Simply Easy!

An Implementation of a Dependently Typed Lambda Calculus

Andres Löh
University of Bonn
loeh@iai.uni-bonn.de

Conor McBride Wouter Swierstra
University of Nottingham
ctm@cs.nott.ac.uk wss@cs.nott.ac.uk

Abstract

We present an implementation in Haskell of a dependently-typed lambda calculus that can be used as the core of a programming language. We show that a dependently-typed lambda calculus is no more difficult to implement than other typed lambda calculi. In fact, our implementation is almost as easy as an implementation of the simply typed lambda calculus, which we emphasize by discussing the modifications necessary to go from one to the other. We explain how to add data types and write simple programs in the core language, and discuss the steps necessary to build a full-fledged programming language on top of our simple core.

1. Introduction

Most Haskell programmers are hesitant to program with dependent types. It is said that type checking becomes undecidable; the phase distinction between type checking and evaluation is irretrievably lost; the type checker will always loop; and that dependent types are just really, really, hard.

The same Haskell programmers, however, are perfectly happy to program with a ghastly hodgepodge of generalized algebraic data types, multi-parameter type classes with functional dependencies, impredicative higher-ranked types, and even data kinds. They will go to great lengths to avoid dependent types.

This paper aims to dispel many misconceptions Haskell programmers may have about dependent types. We will present and explain a dependently-typed lambda calculus λ_Π that can serve as the core of a dependently-typed programming language, much like Haskell can be based on the polymorphic lambda calculus F_ω. We will not only spell out the type rules of λ_Π in detail, but also provide an implementation in Haskell.

To set the scene, we examine the simply-typed lambda calculus (Section 2). We present both the mathematical specification and Haskell implementation of the abstract syntax, evaluation, and type checking. Taking the simply-typed lambda calculus as starting point, we move on to a dependently typed lambda calculus (Section 3). Inspired by Pierce's incremental development of type systems [21], we highlight the changes, both in the specification and the implementation, necessary to shift to a dependently typed lambda calculus. Perhaps surprisingly, the modifications necessary are comparatively small. The resulting implementation of λ_Π is less than 150 lines of Haskell code, and can be generated directly from this paper's sources.

The full power of dependent types can only show if we add actual data types to the base calculus. Hence, we demonstrate in Section 4 how to extend our language with natural numbers and vectors. More data types can be added using the principles explained in this section. Using the added data types, we write a few example programs that make use of dependent types, such as a vector append operation that keeps track of the length of the vectors.

Programming in λ_Π directly is tedious due to the spartan nature of the core calculus. In Section 5, we therefore sketch how to proceed if we want to construct a real programming language on top of our dependently-typed core. Many aspects of designing a dependently-typed programming language in which one can write large, complex, programs, are still subject of ongoing research.

While none of the type systems we implement are new, we believe that our paper can serve as a gentle introduction on how to implement a dependently-typed system in Haskell. The λ_Π calculus has the nature of an internal language: it is explicitly typed, requires a lot of code that one would like to omit in real programs, and it lacks a lot of syntactic sugar. However, the language being explicit also has its merits: writing simple dependently-typed programs in it can be very instructive and reveal a lot about the behaviour of dependently-typed systems. We have therefore included code in the paper sources that provides a small interpreter around the type system and evaluator we describe, together with a few example functions, to serve as a simple environment to play with and perhaps extend. If you want more, we have included pointers in the conclusions (Section 6) to more advanced programming environments that allow dependently-typed programming right now.

2. Simply Typed Lambda Calculus

Roughly speaking, Haskell's type system can be divided into three levels: expressions, types and kinds. Programmers write expressions, and the type checker ensures they are well-typed. The type language itself is extremely rich. For instance, data types, such as lists, can abstract over the type of their elements. The type language is so complex that the types themselves have types, called kinds. The kind system itself is relatively simple: all types inhabited by expressions have kind $*$; type constructors, such as lists, have a 'function kind', in the same was as lambda expressions have a 'function type.'

With its three levels, Haskell is a rich version of the typed lambda calculus called F_ω, which also forms the basis of the core language used in GHC. Compared to pure F_ω, full Haskell is augmented with lots of additional features, most notably the facility to define your own data types and a cunning type inference algorithm.

In this section, we consider the simply-typed lambda calculus, or $\lambda_\to$ for short. It has a much simpler structure than F_ω as there

$$\frac{e \Downarrow v}{e :: \tau \Downarrow v} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e_1 \Downarrow \lambda x \to v_1 \quad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow v_1[x \mapsto v_2]} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow n_1\ v_2} \qquad \frac{e \Downarrow v}{\lambda x \to e \Downarrow \lambda x \to v}$$

Figure 1. Evaluation in $\lambda_\to$

is no polymorphism or kind system. Every term is explicitly typed and no type inference is performed. In a sense, $\lambda_\to$ is the smallest imaginable statically typed functional language.

2.1 Abstract syntax

The type language of $\lambda_\to$ consists of just two constructs:

$\tau ::= \alpha$		base type
	$\mid \tau \to \tau'$	function type

There is a set of base types α; compound types $\tau \to \tau'$ correspond to functions from τ to τ'.

$e ::= e :: \tau$		annotated term[1]
	$\mid x$	variable
	$\mid e_1\ e_2$	application
	$\mid \lambda x \to e$	lambda abstraction

There are four kinds of terms: terms with an explicit type annotation; variables; applications; and lambda abstractions.

Terms can be evaluated to values:

$v ::= n$		neutral term
	$\mid \lambda x \to v$	lambda abstraction
$n ::= x$		variable
	$\mid n\ v$	application

A value is either a *neutral term*, i.e., a variable applied to a (possibly empty) sequence of values, or it is a lambda abstraction.

2.2 Evaluation

Evaluation rules are given in Figure 1. The notation $e \Downarrow v$ means that e directly evaluates to v. The rules we present will evaluate a term to its normal form. As a result, unlike Haskell, we will continue to evaluate under a lambda. Type annotations are ignored during evaluation. Variables evaluate to themselves. The only interesting case is application. In this case, it depends whether the left hand side evaluates to a lambda abstraction or to a neutral term. In the former case, we β-reduce. In the latter case, we add the additional argument to the spine.

The evaluation rules given here are strict. In an application, we always evaluate the argument. This is different from Haskell's non-strict semantics. However, the simply-typed lambda calculus is strongly normalizing: evaluation terminates for any term, and the resulting value is independent of the evaluation strategy.

Here are few example terms in $\lambda_\to$, and their evaluations. Let us write id to denote the term $\lambda x \to x$, and const to denote the term $\lambda x\ y \to x$, which we use in turn as syntactic sugar for $\lambda x \to \lambda y \to x$. Then

$$(\mathsf{id} :: \alpha \to \alpha)\ y \Downarrow y$$

$$(\mathsf{const} :: (\beta \to \beta) \to \alpha \to \beta \to \beta)\ \mathsf{id}\ y \Downarrow \mathsf{id}$$

2.3 Type System

Type rules are generally of the form $\Gamma \vdash e :: t$, indicating that a term e is of type t in context Γ. The context lists valid base types,

[1] Type theorists use ':' or '$\in$' to denote the type inhabitation relation. In Haskell, the symbol ':' is used as the "cons" operator for lists, therefore the designers of Haskell chose the non-standard '::' for type annotations. In this paper, we will stick as close as possible to Haskell's syntax.

$\Gamma ::= \varepsilon$		empty context
	$\mid \Gamma, \alpha :: *$	adding a type identifier
	$\mid \Gamma, x :: \tau$	adding a term identifier

$$\frac{}{\mathsf{valid}(\varepsilon)} \qquad \frac{\mathsf{valid}(\Gamma)}{\mathsf{valid}(\Gamma, \alpha :: *)} \qquad \frac{\mathsf{valid}(\Gamma) \quad \Gamma \vdash \tau :: *}{\mathsf{valid}(\Gamma, x :: \tau)}$$

$$\frac{\Gamma(\alpha) = *}{\Gamma \vdash \alpha :: *} \qquad \frac{\Gamma \vdash \tau :: * \quad \Gamma \vdash \tau' :: *}{\Gamma \vdash \tau \to \tau' :: *}$$

Figure 2. Contexts and well-formed types in $\lambda_\to$

$$\frac{\Gamma \vdash \tau :: * \quad \Gamma \vdash e ::_\downarrow \tau}{\Gamma \vdash (e :: \tau) ::_\uparrow \tau} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x ::_\uparrow \tau} \qquad \frac{\Gamma \vdash e_1 ::_\uparrow \tau \to \tau' \quad \Gamma \vdash e_2 ::_\downarrow \tau}{\Gamma \vdash e_1\ e_2 ::_\uparrow \tau'}$$

$$\frac{\Gamma \vdash e ::_\uparrow \tau}{\Gamma \vdash e ::_\downarrow \tau} \qquad \frac{\Gamma, x :: \tau \vdash e ::_\downarrow \tau'}{\Gamma \vdash \lambda x \to e ::_\downarrow \tau \to \tau'}$$

Figure 3. Type rules for $\lambda_\to$

and associates identifiers with type information. We write $\alpha :: *$ to indicate that α is a base type, and $x :: t$ to indicate that x is a term of type t. Every free variable in both terms and types must occur in the context. For instance, if we want to declare const to be of type $(\beta \to \beta) \to \alpha \to \beta \to \beta$, we need our context to contain at least:

$$\alpha :: *, \beta :: *, \mathsf{const} :: (\beta \to \beta) \to \alpha \to \beta \to \beta$$

Note α and β are introduced before they are used in the type of const. These considerations motivate the definitions of contexts and their validity given in Figure 2.

