Implementing a Dependently Typed Lambda Calculus

Wouter Swierstra

12-04-07

Who am I?

Haskell-cafe

- Wouldn’t it be nice to have dependent types?
- GADTs are darn near dependent types.
- Dependent types force you to write correct code.
- No dependent types are necessary.

[**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**

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 $\mathbb{N} \to \mathbb{R}$ 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

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!

Misconceptions

- Dependent type checking:
  - is undecidable.
  - will cause your type checker to loop.
  - is really, really hard.

Implementing the simply typed lambda calculus

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

Terms

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

data CheckTerm
 = Infer InferTerm
 | Lam CheckTerm
```

Values and types

```
-- with HOAS...
data Value
  = VApp Int [Value]  -- x (\y -> y) z
  | VLam (Value -> Value) -- (\y -> y)

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

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 - I

```
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 - II

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

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)
```

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.

Terms

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

Values

```
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 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.
- Then we need to add **tactics** that do the menial jobs.