Write less, type more
or: How I learned to stop worrying and love the compiler

Niklas Haas

Abstract

Dependent types extend type systems by adding dependent functions and dependent pairs, in which one component can depend on the value of the other (rather than just the type). In these type systems, more programs can be expressed, and existing programs can be restated with increased type safety. Dependent types allow coupling a program with its own proofs of correctness, which enables dependently typed languages to be used as automated proof checkers. In this paper we seek to provide a brief introduction to the syntax, semantics and use cases of dependent type systems by using Haskell as an example and extending it with dependent typing.

1 Introduction

Dependent types are a feature of *type systems*, which are rigorous systems for categorizing terms in order to eliminate absurdities and paradoxes. In the context of a programming language, a type system is responsible for rejecting "invalid" programs – programs that have nonsensical denotational semantics.[1] Every type system must make certain trade-offs between false positives (programs that are nonsensical yet allowed nonetheless) and false negatives (programs that make sense but aren't expressible within the constraints imposed by the type system). Traditionally, programming languages with nonexistant or very liberal type systems (like C or Lisp) have erred on the side of false positives, permitting more programs at the cost of allowing undefined behavior, runtime errors or unsafety. In contrast to this, some recent programming languages such as Standard ML[11] and Haskell[7] include much stronger type systems, but they come with their own cost of disallowing some "valid" programs.[2] The goal of dependent types is to improve both of these aspects, by not only making it possible to write much stronger (and hence safer) types but also to write dependent types that could otherwise not be expressed.

In this paper, we wish to provide a basic overview of dependent types, including where they come from and how they are used. We use a simplified Haskell-like language to provide examples throughout, serving to both outline the shortcomings of Haskell as well as to demonstrate how dependent types can help solve these. We also briefly explore the connections of dependent types to predicate logic, as seen in the framework of automated proof assistants.

[1]This means that they have no useful "meaning", e.g. a program which tries to subtract an integer from a string, divide by zero or order a negative number of shoes.

[2]An example based on `printf` is provided in section 4

2 History and foundation

Before going into the development of dependent typing, it should serve well to explain what the term dependent type actually refers to. It is not a theory per se, but rather a broad label used to refer to any sort of system in which types can "depend" on other values.

A common way to do this is using axioms originally introduced by Per Martin-Löf in his *Intuitionistic Type Theory*[8], which add two new concepts of type abstraction: so-called Π-Types and Σ-Types.[3] Languages which are based on this include proof assistants such as NuPRL[5], LEGO[6], Coq[2] and more recently, programming languages such as Cayenne[1], ATS[17], Epigram[10] and Agda[13]. Alternative foundations exist, but we will not discuss them in greater detail.[4]

Per Martin-Löf's theory was originally developed to serve as a general framework for mathematics in order to deal with shortcomings of systems such as *axiomatic set theory* as well as to bridge the gap between mathematical logic and programming languages.[8] Examples of how to develop a programming language based on intuitionistic type theory are given by [12], and the first actual programming language to use these ideas is Cayenne, which was developed to deal with shortcomings of Haskell.[1]

3 Background

In order to provide familiar examples of dependent typing in action, we will base our examples upon a theoretical extension to the Haskell programming language. This section will serve to establish some syntactical conventions and provide a very short outline of how to read them.

```haskell
data List :: * -> * where
  Nil  :: forall (a :: *). List a
  Cons :: forall (a :: *). a -> List a -> List a

length :: forall (a :: *). List a -> Integer
length Nil          = 0
length (Cons _ xs) = 1 + length xs
```

Figure 1: An example of Haskell syntax, defining a custom list data type and a function to compute its length. A list is either an empty list (`Nil`) or a value prefixed to another list (`Cons`). The length of an empty list is 0, and the length of a value prefixed to another list is 1 + the length of the other list.

Every polymorphic type will be explicitly quantified, using Haskell's `forall` syntax. This is not normally done in Haskell because type abstraction and instantiation is automatic, but

[3]These are also called (dependent) product types and sum types, respectively, but this terminology is overloaded.

[4]An example is Twelf, which is based on the λΠ-calculus.[14]

this is no longer true in general when adding dependent types. In our examples, however, we assume this was still the case and simply use `forall` to note the presence of a parameter which will be automatically instantiated.[5] In regular Haskell, the `length` function shown in figure 1 could equivalently have been given the following type signature: `length :: List a -> Int` which is equivalent. A good way to read `v :: forall a. T` is "`v` has type `T`, given any `a`". For example, "`Nil` has type `List a`, given any type `a` (itself of type `*`)". Note that like Haskell we use `::` for type signatures, not `:` which is more common among mathematical literature and dependently typed languages.