Multiple bindings for the same variable can occur in a context, with the rightmost binding taking precedence. We write $\Gamma(z)$ to denote the information associated with identifier z by context Γ.

The last two rules in Figure 2 explain when a type is well-formed, i.e., when all its free variables appear in the context. In the rules for the well-formedness of types as well as in the type rules that follow, we implicitly assume that all contexts are valid.

Note that $\lambda_\to$ is not polymorphic: a type identifier represents a specific type, and cannot be instantiated.

Finally, we can give the (syntax-directed) type rules (Figure 3). It turns our that for some expressions, we can infer a type, whereas generally, we can only check an expression against a given type. The arrow on the type rule indicates whether the type is an input ($::_\downarrow$) or an output ($::_\uparrow$). For now, this is only a guideline, but the distinction will become more significant in the implementation.

Let us first look at the inferable terms. We check annotated terms against their type annotation, and then return the type. The types of variables are looked up in the environment. For applications, we deal with the function first, which must be of a function type. We can then check the argument against the function's domain, and return the range as the result type.

The final two rules are for type checking. If we want to check an inferable term against a type, then this type must be identical to the one that is inferred for the term. A lambda abstraction can only be checked against a function type. We check the body of the abstraction in an extended context.

Here are type judgements – derivable using the above rules – for our two running examples:

$$\alpha :: *, y :: \alpha \qquad \vdash (\mathsf{id} :: \alpha \to \alpha)\ y :: \alpha$$

$$\alpha :: *, y :: \alpha, \beta :: * \vdash (\mathsf{const} :: (\beta \to \beta) \to \alpha \to \beta \to \beta)\ \mathsf{id}\ y :: \beta \to \beta$$

2.4 Implementation

We now give an implementation of $\lambda_\to$ in Haskell. We provide an evaluator for well-typed expressions, and routines to type-check

$\lambda_{\to}$ terms. The implementation follows the formal description that we have just introduced very closely.

We make use of two simple implementation tricks that help us to focus of the essence of the algorithms.

De Bruijn *indices* In order to save us the work of implementing α-conversion and α-equality – i.e., renaming of variables, preventing name capture, etc. – we use De Bruijn indices: each occurrence of a bound variable is represented by a number indicating how many binders occur between the variable and where it is bound.

Using this notation, we can for example write id as $\lambda \to 0$, and const as $\lambda \to \lambda \to 1$. When using De Bruijn indices, the equality check on types can be implemented as syntactic equality.

While bound variables are represented using natural numbers, we still make use of strings for free variables. The data type of terms has a constructor *Var*, taking an *Int* as argument, for a De Bruijn index, and a constructor *Par* with a *Name* argument, for free variables. Most names are strings, we will introduce the other categories when we need them:

```
data Name
  = Const    String
  | Bound    Int
  | Unquoted Int
  deriving (Show, Eq)
```

Higher-order abstract syntax We make use of the Haskell function space to represent function values. With this representation, we can implement function application using Haskell's own function application, and we do not have to implement β-reduction.

There is a small price to pay, namely that Haskell functions cannot be inspected, i.e., we cannot print them or compare them for equality. We can, however, use our knowledge about the syntax of values to *quote* values and transform the back into types. We will return to the quote function, after we have to defined the evaluator and type checker.

Abstract syntax The type rules in Figure 3 reveal that we can infer the types for annotated terms, variables and application constructs, whereas we can only check the type for lambda abstractions. We therefore make a syntactic distinction between *inferable* ($\mathsf{Term}_{\uparrow}$) and *checkable* ($\mathsf{Term}_{\downarrow}$) terms.

```
data Term↑
  = Ann  Term↓ Type
  | Var  Int
  | Par  Name
  | Term↑ :@: Term↓
  deriving (Show, Eq)

data Term↓
  = Inf  Term↑
  | Lam  Term↓
  deriving (Show, Eq)
```

Annotated terms are represented using *Ann*. As explained above, we use integers to represent bound variables (*Var*), and names for free variables (*Par*). The infix constructor :@: denotes application.

Inferable terms are embedded in the checkable terms via the constructor *Inf*, and lambda abstractions (which do not introduce an explicit variable due to our use of De Bruijn indices) are written using *Lam*.

Types consist only of type identifiers (*TPar*) or function arrows (*Fun*). We reuse the Name data type for type identifiers. In $\lambda_{\to}$, there are no bound names on the type level, so there is no need for a *TVar* constructor.

```
data Type
  = TPar Name
  | Fun  Type Type
  deriving (Show, Eq)
```

```
type Env = [Value]

eval↑ :: Term↑ → Env → Value
eval↑ (Ann e _)  d = eval↓ e d
eval↑ (Par  x)   d = vpar x
eval↑ (Var  i)   d = d !! i
eval↑ (e1 :@: e2) d = vapp (eval↑ e1 d) (eval↓ e2 d)

vapp :: Value → Value → Value
vapp (VLam f)     v = f v
vapp (VNeutral n) v = VNeutral (NApp n v)

eval↓ :: Term↓ → Env → Value
eval↓ (Inf  i) d = eval↑ i d
eval↓ (Lam e) d = VLam (λx → eval↓ e (x : d))
```

Figure 4. Implementation of an evaluator for $\lambda_{\to}$

Values are lambda abstractions (*VLam*) or neutral terms (*VNeutral*).

```
data Value
  = VLam     (Value → Value)
  | VNeutral Neutral
```

As described in the discussion on higher-order abstract syntax, we represent function values as Haskell functions of type Value → Value. For instance, the term *const* – if evaluated – results in the value $VLam\ (\lambda x \to VLam\ (\lambda y \to x))$.

The data type for neutral terms matches the formal abstract syntax exactly. A neutral term is either a variable (*NPar*), or an application of a neutral term to a value (*NApp*).

```
data Neutral
  = NPar Name
  | NApp Neutral Value
```

We introduce a function *vpar* that creates the value corresponding to a free variable:

```
vpar :: Name → Value
vpar n = VNeutral (NPar n)
```

Evaluation The code for evaluation is given in Figure 4. The functions $eval_{\uparrow}$ and $eval_{\downarrow}$ implement the big-step evaluation rules for inferable and checkable terms respectively. Comparing the code to the rules in Figure 1 reveals that the implementation is mostly straightforward.

Substitution is handled by passing around an environment of values. Since bound variables are represented as integers, the environment is just a list of values where the i-th position corresponds to the value of variable i. We add a new element to the environment whenever evaluating underneath a binder, and lookup the correct element (using Haskell's list lookup operator !!) when we encounter a bound variable.

For lambda functions (*Lam*), we introduce a Haskell function and add the bound variable x to the environment while evaluating the body.

