Implementing a Dependently Typed Lambda Calculus

Wouter Swierstra

25-07-07

Goal

Explain dependently typed lambda calculi by presenting a simple, elegant implementation in Haskell.

So what are dependent types?

Google

[Dependent type - Wikipedia, the free encyclopedia](#)
In computer science and logic, a **dependent type** is a **type** which depends on a value. **Dependent types** play a central role in Intuitionistic **Type** Theory and in **...**
en.wikipedia.org/wiki/**Dependent**_type - 19k - Cached - Similar pages

[PDF] **Why Dependent Types Matter**
File Format: PDF/Adobe Acrobat - View as HTML
Dependent types reduce certification to **type** checking, hence they provide **...** In section 5 we how to use **dependent types** to maintain static invariants about **...**
www.cs.nott.ac.uk/~txa/publ/ydtm.pdf - Similar pages

Why **Dependent Types** Matter | Lambda the Ultimate
We discuss the relationship to other proposals to introduce aspects of **dependent types** into functional programming languages and sketch some topics for **...**
lambda-the-ultimate.org/node/653 - 33k - Cached - Similar pages

Lightweight **Dependent-type** Programming
Several techniques to emulate and benefit from **dependent** typing in existing languages such as Haskell or ML.
okmij.org/ftp/Computation/ lightweight-**dependent**-typing.html - 21k - Cached - Similar pages

Dependent ML: DML
Conservative ML extension, has **type** system to enrich ML with restricted form of **dependent types**, to allow many interesting program properties: memory safety **...**
www.cs.bu.edu/~hwxi/DML/DML.html - 8k - Cached - Similar pages

Dependent Types in Practical Programming - Xi, Pfenning ...
Programming is a notoriously error prone process, and a great deal of evidence in practice has demonstrated that the use of a **type** system in a programming **...**
citeseer.ist.psu.edu/xi98**dependent**.html - 22k - Cached - Similar pages

Dependent Types: Resources and Errata
Dependent Types: Resources and Errata. David Aspinall and Martin Hofmann. The OCaml implementation deptypes.tgz (coming soon); A list of errata for the **...**
homepages.inf.ed.ac.uk/da/attapl/ - 1k - Cached - Similar pages

Do we Need **Dependent Types**?
Do we Need **Dependent Types**? **...** within the Hindley-Milner **type** system, some functions which seem to require a language with **dependent types**

Wikipedia

Contents [hide]

Systems of The Lambda Cube [edit]

Pure first order dependent types [edit]

The system λP of pure first order dependent types, corresponding to the logical framework LF, is obtained by generalising the function space type of the simply typed lambda calculus to the dependent product type.

Writing $\mathrm{Vec}(\mathbb{R}, n)$ for n-tuples of real numbers, as above, $\Pi n : \mathbb{N}.\mathrm{Vec}(\mathbb{R}, n)$ stands for the type of functions which given a natural number n returns a tuple of real numbers of size n. The usual function space arises as a special case when the range type does not actually depend on the input, e.g. $\Pi n : \mathbb{N}.\mathbb{R}$ is the type of functions from natural numbers to the real numbers, written as $\mathbb{N} \longrightarrow \mathbb{R}$ in the simply typed lambda calculus.

See also: [edit]

- Lambda cube
- Typed lambda calculus
- Intuitionistic type theory

Languages with dependent types [edit]

- C++
- Epigram
- Dependent ML

Let's start with the simply typed lambda calculus

Simply typed lambda calculus

$$M ::= x \mid (MM) \mid \lambda x.M$$

- Variables
- Application
- Lambda abstraction

Type rules

Simply typed lambda calculus

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau}$$

Type rules

Dependently typed lambda calculus

$$\frac{\Gamma, x : \sigma \vdash t : \tau[x]}{\Gamma \vdash \lambda x.t : (x : \sigma) \to \tau[x]}$$

$$\frac{\Gamma \vdash t_1 : (x : \sigma) \to \tau[x] \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau[t_2]}$$

Why should I care about dependent types?

Polymorphism

- Polymorphism allows abstraction over **types**:

```
id :: forall a, a -> a
id x = x
```

Dependent types facilitate polymorphism:

```
id :: (a :: *) -> a -> a
id _ x = x
```

...but also enable abstraction over **data**.

GADTs

```
data Z = Z
data S k = S k

data Vec n a where
  Nil  :: Vec Z a
  Cons :: a -> Vec a k -> Vec a (S k)

vhead :: Vec (S k) a -> a
```

What about append?

GADTs

- This pattern is very, very common.
  - Red black trees
  - Well-scoped lambda terms
  - Parsers and lexers

Precise Programming

Curry-Howard Isomorphism

- Dependent types provide a mathematical framework for doing proofs.
- At the heart of proof assistants like Coq.
- You can program and prove properties of your programs in the same system.

```
Lemma revLemma
  (a : *) (xs : list a) :
  reverse (reverse xs) = xs.
```

Why should I care about dependent types?

