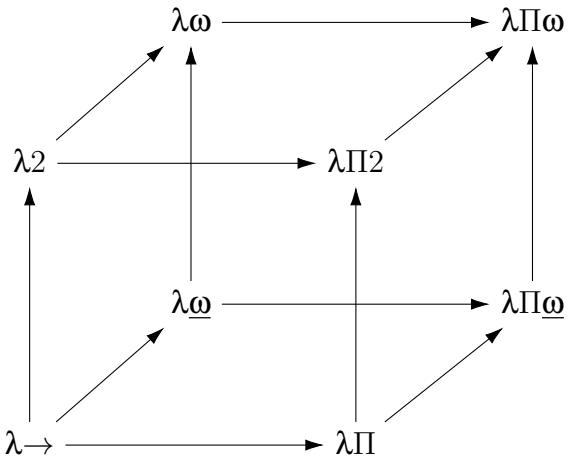

Figure 1. λ-cube by Barendregt showing different abstractions which are in the end just features that a programming language can have [Bar91]

6 Relevancy

Dependent types are not in any mainstream language yet. C++ has some features with non-type template parameters. Scala has path-dependent types which offer some functionality but they are more limited than dependent types. This section shows another more popular language that wants to embody dependent types (Haskell) and shows where dependent types are otherwise currently used.

6.1 Dependent Types in Haskell

Haskell, as implemented in the GHC has been adding new type-level programming features for some time [Eis16]. All of which aim to bring Haskell to dependent types. The GHC compiler unified types and kinds with version 8.0.1 [Gam]. The reasoning behind this is explained in [Eis]. The possiblity to use all types as kinds is a step towards this goal. The following quote is from [WHE13].

> *Is Haskell a dependently typed programming language? Many would say no, as Haskell fundamentally does not allow expressions to appear in types (a defining characteristic of dependently-typed languages). However, the type system of the Glasgow Haskell Compiler (GHC), Haskell's primary implementation, supports two essential features of dependently typed languages: flow-sensitive typing through Generalized Algebraic Datatypes (GADTs), and rich type-level computation through type classes, type families, datatype promotion and kind polymorphism. These two features allow clever Haskellers to encode programs that are typically reputed to need dependent types.*

As the authors said, developers can "fake" some aspects of dependent types in Haskell. [McB01] explains how this can be done by making type-level copies of data, type constructors simulating data constructors and type classes simulating datatypes. It is however painful to do.

6.2 Proof Checking and Other Uses

Dependent type systems are also used for proof checking. Idris is also used for this but aims to be more than that. It features system calls, UIs and concurrency language tools and tries to bridge the gap. Idris is already used in scientific computing [IJ13]. There is also a paper discussing real world questions such as efficiency and concurrency [BH10].

6.3 Outlook

Developers would have to learn to prove their functions which is unfamiliar. However, this is basically just like writing code, but you have to think different. When dependent types were learned by new students, they would not find it to be a problem but people that already can program do not see the advantage and the need to switch. A carefully designed new language could help in raising popularity. Dependent types would get rid of unit tests completely. While unit tests are good, they do not always test each branch, test only extreme values and are sometimes forgotten and not updated. The type checker in a dependently typed language would watch out for such pitfalls.

7 Conclusion

Dependent types extend a type system with types that depend on a value. It is not possible to "just add this" to an existing language. As the example of Haskell shows, there are big efforts required to add them to a language [Eis16]. It shows however, that it is possible to add them to a language while still being backward compatible.

The motivation is there and reasonable. It is easy to write a type checker but hard to write a performant one with good error messages and support for modern IDEs. Agda and Idris (dependently typed languages) mode for Emacs are more promiment examples that deliver some IDE like features for dependent types. Does this mean that dependently typed languages could ever be a popular choice for everyday programming? Benjamin Pierce [Pie02] said the following in 2002:

> They [mathematicians] spend a significant effort writing proof scripts and tactics to guide the tool in constructing and verifying a proof. [...] programmers should expect to expend similar amounts of effort

> annotating programs with hints and explanations to guide the typechecker. For certain critical programming tasks, this degree of effort may be justified, but for day-to-day programming it is almost certainly too costly. [...] The trend in those languages is toward restricting the power of dependent types in various ways, obtaining more tractable systems for which type checking can be better automated.