Contexts Before we can tackle the implementation of type checking, we have to define contexts. Contexts are implemented as (reversed) lists associating names with either $*$ (*HasKind Star*) or a type (*HasType t*):

```
data Kind = Star
  deriving (Show)

data Info = HasKind Kind | HasType Type
  deriving (Show)

type Context = [(Name, Info)]
```

Extending a context is thus achieved by the list "cons" operation; looking up a name in a context is performed by the Haskell standard list function *lookup*.

```
kind↓ :: Context → Type → Kind → Result ()
kind↓ Γ (TPar x) Star
  = case lookup x Γ of
      Just (HasKind Star) → return ()
      Nothing             → throwError "unknown identifier"
kind↓ Γ (Fun κ κ') Star
  = do kind↓ Γ κ  Star
       kind↓ Γ κ' Star

type↑0 :: Context → Term↑ → Result Type
type↑0 = type↑ 0

type↑ :: Int → Context → Term↑ → Result Type
type↑ i Γ (Ann e τ)
  = do kind↓ Γ τ Star
       type↓ i Γ e τ
       return τ
type↑ i Γ (Par x)
  = case lookup x Γ of
      Just (HasType τ) → return τ
      Nothing          → throwError "unknown identifier"
type↑ i Γ (e1 :@: e2)
  = do σ ← type↑ i Γ e1
       case σ of
         Fun τ τ' → do type↓ i Γ e2 τ
                       return τ'
         _        → throwError "illegal application"
type↓ :: Int → Context → Term↓ → Type → Result ()
type↓ i Γ (Inf e) τ
  = do τ' ← type↑ i Γ e
       unless (τ == τ') (throwError "type mismatch")
type↓ i Γ (Lam e) (Fun τ τ')
  = type↓ (i + 1) ((Bound i, HasType τ) : Γ)
          (subst↓ 0 (Par (Bound i)) e) τ'
type↓ i Γ _ _
  = throwError "type mismatch"
```

Figure 5. Implementation of a type checker for $\lambda_{\to}$

Type checking We now implement the rules in Figure 3. The code is shown in Figure 5. The type checking algorithm can fail, and to do so gracefully, it returns a result in the Result monad. For simplicity, we choose a standard error monad in this presentation:

```
type Result α = Either String α
```

We use the function *throwError* :: String → Result α to report an error.

The function for inferable terms $type_{\uparrow}$ returns a type, whereas the the function for checkable terms $type_{\downarrow}$ takes a type as input and returns ().The well-formedness of types is checked using the function $kind_{\downarrow}$. Each case of the definitions corresponds directly to one of the rules.

The type-checking functions are parameterized by an integer argument indicating the number of binders we have encountered. On the initial call, this argument is 0, therefore we provide $type_{\uparrow 0}$ as a wrapper.

We use this integer to simulate the type rules in the handling of bound variables. In the type rule for lambda abstraction, we add the bound variable to the context while checking the body. We do the same in the implementation. The counter i indicates the number of binders we have passed, so *Bound i* is a fresh name that we can associate with the bound variable. We then add *Bound i* to the context Γ when checking the body. However, because we are turning a bound into a free variable, we have to perform the corresponding substitution on the body. The type checker will never encounter a bound variable; correspondingly the function $type_{\uparrow}$ has no case for *Var*.

```
subst↑ :: Int → Term↑ → Term↑ → Term↑
subst↑ i r (Ann e τ)   = Ann (subst↓ i r e) τ
subst↑ i r (Var j)     = if i == j then r else Var j
subst↑ i r (Par y)     = Par y
subst↑ i r (e1 :@: e2) = subst↑ i r e1 :@: subst↓ i r e2

subst↓ :: Int → Term↑ → Term↓ → Term↓
subst↓ i r (Inf e)     = Inf (subst↑ i r e)
subst↓ i r (Lam e)     = Lam (subst↓ (i + 1) r e)
```

Figure 6. Implementation of substitution for $\lambda_{\to}$

```
quote0 :: Value → Term↓
quote0 = quote 0

quote :: Int → Value → Term↓
quote i (VLam f)     = Lam (quote (i + 1) (f (vpar (Unquoted i))))
quote i (VNeutral n) = Inf (neutralQuote i n)

neutralQuote :: Int → Neutral → Term↑
neutralQuote i (NPar x)   = varpar i x
neutralQuote i (NApp n v) = neutralQuote i n :@: quote i v
```

Figure 7. Quotation in $\lambda_{\to}$

Note that the type equality check that is performed when checking an inferable term is implemented by a straightforward syntactic equality on the data type Type. Our type checker does not perform unification.

The code for substitution is shown in Figure 6, and again comprises a function for checkable ($subst_{\downarrow}$) and one for inferable terms ($subst_{\uparrow}$). The integer argument indicates which variable is to be substituted. The interesting cases are the one for *Var* where we check if the variable encountered is the one to be substituted or not, and the case for *Lam*, where we increase i to reflect that the variable to substitute is referenced by a higher number underneath the binder.

Our implementation of the simply-typed lambda calculus is now almost complete. A small problem that remains is the evaluator returns a Value, and we currently have no way to print elements of type Value.

Quotation As we mentioned earlier, the use of higher-order abstract syntax requires us to define a *quote* function that takes a Value back to a term. As the *VLam* constructor of the Value data type takes a function as argument, we cannot simply derive *Show* and *Eq* as we did for the other types. Therefore, as soon as we want to get back at the internal structure of a value, for instance to display results of evaluation, we need the function *quote*. The code is given in Figure 7.

The function *quote* takes an integer argument that counts the number of binders we have traversed. Initially, *quote* is always called with 0, so we wrap this call in the function $quote_0$.

If the value is a lambda abstraction, we generate a fresh variable *Unquoted i* and apply the Haskell function f to this fresh variable. The value resulting from the function application is then quoted at level $i+1$. We use the constructor *Unquoted* that takes an argument of type Int here to ensure that the newly created names do not clash with other names in the value.

If the value is a neutral term (hence an application of a free variable to other values), the function *neutralQuote* is used to quote the arguments. The *varpar* function checks if the variable occurring

at the head of the application is an *Unquoted* bound variable or a constant:

```
varpar :: Int → Name → Term↑
varpar i (Unquoted k) = Var (i − k − 1)
varpar i x            = Par x
```

Quotation of functions is best understood by example. The value corresponding to the term const is $VLam\ (\lambda x \to VLam\ (\lambda y \to x))$. Applying $quote_0$ yields the following:

```
  quote 0 (VLam (λx → VLam (λy → x)))
= Lam (quote 1 (VLam (λy → vpar (Unquoted 0))))
= Lam (Lam (quote 2 (vpar (Unquoted 0))))
= Lam (Lam (neutralQuote 2 (NPar (Unquoted 0))))
= Lam (Lam (Var 1))
```

When *quote* moves underneath a binder, we introduce a temporary name for the bound variable. To ensure that names invented during quotation do not interfere with any other names, we only use the constructor *Unquoted* during the quotation process. If the bound variable actually occurs in the body of the function, we will sooner or later arrive at those occurrences. We can then generate the correct de Bruijn index by determining the number of binders we have passed between introducing and observing the *Unquoted* variable.

Examples We can now test the implementation on our running examples. We make the following definitions

```
id'    = Lam (Inf (Var 0))
const' = Lam (Lam (Inf (Var 1)))

tpar a = TPar (Const a)
par x  = Inf (Par (Const x))

term1 = Ann id' (Fun (tpar "a") (tpar "a")) :@: par "y"
term2 = Ann const' (Fun (Fun (tpar "b") (tpar "b"))
                        (Fun (tpar "a")
                             (Fun (tpar "b") (tpar "b"))))
        :@: id' :@: par "y"

env1 = [(Const "y", HasType (tpar "a")),
        (Const "a", HasKind Star)]
env2 = [(Const "b", HasKind Star)] ++ env1
```

and then run an interactive session in Hugs or GHCi[2]:

```
⟩ quote0 (eval↑ term1 [])
Inf (Par (Const "y"))

⟩ quote0 (eval↑ term2 [])
Lam (Inf (Var 0))

⟩ type↑0 env1 term1
Right (TPar (Const "a"))

⟩ type↑0 env2 term2
Right (Fun (TPar (Const "b")) (TPar (Const "b")))
```

We have implemented a parser, pretty-printer and a small read-eval-print loop,[3] so that the above interaction can more conveniently take place as:

```
⟩⟩ assume (α :: ∗) (y :: α)

⟩⟩ ((λx → x) :: α → α) y
y :: α

⟩⟩ assume β :: ∗

⟩⟩ ((λx y → x) :: (β → β) → α → β → β) (λx → x) y
λx → x :: β → β
```

[2] Using lhs2TEX [10], one can generate a valid Haskell program from the sources of this paper. The results given here automatically generated when this paper is typeset.

[3] The code is included in the paper sources, but omitted from the typeset version for brevity.

With **assume**, names are introduced and added to the context. For each term, the interpreter performs type checking and evaluation, and shows the final value and the type.

3. Dependent types

In this section, we will modify the type system of the simply-typed lambda calculus into a dependently-typed lambda calculus, called λ_Π. The differences are relatively small; in some cases, introducing dependent types even simplifies our code. We begin by discussing the central ideas motivating the upcoming changes.