Type variables quantified in this way will be given a type signature of their own. The special type `*` refers to the type of regular data types, for example `Integer :: *` and `Char :: *`. A type which takes *another* type as variable would itself have type `* -> *`, for example `Maybe :: * -> *` and application works as usual, eg. `Maybe Integer :: *`.[6] Roughly speaking, a value must have a type which is itself contained in `*` for it to actually "exist". For example, one could compute and pass around a concrete value of type `Maybe Integer :: *`, but it makes little sense to think about a value of type `Maybe` – maybe *what*?

Data type definitions will always be given in so-called *generalized algebraic data type* (GADT) form. This makes it explicit not only what the type of a type constructor is but also what the types of its value constructors are. The definition for `List` given in figure 1 could have been written `data List a = Nil | Cons a (List a)`, but this syntax isn't flexible enough to describe dependent data types.[7]

4 Basics of dependent typing

In this context, we will now introduce the two key concepts of dependent typing along with motivational examples.

Suppose we wish to write a function that works like C or Java's `printf`, in Haskell. The way `printf` works is by accepting a *format string* as first parameter, which describes what kinds of further parameters are required by the function as well as how to output them. For example, the code

```
string a = "foo"
int b = 42
printf("a = %s, b = %d", a, b)
```

would output `a = foo, b = 42`.

The problem with trying to implement a function like this in Haskell arises as soon as we begin to think about what type signature it would have in general. For example:

[5]Due to the practical benefits of automatic type instantiation, most dependently typed programming languages support something similar to this for dependent functions – usually called *hidden* or *automatic* parameters.

[6]We assume that `* -> * :: *` due to convenience, however this gives rise to logical paradoxes similar to the untyped lambda calculus. In actual dependently typed languages, there is generally a hierarchy of types $*_0, *_1, ..*_n$ similar to axiomatic set theory.

[7]This is not strictly true in Haskell. For example, one could introduce type equivalence constraints and existential constraint quantification, but this is needlessly complex and comes with its own difficulties.

```
printf "a = %s, b = %d" :: String  -> Integer -> String
printf "%d %d %d"       :: Integer -> Integer -> Integer -> String
```

As we can clearly see, the type of `printf s` depends on the contents of `s` itself – which could be anything. However, Haskell's type system is *static* – meaning types have to be fixed at compile time, and cannot vary based on the value of something not known in advance. As a result of this, there is not a possible type we could give to `printf :: String -> ?` in Haskell.

4.1 Type-level computations

The first thing we notice is that the result type of `printf` can be computed from a `String`, that is, we could in theory write a function as seen in figure 2.

```
printfType :: String -> *
printfType ""                = String
printfType ('%':'d':rest) = Integer -> printfType rest
printfType ('%':'s':rest) = String  -> printfType rest
printfType (_:rest)          = printfType rest
```

Figure 2: A function to compute `printf`'s type given a `String`.[1] An empty format string implies the end of evaluation, which results in the accumulated `String`. Passing a format that begins with `%d` or `%s` means the function has to take at least one extra parameter, of type `Integer` (resp. `String`). Passing anything else just outputs that character and moves on to the rest of the format.

Dependent typing adds the ability for functions to accept and return *types*, in addition to values, as well as the ability to use the result of functions within a type signature, for example:

```
simple :: printfType "%d"
simple = \(n :: Integer) -> show n
```

This could have equivalently been written `simple :: Integer -> String` since the result of `printfType "%d"` is constant and hence fixed at compile time.[8]

However, this alone does not suffice to give a type to `printf` because even if we were to write `printf :: String -> printfType ?` we still don't know what to fill in for the `?`. This is due to the fact that the `?` is not constant but instead *depends* on the actual value passed to the function.

[8]Indeed, Haskell with GHC extensions can already represent and work with functions similar to this – except they're called *type (synonym) families* and come with their own syntax and rules.

4.2 Dependent functions

Dependent typing deals with this shortcoming by making it possible for us to write type signatures of the form `(a :: A) -> B` where the type `B` can involve the name `a`. If `B` does not mention `a`, this would be equivalent to `A -> B` and type-checks similarly. The name `a` assigned to the parameter is irrelevant other than allowing it to be mentioned in the type of the result. An example of this as applied to `printf` can be seen in figure 3.

```
printf :: (fmt :: String) -> printfType fmt
printf fmt = go fmt ""
  where go ""              output = output
        go ('%':'d':cs) output = \(n :: Integer) ->
                                     go cs (output ++ show n)
        go ('%':'s':cs) output = \(s :: String)  ->
                                     go cs (output ++ s)
        go (c:cs)        output = go cs (output ++ [c])
```

Figure 3: An example of a dependently typed `printf` function.[1] The function is defined recursively, using `go` to step through the format while accumulating output. When the end of the format is reached, the output is just returned as-is. When a `%d` or `%s` is reached, we return a function that takes one `Integer` or `String`, formats it, and appends that to the output (while moving on to the rest of the format). Finally, if any other character is reached, that simply gets appended to the output. The function `printfType` computes precisely the type necessary to make this function type-check.