More abstraction.

When are two types ‘the same’?

- Syntactic equality
- Unifiable
- What about these two types?
  - `Vec 4 Int`
  - `Vec (2+2) Int`

The conversion rule

$$\frac{\Gamma \vdash t : \sigma \quad \sigma \simeq_\beta \tau}{\Gamma \vdash t : \tau}$$

Type checking needs to perform evaluation!

What now?

- Implement the simply typed lambda calculus.
- Modify our implementation to deal with dependent types.
- Add data types to our mini language.

Implementing the simply typed lambda calculus

- Terms and values
- Types
- Substitution
- Evaluation
- Type checking

Term – examples

$\lambda x.x$	the identity function
$\lambda y.\lambda x.y$	constant functions
$\lambda f.\lambda x.fx$	application

Terms – specification

$$M ::= x \mid (MM) \mid \lambda x.M$$

Sticky implementation details

- How do we treat variables?
- Are bound variables the same as free variables?
- If we do type checking, where should we have type annotations?

De Bruijn indices

- We use **de Bruijn indices** to represent variables.
- "The variable *k* is bound *k* lambdas up"
- Examples:

$$\lambda x.x \qquad \lambda.0$$

$$\lambda x.\lambda y.x \qquad \lambda.\lambda.1$$

Type annotations

- We distinguish between **checkable** and **inferable** terms.
- The checkable terms need a type annotation to type check.
- The inferable terms require no such annotation.

Terms

```
data InferTerm
 = Check CheckTerm Type -- annotation
 | Var Int              -- bound variables
 | Par Int              -- parameters
 | App InferTerm CheckTerm
                        -- application

data CheckTerm
 = Infer InferTerm
 | Lam CheckTerm -- lambda abstraction
```

Values – specification

- We want to evaluate lambda terms to their normal form.
- A **value** is a fully evaluated lambda expression.

$$v ::= \lambda x.v \mid x\, v$$

Examples: $\lambda x.x$

$x(\lambda y.yz)$

Values – implementation

```
data Value
  = VApp Int [Value]
  | VLam (Value -> Value)
```

Types – implementation

```
data Type
  = TPar Int            -- sigma, tau, etc.
  | Fun Type Type       -- sigma -> tau
```

Evaluation – examples

Evaluation turns a **term** into a **value**.

$$(\lambda x.x)z \ \Downarrow \ z$$

$$(\lambda x.\lambda y.x)(\lambda z.z) \ \Downarrow \ \lambda y.\lambda z.z$$

Evaluation – specification

$$x \Downarrow x \qquad \frac{e \Downarrow v}{\lambda x.e \Downarrow \lambda x.v}$$

$$\frac{e_1 \Downarrow \lambda x.v_1 \quad e_2 \Downarrow v_2}{e_1\, e_2 \Downarrow v_1[x \mapsto v_2]}$$

Evaluation – implementation

- To implement evaluation we need to write a substitution function.
- We will keep track of an **environment** containing a list of values for the variables that we have encountered so far.

Substitution is easy

```
subst i t (Par y) = Par y
subst i t (Var j)
  | i == j    = t
  | otherwise = Var j
```

All the other cases follow the structure of terms.

Evaluation – implementation

```
type Env = [Value]

evalInfer :: InferTerm -> Env -> Value
evalInfer (Par x) env = VApp x []
evalInfer (Var i) env = env !! i
evalInfer (App f x) env =
 app (evalInfer f env) (evalInfer x env)

app :: Value -> Value -> Value
app (VLam f) x = f x
app (VApp x vs) v = VApp x (vs ++ [v])
```

Evaluation (continued)

```
evalCheck :: CheckTerm -> Env -> Value
evalCheck (Lam f) env =
  VLam (\v -> eval f (v : env))
```

Type checking – examples

$$\lambda x : \sigma . x : \sigma \rightarrow \sigma$$

$$\lambda x : \sigma \lambda y : \tau . x : \sigma \rightarrow \tau \rightarrow \sigma$$

Type checking - specification

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash f x : \tau}$$

Type checking

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

inferType :: Int -> Context
  -> InferTerm -> Maybe Type

checkType :: Int -> Context -> Type
  -> CheckTerm -> Maybe ()
```

A few interesting cases

```
inferType i g (Par x) = lookup x g

inferType i g (App f x) = do
  Fun d r <- inferType i g f
  checkType i g d x
  return r

checkType i g (Fun d r) (Lam t) = do
  checkType (i + 1) ((i,d):g) r
     (subst 0 (Par i) t)
```

Things to notice

- When type checking a lambda term, we assume that we have a function type.
- There is no case for bound variables, when we go under a lambda the bound variable is "freed".

What about dependent types?

Implementing dependent types

- Terms and values
  - No separate data type for our types!
- Substitution
- Evaluation
- Type checking
  - Avoid conversion check by fully evaluating types.

Abstract syntax – examples

$$\lambda a.\lambda x.x \ : \ (a : \star) \rightarrow a \rightarrow a$$

$$(a : *) \rightarrow a \rightarrow a \ : \ *$$

Aside: there are some theoretical problems with the system I present here.

Abstract syntax – specification

$$
\begin{array}{rcl}
e, \tau, \sigma & ::= & x \\
& | & e_1\, e_2 \\
& | & \lambda x.e \\
& | & * \\
& | & (x : \sigma) \to \tau
\end{array}
$$

Terms