Dependent function space In Haskell we can define *polymorphic* functions, such as the identity:

```
id :: ∀α.α → α
id = λx → x
```

By using polymorphism, we can avoid writing the same function on, say, integers and booleans. When such expressions are translated to GHC's core language, the polymorphism does not disappear. Instead, the identity function in the core takes *two* arguments: a type α and a value of type α. Calls to the identity function in the core, must explicitly instantiate the identity function with a *type*:

```
id Bool True :: Bool
id Int 3 :: Int
```

Haskell's polymorphism allows types to abstract over types. Why would you want to do anything different? Consider the following data types:

```
data Vector1 α = Vector1 α
data Vector2 α = Vector2 α α
data Vector3 α = Vector3 α α α
```

Clearly, there is a pattern here. We would like to write down a type with the following kind:

```
∀α :: ∗.∀n :: Nat.Vec α n
```

but we cannot do this in Haskell. The problem is that the *type* Vec abstracts over the *value n*.

The *dependent function space* '∀' generalizes the usual function space '→' by allowing the range to *depend* on the domain. The parametric polymorphism known from Haskell can be seen as a special case of a dependent function, motivating our use of the symbol '∀'.[4] In contrast to polymorphism, the dependent function space can abstract over more than just types. The Vec type above is a valid dependent type.

It is important to note that the dependent function space is a generalization of the usual function space. We can, for instance, type the identity function on vectors as follows:

```
∀α :: ∗.∀n :: Nat.∀v :: Vec α n.Vec α n
```

Note that the type v does not occur in the range: this is simply the non-dependent function space already familiar to Haskell programmers. Rather than introduce unnecessary variables, such as v, we use the ordinary function arrow for the non-dependent case. The identity on vectors then has the following, equivalent, type:

```
∀α :: ∗.∀n :: Nat.Vec α n → Vec α n
```

In Haskell, one can sometimes 'fake' the dependent function space [15], for instance by defining natural numbers on the type level (i.e., by defining data types Zero and Succ). Since the type level numbers are different from the value level natural numbers, one then end up duplicating a lot of concepts on both levels. Furthermore, even though one can lift certain values to the type level in this fashion, additional effort – in the form of advanced type

[4] Type theorists call dependent function types Π-types and would write $\Pi\alpha : *.\Pi n :$ Nat.Vec $\alpha\ n$ instead.

$$\frac{e \Downarrow v}{e :: \tau \Downarrow v} \qquad \frac{}{* \Downarrow *} \qquad \frac{\tau \Downarrow v \quad \tau' \Downarrow v'}{\forall x :: \tau.\tau' \Downarrow \forall x :: v.v'} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e_1 \Downarrow \lambda x \to v_1 \quad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow v_1[x \mapsto v_2]} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow n_1\ v_2} \qquad \frac{e \Downarrow v}{\lambda x \to e \Downarrow \lambda x \to v}$$

Figure 8. Evaluation in λ_Π

class programming – is required to perform any computation on such types. Using dependent types, we can parameterize our types by *values*, and as we will shortly see, the normal evaluation rules apply.

Everything is a term Allowing values to appear freely in types breaks the separation of expressions, types, and kinds we mentioned in the introduction. There is no longer a distinction between these different levels: everything is a term. In Haskell, the symbol '::' relates entities on different levels: In $0 :: \mathsf{Nat}$, the 0 is a value and Nat a type, in $\mathsf{Nat} :: *$, the Nat is a type and $*$ is a kind. Now, $*$, Nat and 0 are all terms. While $0 :: \mathsf{Nat}$ and $\mathsf{Nat} :: *$ still hold, the symbol '::' relates two terms. All these entities now reside on the same syntactic level.

We have now familiarized ourselves with the core ideas of dependently-typed systems. Next, we discuss what we have to change in $\lambda_\to$ in order to accomplish these ideas and arrive at λ_Π.

3.1 Abstract syntax

We no longer have the need for a separate syntactic category of types or kinds, all constructs for all levels are now integrated into the expression language:

$e, \tau, \kappa ::=$	$e :: \tau$	annotated term
	$*$	the type of types
	$\forall x :: \tau.\tau'$	dependent function space
	x	variable
	$e_1\ e_2$	application
	$\lambda x \to e$	lambda abstraction

The modifications compared to the abstract syntax of the simply-typed lambda calculus in Section 2.1 are highlighted.

We now also use the symbols τ and κ to refer to expressions, that is, we use them if the terms denoted play the role of types or kinds, respectively. For instance, the occurrence of τ in an annotated term now refers to another expression.

New constructs are imported from what was formerly in the syntax of types and kinds. The kind $*$ is now an expression. Arrow kinds and arrow types are subsumed by the new construct for the dependent function space. Type variables and term variables now coincide.

3.2 Evaluation

The modified evaluation rules are in Figure 8. All the rules are the same as in the simply-typed case in Figure 1, only the rules for the two new constructs are added. Perhaps surprisingly, evaluation now also extends to types. But this is exactly what we want: The power of dependent types stems exactly from the fact that we can mix values and types, and that we have functions, and thus computations on the type level. However, the new constructs are comparatively uninteresting for computation: the $*$ evaluates to itself; in a dependent function space, we recurse structurally, evaluating the domain and the range. Therefore, we must extend the abstract syntax of values:

$v ::=$	$x\ \overline{v}$	name application
	$*$	the type of types
	$\forall x :: v.v'$	dependent function space
	$\lambda x \to v$	lambda abstraction

$\Gamma ::= \varepsilon$	empty context	$\frac{}{\mathsf{valid}(\varepsilon)}$	$\frac{\mathsf{valid}(\Gamma) \quad \Gamma \vdash \tau ::_\downarrow *}{\mathsf{valid}(\Gamma, x :: \tau)}$
$\mid \Gamma, x :: \tau$	adding a variable		

Figure 9. Contexts in λ_Π

$$\frac{\Gamma \vdash \tau ::_\downarrow * \quad \Gamma \vdash e ::_\downarrow \tau}{\Gamma \vdash (e :: \tau) ::_\uparrow \tau} \qquad \frac{}{\Gamma \vdash * ::_\uparrow *} \qquad \frac{\Gamma \vdash \tau ::_\downarrow * \quad \Gamma, x :: \tau \vdash \tau' ::_\downarrow *}{\Gamma \vdash \forall x :: \tau.\tau' ::_\uparrow *}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x ::_\uparrow \tau} \qquad \frac{\Gamma \vdash e_1 ::_\uparrow \forall x :: \tau.\tau' \quad \Gamma \vdash e_2 ::_\downarrow \tau}{\Gamma \vdash e_1\ e_2 ::_\uparrow \tau'[x \mapsto e_2]}$$

$$\frac{\Gamma \vdash e ::_\uparrow \tau' \quad \tau \Downarrow v \quad \tau' \Downarrow v}{\Gamma \vdash e ::_\downarrow \tau} \qquad \frac{\Gamma, x :: \tau \vdash e ::_\downarrow \tau'}{\Gamma \vdash \lambda x \to e ::_\downarrow \forall x :: \tau.\tau'}$$

Figure 10. Type rules for λ_Π

3.3 Type system

Before we approach the type rules themselves, we must take a look at contexts again. It turns out that because everything is a term now, the syntax of contexts becomes simpler, as do the rules for the validity of contexts (Figure 9, compare with Figure 2).

Contexts now contain only one form of entry, stating the type a variable is assumed to have. The precondition $\Gamma \vdash \tau ::_\downarrow *$ in the validity rule for non-empty contexts no longer refers to a special judgement for the well-formedness of types, but to the type rules we are about to define – we no longer need special well-formedness rules for types. The precondition ensures in particular that τ does not contain unknown variables.

The type rules are given in Figure 10. Again, we have highlighted the differences to Figure 3. We keep the difference between rules for inference ($::_\uparrow$), where the type is an output, and checking ($::_\downarrow$), where the type is an input. The new constructs $*$ and $\forall$ are among the constructs for which we can infer the type. As before for $\lambda_\to$, we assume that all the contexts that occur in the type rules are valid.

The only change for an annotated term is that – similar to what we have already seen for contexts – the kind check for τ no longer refers to the well-formedness rules for types, but is an ordinary type check itself.

The kind $*$ is itself of type $*$. Although there are theoretical objections to this choice [6], we believe that for the purpose of this paper, the simplicity of our implementation outweighs any such objections.

The rule for the dependent function space is somewhat similar to the well-formedness rule for arrow types for $\lambda_\to$ in Figure 2. Both the domain τ and the range τ' of the dependent function are required to be of kind $*$. In contrast to the rule in Figure 2, τ' may refer to x, thus we extend Γ by $x :: \tau$ while checking e'.

Nothing significant changes for variables.

In a function application, the function must now be of a dependent function type $\forall x :: \tau.\tau'$. The difference to an ordinary function type is that τ' can refer to x. In the result type of the application, we therefore substitute the actual argument e_2 for the formal parameter x in τ'.