This form of type dependency is an example of intuitionistic type theory's "dependent functions", formally known as Π-types.[8] The result type of a dependent function can *depend* on the value of the parameter passed to it.

4.3 Lists with known length

These kinds of dependency are not only allowed in function type signatures, but also on constructors of data structures. For example, we could define a type of lists with fixed lengths, also called *vectors* in the dependently-typed world[1], as seen in figure 4.[9]

The type of `head` as presented here is particularly interesting. In Haskell, we're faced with the problem that `head []` throws an exception, yet `head :: forall (a :: *). [a] -> a` clearly permits applications of itself to `[] :: forall (a :: *). [a]`. With our dependently typed `head` function, the expression `head VNil` is a static type error, because `head` requires a `Vec` of length `n+1`, yet `VNil` has length `0` as specified in the signature. Since $0 = n+1$ has no solution for $n \in \mathbb{N}$, this parameter `n` cannot possibly be chosen by the type checker.

[9]In this example, we assume `Natural` is a type like `Integer` but which only permits positive values.

```
data Vec :: Natural -> * -> * where
  VNil  :: forall (a :: *). Vec 0 a
  VCons :: forall (a :: *) (n :: Natural).
           a -> Vec n a -> Vec (n+1) a

replicate :: forall (a :: *).
             (n :: Natural) -> a -> Vec n a
replicate 0 _ = VNil
replicate n x = VCons x (replicate (n-1) x)

head :: forall (a :: *) (n :: Natural).
        Vec (n+1) a -> a
head (VCons x _) = x

tail :: forall (a :: *) (n :: Natural).
        Vec (n+1) a -> Vec n a
tail (VCons _ xs) = xs
```

Figure 4: An example of dependently typed vectors. A vector is defined like a list, except for the presence of an extra type parameter, of type `Natural`. A `VNil` has the length 0, a `VCons x xs` has the length of `xs` $+ 1$. The function `replicate` produces a vector of given length, it works like the equivalent function on lists except for the extra type parameter, which is captured from the passed length `n` using dependent typing.

This is an example of additional type safety that can be achieved very easily with the help of dependent typing.

4.4 Dependent pair

In addition to dependent functions like `(a :: A) -> B`, dependently typed programming languages introduce so-called *dependent pairs* as in the type `(a :: A, B)` where `B` involves `a`. For example, suppose we want to implement traditional lists using our newly defined `Vec`. However, a list can be of *any* length, that is to say, a value of type `List a` can be a vector of type `Vec n a` for *some* `n :: Natural`. Using dependent pairs, this could be implemented as in figure 5.[10] Formally, these are called Σ-types in intuitionistic type theory.[8]

As demonstrated, they're mainly useful when programming to carry around otherwise dependently typed structures when we *don't* care about their properties, or when we want to abstract over different possible implementations of something – for example `type Widget = (t :: *, t, t -> Output, Input -> t -> t)` is the type of "things" that can respond to Input and produce Output, where we don't care about the internal representation of `t`. (This is

[10]This could also have been implemented using existential quantification in Haskell.

```
type List a = (n :: Natural, Vec n a)

listify :: forall (n :: Natural) (a :: *).
           Vec n a -> List a
listify v = (length v, v)
  where length VNil          = 0
        length (VCons _ xs) = 1 + length xs
```

Figure 5: Lists implemented in terms of dependently typed vectors. A `List` is just `Vec n` of some arbitrary length n, which is stored alongside it. To convert a vector into a list, we compute its length and store this as the required `Natural`.

equivalent to the non-dependent `type Widget = (Output, Input -> Widget)` if given general recursion, but that language feature is impossible in languages that aren't turing complete.)

5 Implications of dependent typing

5.1 Turing completeness concerns

The most immediate effect of allowing types to depend on and be computed from values is that in any Turing-complete language, due to limitations imposed by the halting problem, it is impossible to write a general purpose type checking algorithm that will always terminate with a result. Instead, there is the possibility that the type checking process will loop and continue forever.[1]

Some programming languages choose to ignore this possibility, for example Cayenne only provides an adjustable recursion limit during type checking to catch obvious errors but otherwise makes no attempt to guarantee type checking will always succeed.[1]

Other dependently typed systems, especially those intended to be used as proof assistants (such as Coq), instead impose a provable termination condition on functions – that is, recursion is only allowed within certain parameters that the compiler can prove always lead to a result. A direct consequence of this is that these languages are *not Turing-complete*! Hence, they cannot strictly be used as "general purpose programming languages". In practice, even this restriction is not that problematic because many ways have been developed to represent common types of recursion in terms of patterns that are known to terminate.