The current answer is also probably no, they cannot be a popular choice for everyday programming. At least not if they only offer dependent types but not much more. Such languages are and will most likely be used for theorem proving. If they are coupled with other paradigms and not every little obvious detail has to be proven then there is a good chance that dependent types will be used for everyday and business programming in the future. On the other side, we lose the most important property when not everything needs to be proven as this is the case with Idris and possible partial functions. A good way would be to have STM solvers for carrying easy proofs that are repetitive.

The strong desire for Haskell for dependent types and the upcoming of new dependently typed languages such as Idris [idr] and ATS [Xi] shows the interest in it. Functional programming languages are slowly on the rise in smaller companies after they have been around for decades. Maybe dependently typed languages will also become more popular in 20 or more years after they have been better researched and used for many years outside the industry.

References

[Amr16] Christoph Amrein. Simply Typed Lambda Calculus with Parametric Polymorphism, 2016.

[Bar91] Hendrik Pieter Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[BCAS] Edwin Brady, David Christiansen, and Ahmad Salim Al-Sibahi. Idris Vect Implementation on Github. `https://github.com/idris-lang/Idris-dev/blob/61d5865afcbb65eb79125ddfdb23dcb0ec77f181/libs/base/Data/Vect.idr`. [Online; accessed 29-November-2016].

[BH10] Edwin Brady and Kevin Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundam. Inf.*, 102(2):145–176, April 2010.

[Bra] Edwin Brady. @edwinbrady on Twitter. `https://twitter.com/edwinbrady/status/431415892233428992`. [Online; accessed 29-October-2016].

[Com] The Idris Community. Types and Functions — Idris documentation. `http://docs.idris-lang.org/en/latest/tutorial/typesfuns.html`. [Online; accessed 25-November-2016].

[Dio] Louis Dionne. Boost.Hana: User Manual. `http://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/index.html`. [Online; accessed 21-December-2016].

[Eis] Richard Eisenberg. Planned Change to GHC: merging types and kinds | Types and Kinds. `https://typesandkinds.wordpress.com/2015/08/19/planned-change-to-ghc-merging-types-and-kinds/`. [Online; accessed 7-November-2016].

[Eis16] Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016. unpublished thesis.

[Fos] David Foster. Idris insertion-sort Implementation on Github. `https://github.com/davidfstr/idris-insertion-sort/blob/136cb0f53b7dbc705e59e1ea1cb579cbd9e169e5/InsertionSort.idr`. [Online; accessed 20-December-2016].

[Gam] Ben Gamari. GHC 8.0.1 is available! `https://ghc.haskell.org/trac/ghc/blog/ghc-8.0.1-released`. [Online; accessed 7-November-2016].

[Gri15] Jannis Grimm. Curry-Howard Isomorphism Down to Earth, 2015.

[idr] Idris | A Language with Dependent Types. `http://www.idris-lang.org/`. [Online; accessed 13-October-2016].

[IJ13] Cezar Ionescu and Patrik Jansson. Dependently-typed programming in scientific computing. In Ralf Hinze, editor, *Implementation and Application of Functional Languages: 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, pages 140–156. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Jha] Ranjit Jhala. Programming with Refinement Types—Insertion Sort in LiquidHaskell. `http://ucsd-progsys.github.io/lh-workshop/04-case-study-insertsort.html#/ordered-lists`. [Online; accessed 24-November-2016].

[McB01] Connor McBride. Faking It: Simulating Dependent Types in Haskell, 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[red] Discussion on reddit's /r/dependent_types. `https://www.reddit.com/r/dependent_types/comments/5j68ww/what_can_i_do_with_dependent_types_that_i_cannot/`. [Online; accessed 20-December-2016].

[Rei16] Micha Reiser. Liquid Type Inference Under the Hood, 2016.

[Syf15] Marco Syfrig. Typed Lambda Calculus, 2015.

[VSJ+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, August 2014.

[WHE13] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with Explicit Kind Equality. *SIGPLAN Not.*, 48(9):275–286, September 2013.

[Xi] Hongwei Xi. The ATS Programming Language. `http://www.ats-lang.org/`. [Online; accessed 29-November-2016].

[Xi10] Hongwei Xi. Introduction to Programming in ATS. `http://ats-lang.sourceforge.net/DOCUMENT/INT2PROGINATS/PDF/main.pdf`, 2010. [Online; accessed 25-November-2016].