Checking an inferable term works as before: we first infer a type, then compare the two types for equality. However, types are now terms and can contain computations, so syntactic equality would be far too restrictive: it would be rather unfortunate if $\mathsf{Vec}\ \alpha\ 2$ and $\mathsf{Vec}\ \alpha\ (1 + 1)$ did not denote the same type. As a result, we evaluate the types to normal forms and compare the normal forms syntactically. Most type systems with dependent types

have a rule of the form:

$$\frac{\Gamma \vdash e :: \tau \quad \tau =_\beta \tau'}{\Gamma \vdash e :: \tau'}$$

This rule, referred to as the *conversion rule*, however, is clearly not syntax directed. Our distinction between inferable and checkable terms ensures that the only place where we need to apply the conversion rule, is when a term is explicitly annotated with its type.

The final type rule is for checking a lambda abstraction. The difference here is that the type is a dependent function type. Note that the bound variable x may now not only occur in the body of the function e. The extended context $\Gamma, x :: \tau$ is therefore used both for type checking the function body and kind checking the the range τ'.

To summarize, all the modifications are motivated by the two key concepts we have introduced in the beginning of Section 3: the function space is generalized to the dependent function space; types and kinds are also terms.

3.4 Implementation

The type rules we have given are still syntax-directed and algorithmic, so no major changes to the implementation is required. However, one difference between the implementation and the rules is that during the implementation, we always evaluate types as soon as they have been kind checked. This means that most of the occurrences of types (τ or τ') in the rules are actually values in the implementation.

In the following, we go through all aspects of the implementation, but only look at the definitions that have to be modified.

Abstract syntax The type Name remains unchanged. So does the type $\mathsf{Term}_\downarrow$. We no longer require Type and Kind, but instead add two new constructors to $\mathsf{Term}_\uparrow$ and replace the occurrence of Type in *Ann* with a $\mathsf{Term}_\downarrow$:

```
data Term↑
  = Ann Term↓ Term↓
  | Star
  | Pi Term↓ Term↓
  | Var Int
  | Par Name
  | Term↑ :@: Term↓
deriving (Show, Eq)
```

We also extend the type of values with the new constructs.

```
data Value
  = VLam (Value → Value)
  | VStar
  | VPi Value (Value → Value)
  | VNeutral Neutral
```

As before, we use higher-order abstract syntax for the values, i.e., we encode binding constructs using Haskell functions. With *VPi*, we now have a new binding construct. In the dependent function space, a variable is bound that is visible in the range, but not in the domain. Therefore, the domain is simply a Value, but the range is represented as a function of type Value → Value.

Evaluation To adapt the evaluator, we just add the two new cases for *Star* and *Pi* to the $eval_\uparrow$ function, as shown in Figure 11 (see Figure 4 for the evaluator for $\lambda_\to$). Evaluation of *Star* is trivial. For a *Pi* type, both the domain and the range are evaluated. In the range, where the bound variable x is visible, we have to add it to the environment.

Contexts Contexts map variables to their types. Types are on the term level now. We store types in their evaluated form, and thus define:

```
type Type    = Value
type Context = [(Name, Type)]
```

```
eval↑ Star      d = VStar
eval↑ (Pi τ τ') d = VPi (eval↓ τ d) (λx → eval↓ τ' (x : d))

iSubst i r Star      = Star
iSubst i r (Pi τ τ') = Pi (cSubst i r τ) (cSubst (i + 1) r τ')

quote i VStar = Inf Star
quote i (VPi v f)
  = Inf (Pi (quote i v) (quote (i + 1) (f (vpar (Unquoted i)))))
```

Figure 11. Extending evaluation, substitution and quotation to λ_Π

```
type↑ :: Int → Context → Term↑ → Result Type
type↑ i Γ (Ann e τ)
  = do type↓ i Γ τ VStar
       let v = eval↓ τ []
       type↓ i Γ e v
       return v
type↑ i Γ Star
  = return VStar
type↑ i Γ (Pi τ τ')
  = do type↓ i Γ τ VStar
       let v = eval↓ τ []
       type↓ (i + 1) ((Bound i, v) : Γ)
             (subst↓ 0 (Par (Bound i)) τ') VStar
       return VStar
type↑ i Γ (Par x)
  = case lookup x Γ of
      Just v  → return v
      Nothing → throwError "unknown identifier"
type↑ i Γ (e1 :@: e2)
  = do σ ← type↑ i Γ e1
       case σ of
         VPi v f → do type↓ i Γ e2 v
                      return (f (eval↓ e2 []))
         _       → throwError "illegal application"

type↓ :: Int → Context → Term↓ → Type → Result ()
type↓ i Γ (Inf e) v
  = do v' ← type↑ i Γ e
       unless (quote0 v == quote0 v') (throwError "type mismatch")
type↓ i Γ (Lam e) (VPi v f)
  = type↓ (i + 1) ((Bound i, v) : Γ)
          (subst↓ 0 (Par (Bound i)) e) (f (vpar (Bound i)))
type↓ i Γ _ _
  = throwError "type mismatch"
```

Figure 12. Implementation of a type checker for λ_Π

Type checking Let us go through each of the cases in Figure 12 one by one. The cases for $\lambda_\to$– for comparison – are in Figure 5. For an annotated term, we first check that the annotation is a type of kind $*$, using the type-checking function $type_\downarrow$. We then evaluate the type. The resulting value v is used to check the term e. If that succeeds, the entire expression has type v.

The (evaluated) type of *Star* is *VStar*.

For a dependent function type, we first kind-check the domain τ. Then the domain is evaluated to v. The value is added to the context while kind-checking the range – the idea is similar to the type-checking rules for *Lam* in $\lambda_\to$ and λ_Π.

There are no significant changes in the *Par* case.

In the application case, the type inferred for the function is a Value now. This type must be of the form *VPi v f*, i.e., a dependent function type. In the corresponding type rule in Figure 10, the bound variable x is substituted by e_2 in the result type τ'. In the implementation, f is the function corresponding to τ', and the substitution is performed by applying it to the (evaluated) e_2.

In the case for *Inf*, we have to perform the type equality check. In contrast to the type rules, we already have two values v and v'. To compare the values, we *quote* them and compare the resulting terms syntactically.

In the case for *Lam*, we require a dependent function type of form *VPi v f* now. As in the corresponding case for $\lambda_{\to}$, we add the bound variable (of type v) to the context while checking the body. But we now perform substitution on the function body e (using $subst_{\downarrow}$) and on the result type f (by applying f).

We thus only have to extend the substitution functions, by adding the usual two cases for *Star* and *Pi* as shown in Figure 11. There's nothing to subsitute for *Star*. For *Pi*, we have to increment the counter before substituting in the range because we pass a binder.

Quotation To complete our implementation of λ_{Π}, we only have to extend the quotation function. This operation is more important than for $\lambda_{\to}$, because as we have seen, it is used in the equality check during type checking. Again, we only have to add equations for *VStar* and *VPi*, which are shown in Figure 11.

Quoting *VStar* yields *Star*. Since the dependent function type is a binding construct, quotation for *VPi* works similar to quotation of *VLam*: to quote the range, we increment the counter i, and apply the Haskell function representing the range to *Unquoted i*.

3.5 Where are the dependent types?

We now have adapted our type system and its implementation to dependent types, but unfortunately, we have not yet seen any examples.

Again, we have written a small interpreter around the type checker we have just presented, and we can use it to define and check, for instance, the polymorphic identity function (where the type argument is explicit), as follows:

```
⟩⟩ let id = (λα x → x) :: ∀(α :: ∗).α → α
id :: ∀(x :: ∗) (y :: x).x
⟩⟩ assume (Bool :: ∗) (False :: Bool)

⟩⟩ id Bool
λx → x :: ∀x :: Bool.Bool
⟩⟩ id Bool False
False :: Bool
```

This is more than we can do in the simply-typed setting, but it is by no means spectacular and does not require dependent types. Unfortunately, while we have a framework for dependent types in place, we cannot write any interesting programs as long as we do not add at least a few specific data types to our language.

4. Beyond λ_{Π}

In Haskell, data types are introduced by special data declarations:

```
data Nat = Zero | Succ Nat
```

This introduces a new type Nat, together with two constructors *Zero* and *Succ*. In this section, we investigate how to extend our language with data types, such as natural numbers.