Most languages, such as Idris or Agda, allow one to choose between these two compromises by having termination checking enabled by default but allowing one to selectively disable them for specific functions or files, in which case the compiler simply assumes it actually terminates.

5.2 Curry-Howard correspondence

As originally noted by Haskell Curry in [4], the simply typed lambda calculus can be used to model simple intuitionistic logic – a value of type `T` can be seen as a proof that `T` is inhabited ("true"), and a function of type `A -> B` can be seen as a logical implication that given a proof for `A` there exists a proof for `B`, or $A \to B$. In this context, a pair type like `(A, B)` is like a proof for *both* `A` and `B`, or $A \wedge B$, and a disjoint union type like `Either a b` is like a proof for *either* `A` or `B`, or $A \vee B$.

In this system, the two forms of dependent types correspond to intuitionistic predicate logic's universal and existential quantifications. That is to say, `(a :: A) -> P a` corresponds to $\forall a : A.P(a)$ and `(a :: A, P a)` corresponds to $\exists a : A.P(a)$. This greatly extends the possibilities of using programs as proofs, which is the foundation of dependently typed proof assistants such as Coq.

5.3 Program derivation

With the help of dependent types, it is possible to write type signatures for functions that are so strict that there is only one or a handful of possible ways to implement them. Based on these restrictions, some programming assistants such as those for Idris and Agda can *automatically* implement the function, as well as provide interactive programming via automatic case splits, holes and environments.[11]

Some proof assistants, such as Coq, provide a number of so-called program derivation *tactics* that can automatically come up with even elaborate proofs of properties, based on axioms and postulates introduced by the programmer. In practice, this can make programming in dependently typed languages efficient and fun.

5.4 Impact and future work

Due to the amounts of safety that can be obtained from type-level computations, these features of dependent typing have begun to make their way into non-dependently typed languages such as GHC Haskell. Within certain parameters, GHC Haskell can even simulate full dependent typing.[9]

Anecdotally, most recent efforts related to dependently typed programming languages seem to go into both Agda and Idris, the latter of which is closer to Haskell. Meanwhile, Microsoft is proposing F*, which is based on F# but dependently typed.[15] Other dependently typed languages worth mentioning are Ur, which is a domain specific language designed for writing provably safe web applications;[3] Dependent ML which is an extension of the ML family and Epigram, which comes with an interactive IDE and unconventional two-dimensional syntax.[10]

Dependent type systems enjoy an active front of research and experimental languages developed to support it. Current topics of interest include alternative or additional foundations (such as homotopy type theory[16]), as well as efforts on how to bridge the gap between existing programming languages and dependently typed ones without imposing major constraints or sacrificing provable termination (e.g. using inductive *codata* types to model recursion).

[11]Some of these apply to and are possible for Haskell as well, but the range of possibilities is less rich.

Certainly, a lot remains to be clarified and experimented with and dependent typing may or may not gradually find its way moving into mainstream languages, but as of 2014 it remains a topic of experimental and research languages.

Perhaps in a near or distant future, we will *all* be programming in languages that allow us to write less code and more type signatures, having our compilers not only infer the rest but also automatically prove the result correct.

References

[1] Lennart Augustsson. Cayenne – A language with dependent types. *Lecture Notes in Computer Science*, 1608, 1999.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[3] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Conference on Programming Language Design and Implementation*, ACM Special Interest Group on Programming Languages, 2010.

[4] Haskell Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[5] Paul Jackson. The NuPRL proof development system. *Reference manual and Users's Guide (Version 4.1)*, 1994.

[6] Zhaohui Luo and Robert Pollack. *LEGO proof development system: User's manual.* University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1992.

[7] Simon Marlow et al. Haskell 2010 language report. 2010. http://www.haskell.org/onlinereport/haskell2010.

[8] Per Martin-Lof and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, 1984.

[9] Conor McBride. Faking it (simulating dependent types in Haskell). *Journal of Functional Programming*, 12(4& 5):375–392, 2002. Special Issue on Haskell.

[10] Conor McBride. Epigram, 2004. http://www.dur.ac.uk/CARG/epigram.

[11] Robin Milner. *The definition of standard ML: revised.* MIT press, 1997.

[12] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[13] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[14] Frank Pfenning and Carsten Schuermann. Twelf user's guide. Technical report, version 1.2. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998.

[15] Microsoft Research. The F* Project. https://research.microsoft.com/en-us/projects/fstar/.

[16] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[17] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.