```
data InferTerm
  = -- Check, Var, Par, App and
  | Star -- the type of all types
  | Pi CheckTerm CheckTerm
    -- dependent function space
    -- (x : sigma) -> tau[x]
```

Evaluation - new spec

$$* \Downarrow *$$

$$\frac{\tau \Downarrow v \quad \tau' \Downarrow v'}{(x : \tau) \to \tau' \Downarrow (x : v) \to v'}$$

Values – implementation

```
data Value
  = VApp Name [Value]  -- x (\y -> y)
  | VLam (Value -> Value) -- (\y -> y)
  | VStar -- just like the Star term
  | VPi Value (Value -> Value)
```

Substitution is still easy.

Evaluation - what's new

```
type Env = [Value]

evalInfer Star env = VStar
evalInfer (Pi d r) env =
  VPi (evalInfer d env)
      (\v -> evalInfer r (v:env))
```

The rest is unchanged.

Type checking – new specification

$$\frac{}{\Gamma \vdash * : *}$$

$$\frac{\Gamma \vdash \sigma : * \quad \Gamma, x : \sigma \vdash \tau : *}{\Gamma \vdash (x : \sigma) \to \tau : *}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \simeq_\beta \tau}{\Gamma \vdash e : \tau}$$

Conversion rule

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \simeq_\beta \tau}{\Gamma \vdash e : \tau}$$

- Note that the conversion rule is not **syntax directed**.
- We ensure all our **types** are always **fully evaluated**.
- Conversion then boils down to checking if two values (types) coincide.

Type inference

```
inferType :: Int -> Context -> InferTerm
  -> Maybe Value

inferType i g (Star) = return VStar

inferType i g (Pi d r) = do
  checkType i g VStar d
  let dVal = eval d []
  checkType (i+1) ((i,dVal) : g) VStar
    (subst 0 (Par i) r)
  return VStar
```

Type checking

```
checkType :: Int -> Context -> Value
  -> CheckTerm -> Maybe ()

checkType i g (VPi d r) (Lam t) =
  checkType (i+1) ((i,d):g)
   (r (VApp i [])) (subst 0 (Par i) t)
```

Dependent types

```
inferType i g (App f x) = do
  VPi d r <- inferType i g f
  checkType i g d x
  return (r (eval x []))
```

Quoting

- To actually display and compare values, we need a function:

```
quote :: Value -> CheckTerm
```

- Idea: fully apply a value to fresh variables.

And now to write a programming language...

- This calculus is not much more useful than the simply typed lambda calculus.
- We need to add data types, pattern matching, and recursion.

Adding natural numbers

- To do any "real programming" with dependent types, we need to add **data types**.
- I'll introduce natural numbers to the language – most other types can be implemented analogously.

Natural numbers in Haskell

```
data Nat = Zero | Succ Nat

plus :: Nat -> Nat -> Nat
plus Zero n = n
plus (S k) n = S (plus k n)
```

We need to add a new type and the constructors.

Eliminators

- How should we write functions, such as plus, using pattern matching and recursion?
- We write functions over natural numbers using the **eliminator**, a higher order function similar to a fold.

Folding over natural numbers

- In Haskell, we could write the fold over natural numbers as:

```
foldNat ::
 forall a .     -- target type
 a ->           -- the Zero case
 (a -> a) ->    -- the Succ case
 Nat ->
 a
```

Dependent eliminators

- Using dependent types we can be more general.
- We distinguish **in the type** between the cases for successor and zero.

Eliminator for natural numbers

```
natElim :
(m : (n : Nat) -> *) -- motive
 m Zero ->    -- the Zero case
 ((k:Nat) -> -- predecessor
   m k ->     -- ind. hypothesis
   m (Succ k)) -- ind. step
 (n : Nat) ->
 m n
```

Using the eliminator

- We can use the eliminator to write functions, like plus:

```
plus m n =
  natElim (\m -> Nat)
    n
    (\pred rec -> S rec)
    m
```

Implementation – overview

- To implement natural numbers we need to:
  - add a new **type**, new **constructors**, and the **eliminator** to the abstract syntax
  - add new values, corresponding to the new normal forms
  - extend our functions for type checking and substitution.

More reading

- The details of everything I discussed (and more!) is in the paper:

Simply Easy! (An Implementation of a Dependently Typed Lambda Calculus), *Andres Loeh, Conor McBride, Wouter Swierstra*

More hacking

The code (together with a small interpreter) is available online:

www.informatik.uni-bonn.de/~loeh/LambdaPi.html