Obviously, we will need to add the type Nat together with its constructors; but how should we define functions, such as addition, that manipulate numbers? In Haskell, we would define a function that pattern matches on its arguments and makes recursive calls to smaller numbers:

$$\frac{}{\mathsf{Nat} \Downarrow \mathsf{Nat}} \quad \frac{}{Zero \Downarrow Zero} \quad \frac{k \Downarrow l}{Succ\ k \Downarrow Succ\ l}$$

$$\frac{pz \Downarrow v}{natElim\ m\ mz\ ms\ Zero \Downarrow v} \quad \frac{ps\ k\ (natElim\ m\ mz\ ms\ k) \Downarrow v}{natElim\ m\ mz\ ms\ (Succ\ k) \Downarrow v}$$

Figure 13. Evaluation of natural numbers

$$\frac{}{\Gamma \vdash \mathsf{Nat} :: *} \quad \frac{}{\Gamma \vdash Zero :: \mathsf{Nat}} \quad \frac{\Gamma \vdash k :: \mathsf{Nat}}{\Gamma \vdash Succ\ k :: \mathsf{Nat}}$$

$$\frac{\begin{array}{c}\Gamma \vdash m :: \mathsf{Nat} \to * \\ \Gamma, m :: \mathsf{Nat} \to * \vdash mz :: m\ Zero \\ \Gamma, m :: \mathsf{Nat} \to * \vdash ms :: \forall k :: \mathsf{Nat}.m\ k \to m\ (Succ\ k) \\ \Gamma \vdash n :: \mathsf{Nat}\end{array}}{\Gamma \vdash natElim\ m\ mz\ ms\ n :: m\ n}$$

Figure 14. Typing rules for natural numbers

```
plus :: Nat → Nat → Nat
plus Zero n      = n
plus (Succ k) n = Succ (plus k n)
```

In our calculus so far, we can neither pattern match nor make recursive calls. How could we hope to define *plus*?

In Haskell, we can define recursive functions on data types using a fold [18]. Rather than introduce pattern matching and recursion, and all the associated problems, we define functions over natural numbers using the corresponding fold. In a dependently type setting, however, we can define a slightly more general version of a fold called the *eliminator*.

The eliminator is a higher-order function, similar to a fold, describing how to write programs over natural numbers. The fold for natural numbers has the following type:

```
foldNat :: ∀α :: ∗.α → (α → α) → Nat → α
```

This much should be familiar. In the context of dependent types, however, we can be more general. There is no need for the type α to be uniform across the constructors for natural numbers: rather than use $\alpha :: *$, we use $m :: \mathsf{Nat} \to *$. This leads us to the following type of *natElim*:

```
natElim :: ∀m :: Nat → ∗.    m Zero
                          → (∀k :: Nat.m k → m (Succ k))
                          → ∀n :: Nat.m n
```

The first argument of the eliminator is the sometimes referred to as the motive [14]; it explains the reason we want to eliminate natural numbers. The second argument corresponds to the base case, where n is *Zero*; the third argument corresponds to the inductive case where n is *Succ k*, for some k. In the inductive case, we must describe how to construct m (*Succ k*) from k and $m\ k$. The result of *natElim* is a function that given any natural number n, will compute a value of type $m\ n$.

In summary, adding natural numbers to our language involves adding three separate elements: the type Nat, the constructors *Zero* and *Succ*, and the eliminator *natElim*.

4.1 Implementing natural numbers

To implement these three components, we extend the abstract syntax and correspondingly add new cases to the evaluation and type checking functions. These new cases do not require any changes to existing code; we choose to focus only on the new code fragments.

Abstract Syntax To implement natural numbers, we extend our abstract syntax as follows:

```
data Term↑ = ...
  | Nat
  | NatElim Term↓ Term↓ Term↓ Term↓
```

```
eval↓ Zero      d = VZero
eval↓ (Succ k) d = VSucc (eval↓ k d)

eval↑ Nat                      d = VNat
eval↑ (NatElim m mz ms n) d
  = let mzVal = eval↓ mz d
        msVal = eval↓ ms d
        rec nVal =
          case nVal of
            VZero      → mzVal
            VSucc k    → msVal `vapp` k `vapp` rec k
            VNeutral n → VNeutral
                           (NNatElim (eval↓ m d) mzVal msVal n)
            _          → error "internal: eval natElim"
    in rec (eval↓ n d)
```

Figure 15. Extending the evaluator natural numbers

```
data Term↓ = ...
  | Zero
  | Succ Term↓
```

We add new constructors corresponding to the type of and eliminator for natural numbers to the $\mathsf{Term}_\uparrow$ data type. The *NatElim* constructor is fully applied: it expects no further arguments.

Similarly, we extend $\mathsf{Term}_\downarrow$ with the constructors for natural numbers. This may seem odd: we will always know the type of *Zero* and *Succ*, so why not add them to $\mathsf{Term}_\uparrow$ instead? For more complicated types, however, such as dependent pairs, it is not always possible to infer the type of the constructor without a type annotation. We choose to add all constructors to $\mathsf{Term}_\downarrow$, as this scheme will work for all data types.

Evaluation We need to rethink our data type for values. Previously, values consisted exclusively of lambda abstractions and 'stuck' applications. Clearly, we will need to extend the data type for values to cope with the new constructors for natural numbers.

```
data Value = ...
  | VNat
  | VZero
  | VSucc Value
```

Introducing the eliminator, however, also complicates evaluation. The eliminator for natural numbers can also be stuck when the number being eliminated does not evaluate to a constructor. Correspondingly, we extend the data type for neutral terms covers this case:

```
data Neutral = ...
  | NNatElim Value Value Value Neutral
```

The implementation of evaluation in Figure 15 closely follows the rules in Figure 13. The eliminator is the only interesting case. Essentially, the eliminator evaluates to the Haskell function with the behaviour you would expect: if the number being eliminated evaluates to *VZero*, we evaluate the base case *mz*; if the number evaluates to *VSucc k*, we apply the step function *ms* to the predecessor *k* and the recursive call to the eliminator; finally, if the number evaluates to a neutral term, the entire expression evaluates to a neutral term. If the value being eliminated is not a natural number or a neutral term, this would have already resulted in a type error. Therefore, the final catch-all case should never be executed.

Typing Figure 16 contains the implementation of the type checker that deals with natural numbers. Checking that *Zero* and *Succ* construct natural numbers is straightforward.

Type checking the eliminator is bit more involved. Remember that the eliminator has the following type:

```
type↓ i Γ Zero VNat      = return ()
type↓ i Γ (Succ k) VNat = type↓ i Γ k VNat

type↑ i Γ Nat                      = return VStar
type↑ i Γ (NatElim m mz ms n) =
  do type↓ i Γ m (VPi VNat (const VStar))
     let mVal = eval↓ m [ ]
     type↓ i Γ mz (mVal `vapp` VZero)
     type↓ i Γ ms
       (VPi VNat (λn →
        VPi (mVal `vapp` n) (λ_ →
        mVal `vapp` VSucc n)))
     type↓ i Γ n VNat
     let nVal = eval↓ n [ ]
     return (mVal `vapp` nVal)
```

Figure 16. Extending the type checker for natural numbers

```
natElim :: ∀m :: Nat → ∗.    m Zero
                          → (∀k :: Nat.m k → m (Succ k))
                          → ∀n :: Nat.m n
```

We begin by type checking and evaluating the motive *m*. Once we have the value of *m*, we type check the two branches. The branch for zero should have type *m Zero*; the branch for successors should have type $\forall k :: \mathsf{Nat}.m\ k \to m\ (Succ\ k)$. Despite the apparent complication resulting from having to hand code complex types, type checking these branches is exactly what would happen when type checking a fold over natural numbers in Haskell. Finally, we check that the *n* we are eliminating is actually a natural number. The return type of the entire expression is the motive, accordingly applied to the number being eliminated.

Other functions To complete the implementation of natural numbers, we must also extend the auxiliary functions for substitution and quotations with new cases. All new code is, however, completely straight-forward, because no new binding constructs are involved.

Addition With all the ingredients in place, we can finally define addition in our interpreter as follows:

```
⟩⟩ let plus = natElim (λ_ → Nat → Nat)
                      (λn → n)
                      (λk rec n → Succ (rec n))
plus :: ∀(x :: Nat) (y :: Nat).Nat
```

We define a function *plus* by eliminating the first argument of the addition. In each case branch, we must define a function of type $\mathsf{Nat} \to \mathsf{Nat}$; we choose our motive correspondingly. In the base case, we must add zero to the argument *n* – we simply return *n*. In the inductive case, we are passed the predecessor *k*, the recursive call *rec* (that corresponds to adding *k*), and the number *n*, to which we must add *Succ k*. We proceed by adding *k* to *n* using *rec*, and wrapping an additional *Succ* around the result. After having defined *plus*, we can evaluate simple additions in our interpreter[5]:

```
⟩⟩ plus 40 2
42 :: Nat
```

Clearly programming with eliminators does not scale very well. We defer the discussion about how this might be fixed to Section 5.

4.2 Implementing vectors

Natural numbers are still not particularly exciting: they are still the kind of data type we can write quite easily in Haskell. As an example of a data type that really makes use of dependent types, we show how to implement vectors.

[5] For convenience, our parser and pretty-printer support literals for natural numbers. For instance, 2 is translated to *Succ* (*Succ Zero*)) :: Nat on the fly.

```
eval↑ (VecElim α m mn mc n xs) d =
  let mnVal = eval↓ mn d
      mcVal = eval↓ mc d
      rec nVal xsVal =
        case xsVal of
          VNil _          → mnVal
          VCons _ k x xs  → foldl vapp mcVal [k, x, xs, rec k xs]
          VNeutral n      → VNeutral
                              (NVecElim (eval↓ α d) (eval↓ m d)
                                        mnVal mcVal nVal n)
          _               → error "internal: eval vecElim"
  in rec (eval↓ n d) (eval↓ xs d)
```

Figure 17. Implementation of the evaluation of vectors

As was the case for natural numbers, we need to define three separate components: the type of vectors, its the constructors, and the eliminator. We have already mentioned that vectors are parameterized by both a type and a natural number:

$$\forall \alpha :: * . \forall n :: \mathsf{Nat} . \mathsf{Vec}\ \alpha\ n :: *$$

The constructors for vectors are analogous to those for Haskell lists. The only difference is that their types record the length of the vector:

$$\begin{array}{l} \mathit{Nil} \quad :: \forall \alpha :: * . \mathsf{Vec}\ \alpha\ \mathit{Zero} \\ \mathit{Cons} :: \forall \alpha :: * . \forall n :: \mathsf{Nat} . \alpha \to \mathsf{Vec}\ \alpha\ n \to \mathsf{Vec}\ \alpha\ (\mathit{Succ}\ n) \end{array}$$

The eliminator for vectors behaves essentially the same as *foldr* on lists, but its type is a great deal more specific (and thus, more involved):

$$\begin{array}{l} \mathit{vecElim} :: \forall \alpha :: * . \forall m :: (\forall n :: \mathsf{Nat} . \mathsf{Vec}\ \alpha\ n \to *) . \\ \qquad\qquad m\ \mathit{Zero}\ (\mathit{Nil}\ \alpha) \\ \qquad\qquad \to (\forall n :: \mathsf{Nat} . \forall x :: \alpha . \forall xs :: \mathsf{Vec}\ \alpha\ n . \\ \qquad\qquad\quad m\ n\ xs \to m\ (\mathit{Succ}\ n)\ (\mathit{Cons}\ \alpha\ n\ x\ xs)) \\ \qquad\qquad \to \forall n :: \mathsf{Nat} . \forall xs :: \mathsf{Vec}\ \alpha\ n . m\ n\ xs \end{array}$$

The whole eliminator is quantified over the element type α of the vectors. The next argument of the eliminator is the motive. As was the case for natural numbers, the motive is a type (kind $*$) parameterized by a vector. As vectors are themselves parameterized by their length, the motive expects an additional argument of type Nat. The following two arguments are the cases for the two constructors of Vec. The constructor *Nil* is for empty vectors, so the corresponding argument is of type $m\ \mathit{Zero}\ (\mathit{Nil}\ \alpha)$. The case for *Cons* takes a number n, a element x of type α, a vector xs of length n, and the result of the recursive application of the eliminator of type $m\ n\ xs$. It combines those elements to form the required type, for the vector of length $\mathit{Succ}\ n$ where x has been added to xs. The final result is a function that eliminates a vector of any length.

The type of the eliminator may look rather complicated. However, if we compare with the type of *foldr* on lists

$$\mathit{foldr} :: \forall \alpha :: * . \forall m :: * . m \to (\alpha \to m \to m) \to [\alpha] \to m$$

we see that the structure is the same, and the additional complexity stems only from the fact that the motive is parameterized by a vector, and vectors are in turn parameterized by natural numbers.

Not all of the arguments of *vecElim* are actually required – some of the arguments can be inferred from others, to reduce the noise and make writing programs more feasible. We would like to remind you that λ_Π is designed to be a very explicit, low-level language.

Abstract syntax As was the case for natural numbers, we extend the abstract syntax. We add the type of vectors and its eliminator to $\mathsf{Term}_\uparrow$; we extend $\mathsf{Term}_\downarrow$ with the constructors *Nil* and *Cons*.

```
data Term↑ = ...
  | Vec Term↓ Term↓
  | VecElim Term↓ Term↓ Term↓ Term↓ Term↓ Term↓
```

```
type↓ i Γ (Nil α) (VVec bVal VZero) =
  do type↓ i Γ α VStar
     let aVal = eval↓ α [ ]
     unless (quote0 aVal == quote0 bVal)
            (throwError "type mismatch")
type↓ i Γ (Cons α n x xs) (VVec bVal (VSucc k)) =
  do type↓ i Γ α VStar
     let aVal = eval↓ α [ ]
     unless (quote0 aVal == quote0 bVal)
            (throwError "type mismatch")
     type↓ i Γ n VNat
     let nVal = eval↓ n [ ]
     unless (quote0 nVal == quote0 k)
            (throwError "number mismatch")
     type↓ i Γ x aVal
     type↓ i Γ xs (VVec bVal k)

type↑ i Γ (Vec α n) =
  do type↓ i Γ α VStar
     type↓ i Γ n VNat
     return VStar
type↑ i Γ (VecElim α m mn mc n vs) =
  do type↓ i Γ α VStar
     let aVal = eval↓ α [ ]
     type↓ i Γ m
       (VPi VNat (λn →
        VPi (VVec aVal n) (λ_ →
        VStar)))
     let mVal = eval↓ m [ ]
     type↓ i Γ mn (foldl vapp mVal [VZero, VNil aVal])
     type↓ i Γ mc
       (VPi VNat (λn →
        VPi aVal (λy →
        VPi (VVec aVal n) (λys →
        VPi (foldl vapp mVal [n, ys]) (λ_ →
        (foldl vapp mVal [VSucc n, VCons aVal n y ys]))))))
     type↓ i Γ n VNat
     let nVal = eval↓ n [ ]
     type↓ i Γ vs (VVec aVal nVal)
     let vsVal = eval↓ vs [ ]
     return (foldl vapp mVal [nVal, vsVal])
```

Figure 18. Extending the type checker for vectors

```
data Term↓ = ...
  | Nil Term↓
  | Cons Term↓ Term↓ Term↓ Term↓
```

Note that also *Nil* takes an argument, because both constructors are polymorphic in the element type. Correspondingly, we extend the data types for values and neutral terms:

```
data Value = ...
  | VNil Value
  | VCons Value Value Value Value
  | VVec Value Value

data Neutral = ...
  | NVecElim Value Value Value Value Value Neutral
```

Evaluation Evaluation of constructors or the Vec type proceeds structurally, turning terms into their value counterparts. Once again, the only interesting case is the evaluation of the eliminator for vectors, shown in Figure 17. As indicated before, the behaviour resembles a fold on lists: depending on whether the vector is a *VNil* or a *VCons*, we apply the appropriate argument. In the case

for *VCons*, we also call the eliminator recursively on the tail of the vector (of length k). If the eliminated vector is a neutral element, we cannot reduce the eliminator, and produce a neutral term again.

Type checking We extend the type checker as shown in Figure 18. The code is relatively long, but keeping the types of each of the constructs in mind, there are absolutely no surprises.

As for natural numbers, we have omitted the new cases for substitution and quotation, because they are entirely straight-forward.

Append We are now capable of demonstrating a real dependently typed program in action, a function that appends two vectors while keeping track of their lengths. The definition in the interpreter looks as follows:

```
⟩ let append =
    (λα → vecElim α
            (λm _ → ∀(n :: Nat).Vec α n → Vec α (plus m n))
            (λ_ v → v)
            (λm v vs rec n w → Cons α (plus m n) v (rec n w)))
  :: ∀(α :: ∗) (m :: Nat) (v :: Vec α m) (n :: Nat) (w :: Vec α n).
     Vec α (plus m n)
```

Like for *plus*, we define a binary function on vectors by eliminating the first argument. The motive is chosen to expect a second vector. The length of the resulting vector is the sum of the lengths of the argument vectors *plus m n*. Appending an empty vector to another vector *v* results in *v*. Appending a vector of the form *Cons m v vs* to a vector *v* works by invoking recursion via *rec* (which appends *vs* to *w*) and prepending *v*. Of course, we can also apply the function thus defined:

```
⟩ assume (α :: ∗) (x :: α) (y :: α)
⟩ append α 2 (Cons α 1 x (Cons α 0 x (Nil α)))
             1 (Cons α 0 y (Nil α))
Cons α 2 x (Cons α 1 x (Cons α 0 y (Nil α))) :: Vec α 3
```

We assume a type α with two elements x and y, and append a vector containing two x's to a vector containing one y.

4.3 Summary

In this section, we have demonstrated how to add two data types to the λ_Π: natural numbers and vectors. Using exactly the same principles, many more data types can be added. For example, for any natural number n, we can define the type Fin n that contains exactly n elements. In particular, Fin 0, Fin 1 and Fin 2 are the empty type, the unit type, and the type of booleans respectively. Furthermore, Fin can be used to define a total projection function from vectors, of type *project* :: $\forall(\alpha :: *)$ $(n ::$ Nat$).$*Vec* α $n \rightarrow$ Fin $n \rightarrow \alpha$.

Another interesting dependent type is the equality type *Eq* :: $\forall(\alpha :: *).\alpha \rightarrow \alpha \rightarrow *$ with a single constructor *Refl* :: $\forall(\alpha :: *)$ $(x :: \alpha) \rightarrow$ *Eq* α x x, Using *Eq*, we can state and prove theorems about our code directly in λ_Π. For instance, the type $\forall(\alpha :: *)$ $(n ::$ Nat$).$*Eq* Nat (*plus n Zero*) n states that *Zero* is the right-neutral element of addition. Any term of that type serves as a proof of that theorem, via the Curry-Howard isomorphism.

A few of these examples are included with the interpreter in the paper sources, which can be downloaded via the λ_Π homepage [11]. More about suitable data types for dependently typed languages and writing dependently-typed programs can be found in another tutorial [16].

5. Toward dependently-typed programming

The calculus we have described is far from a real programming language. Although we can write, type check, and evaluate simple expressions there is still a lot of work to be done before it becomes feasible to write large, complex programs. Most of the remaining work is, in fact, still subject to exciting new research. There is really too much to cover, but we attempt to identify several important points.

Program development As it stands, the core system we have presented requires programmers to explicitly instantiate polymorphic functions. This is terribly tedious! Take the *append* function we defined: of its five arguments, only two are interesting. Fortunately, uninteresting arguments can usually be inferred. Many programming languages and proof assistants based on dependent types have support for *implicit arguments* that the user can omit when calling a function. Note that these arguments need not be types: the *append* function is 'polymorphic' in the length of the vectors.

Writing programs with complex types in one go is not easy. Epigram [9] and Agda [20] allow programmers to put 'holes' in their code, leaving parts of their programs undefined. Programmers can then ask the system what type a specific hole has, effectively allowing the incremental development of complex programs.

Our system cannot do any type inference. The distinction between $\mathsf{Term}_\uparrow$s and $\mathsf{Term}_\downarrow$s may minimize the number of annotations that must be present, but types are never inferred. This may seem quite a step back from Haskell. On the contrary, once you provide such detailed type information parts of *your program* can actually be inferred. This is demonstrated by Epigram. In Epigram all high-level programs are 'compiled' down to a core type theory, similar to the one we have defined hered. When you perform case analysis on a non-empty vector, however, you do not need to write the case for *Nil*. In reality, a clever choice of motive and automation explains to the eliminator *why* that branch is impossible. While we may need to be more explicit about our types, the type information can help guide our program development.

Totality As our implementation illustrates, typing checking a dependently typed programming language involves evaluating functions statically. For this reason, it is important to know when a function is *total*, i.e., when a function is guaranteed to produce a result for every in input in finite time. If we guarantee that only total functions may be evaluated during type checking, type checking will remain decidable.

On the other hand, most functional programmers write partial functions all the time. For example, the *head* function only works on non-empty lists. Similarly, functions may use general recursion and diverge on malformed input. In a sense the situation is akin to Haskell, before the introduction of monads to encapsulate IO. It is clear that dependently typed programming languages must come to terms with partial functions. Finding the right solution, however, is still very much an open problem.

Cayenne [1], for instance, allowed programmers to freely define functions using general recursion. As a result, Cayenne's type checker could loop. Similar problems arise when we allow Haskell's case construct: the type checker could crash with an exception equivalent to `Prelude.head: empty list`. In this paper we taken a more prudent approach and only allow recursion and pattern matching via eliminators, which are guaranteed to produce total functions.

As our examples illustrate, however, programming with eliminators does not scale. Epigram uses a clever choice of motive to make programming with eliminators a great deal more practical [13, 17]. By choosing the right motive, we can exploit type information when defining complicated functions. Eliminators may not appear to be terribly useful, but they form the foundations on which dependently typed programming languages may be built.

There may be situations, however, where programming with eliminators is just not enough. One possible solution is to permit general recursion, encapsulated in a suitable monad. Just as the IO monad encapsulates impure functions in Haskell, monads can

also be used to introduce partial functions into a total language [5]. Finding the right way to tackle partiality, without sacrificing theoretical properties of the type system, requires both more research and practical experience.

Real world programming Although we have added several data types to the core theory, we cannot expect programmers to implement new data types by hacking them into the compiler. We need to add support for user-defined data types. Once again, we must be careful about *which* data types we allow. For example, general recursion can be introduced via data types with negative occurrences, such as our Value type.

Recently, generalized algebraic data types have caused quite some excitement in the Haskell community. In the presence of dependent types, GADTs (or *indexed families* as they are known in the type theory community) become even more expressive. Vectors already illustrate that indexing by a value, instead of a type, allows you to be more precise about the structure of data and the invariants it satisfies. This pattern pops up everywhere: well-typed lambda terms; proof-carrying code; red-black trees; the list goes on and on.

There has been very little research on how to compile dependently typed languages. As a result, many people believe dependent types are inherently inefficient: naively compiling *append* would result in code that computes the length of the resulting vector, even if this information is not used anywhere. Such computations, however, are only relevant during type checking. This illustrates how there is still a distinction between evaluation for the sake of type checking and evaluation to compute the result of your program. Edwin Brady covers this, together with various optimizations only possible due to the presence of richer type information, in his thesis [3, 4].

6. Discussion

There is a large amount of relevant literature regarding both implementing type systems and type theory. Pierce's book [21] is an excellent place to start. Martin-Löf's notes on type theory [12] are still highly relevant and form an excellent introduction to the subject. More recent books by Nordström et al. [19] and Thompson [22] are freely available online.

There are several dependently typed programming languages and proof assistants readily available. Coq [2] is a mature, well-documented proof assistant. While it is not primarily designed for dependently typed *programming*, learning Coq can help get a feel for type theory. Haskell programmers may feel more at home using recent versions of Agda [20], a dependently typed programming language. Not only does the syntax resemble Haskell, but functions may be defined using pattern matching and general recursion. Finally, Epigram [9, 16] proposes a more radical break from functional programming as we know it. While the initial implementation is far from perfect, many of Epigram's ideas are not yet implemented elsewhere.

Other implementations of the type system we have presented here have been published elsewhere [7, 8]. These implementations are given in pseudocode and accompanied by a proof of correctness. The focus of our paper is somewhat different: we have chosen to describe a concrete implementation as a vehicle for explanation.

In the introduction we mentioned some of the concerns Haskell programmers have regarding dependent types. The type checking algorithm we have presented here is decidable and will always terminate. The phase distinction between evaluation and type checking becomes more subtle, but is not lost. The fusion of types and terms introduces new challenges, but also has a lot to offer. Most importantly, though, getting started with dependent types is not as hard as you may think. We hope to have whet your appetite, guiding you through your first steps, but encourage you to start exploring dependent types yourself!

References

[1] Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[3] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[4] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, volume 3085 of *LNCS*. Springer-Verlag, 2004.

[5] Venanzio Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[6] T. Coquand. An analysis of Girard's paradox. In *First IEEE Symposium on Logic in Computer Science*, 1986.

[7] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[8] Thierry Coquand and Makoto Takeyama. An implementation of Type: Type. In *International Workshop on Types for Proofs and Programs*, 2000.

[9] Conor McBride et al. Epigram, 2004. `http://www.e-pig.org`.

[10] Ralf Hinze and Andres Löh. lhs2TEX, 2007. `http://www.iai.uni-bonn.de/~loeh/lhs2tex`.

[11] λ_Π homepage, 2007. `http://www.iai.uni-bonn.de/~loeh/LambdaPi`.

[12] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[13] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[14] Conor McBride. Elimination with a motive. In *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 197–216. Springer-Verlag, 2000.

[15] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

[16] Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170, 2004.

[17] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th Conf. on Functional Programming Languages and Computer Architecture*, 1991.

[19] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon, 1990.

[20] Ulf Norell. Agda 2. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

[21] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[22] Simon Thompson. *Type theory and functional programming*. Addison Wesley Longman Publishing Co., Inc., 1991.