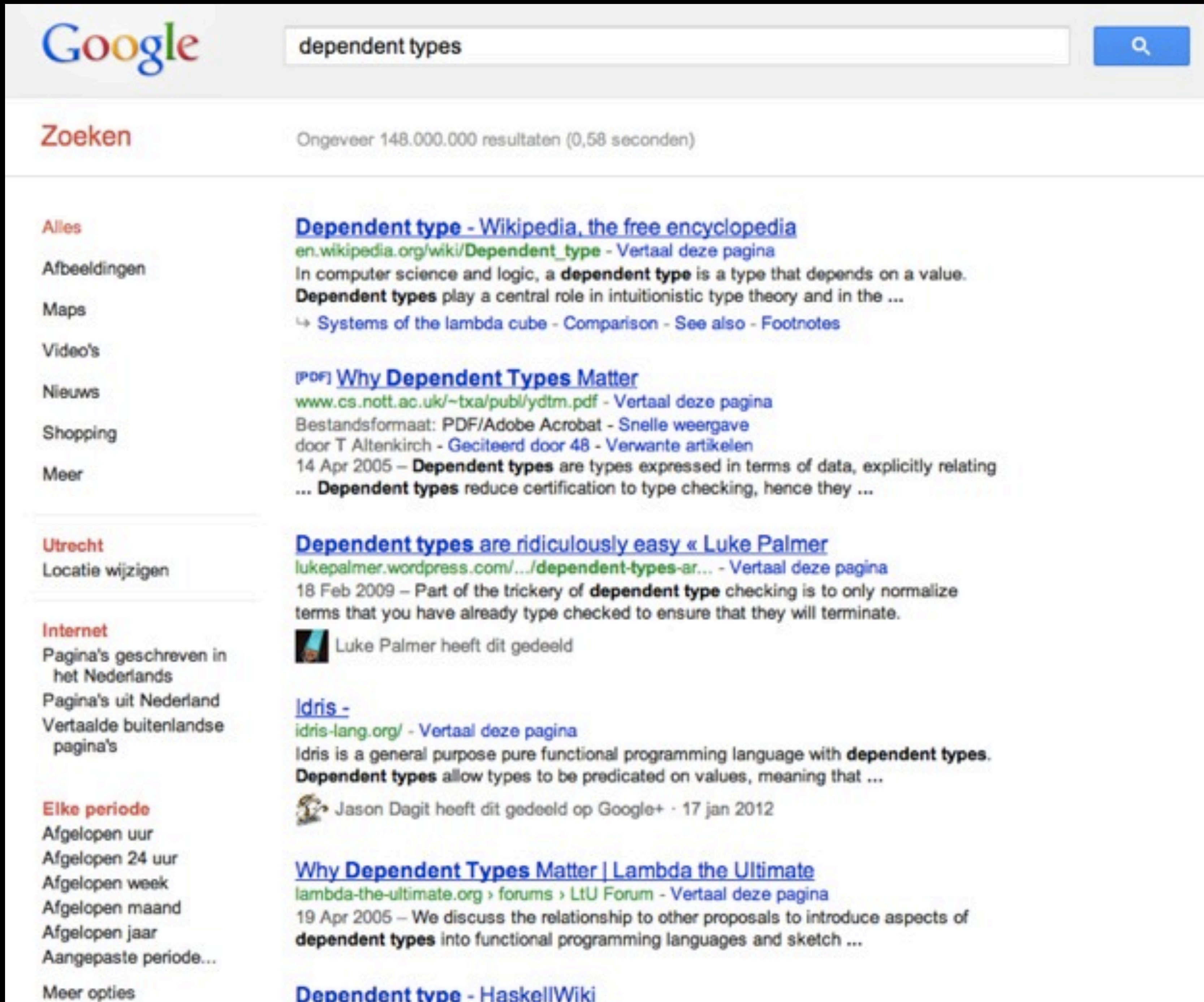
Seminar on Dependently Typed Programming

Wouter Swierstra

23-04-13

What are dependent types?

Google


Google
dependent types
Zoeken
Ongeveer 148.000.000 resultaten (0,58 seconden)
Alles
Afbeeldingen
Maps
Video's
Nieuws
Shopping
Meer
Utrecht
Locatie wijzigen
Internet
Pagina's geschreven in het Nederlands
Pagina's uit Nederland
Vertaalde buitenlandse pagina's
Elke periode
Afgelopen uur
Afgelopen 24 uur
Afgelopen week
Afgelopen maand
Afgelopen jaar
Aangepaste periode...
Meer opties
Dependent type - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Dependent_type - Vertaal deze pagina
In computer science and logic, a dependent type is a type that depends on a value. Dependent types play a central role in intuitionistic type theory and in the ...
Systems of the lambda cube - Comparison - See also - Footnotes
[PDF] Why Dependent Types Matter
www.cs.nott.ac.uk/~txa/publ/ydtm.pdf - Vertaal deze pagina
Bestandsformaat: PDF/Adobe Acrobat - Snelle weergave
door T Altenkirch - Geciteerd door 48 - Verwante artikelen
14 Apr 2005 – Dependent types are types expressed in terms of data, explicitly relating ... Dependent types reduce certification to type checking, hence they ...
Dependent types are ridiculously easy « Luke Palmer
lukepalmer.wordpress.com/.../dependent-types-ar... - Vertaal deze pagina
18 Feb 2009 – Part of the trickery of dependent type checking is to only normalize terms that you have already type checked to ensure that they will terminate.
Luke Palmer heeft dit gedeeld
Idris -
idris-lang.org/ - Vertaal deze pagina
Idris is a general purpose pure functional programming language with dependent types. Dependent types allow types to be predicated on values, meaning that ...
Jason Dagit heeft dit gedeeld op Google+ · 17 jan 2012
Why Dependent Types Matter | Lambda the Ultimate
lambda-the-ultimate.org › forums › LtU Forum - Vertaal deze pagina
19 Apr 2005 – We discuss the relationship to other proposals to introduce aspects of dependent types into functional programming languages and sketch ...
Dependent type - HaskellWiki

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

Dependent type

From Wikipedia, the free encyclopedia

(Redirected from Dependent types)

In computer science and logic, a **dependent type** is a type that depends on a value. Dependent types play a central role in intuitionistic type theory and in the design of functional programming languages like ATS, Agda and Epigram.

An example is the type of *n*-tuples of real numbers. This is a dependent type because the type *depends* on the value *n*.

Deciding equality of dependent types in a program may require computations. If arbitrary values are allowed in dependent types, then deciding type equality may involve deciding whether two arbitrary programs produce the same result; hence type checking becomes undecidable.

The Curry–Howard correspondence implies that types can be constructed that express arbitrarily complex mathematical properties. If the user can supply a constructive proof that a type is *inhabited* (i.e., that a value of that type exists) then a compiler can check the proof and convert it into executable computer code that computes the value by carrying out the construction. The proof checking feature makes dependently typed languages closely related to proof assistants. The code-generation aspect provides a powerful approach to formal program verification and proof-carrying code, since the code is derived directly from a mechanically verified mathematical proof.

Contents [hide]

Systems of the lambda cube

Henk Barendregt developed the lambda cube as a means of classifying type systems along three axes. The eight corners of the resulting cube-shaped diagram each correspond to a type s
lambda calculus in the least expressive corner, and calculus of constructions in the most expressive. The three axes of the cube correspond to three different augmentations of the simply t
addition of dependent types, the addition of polymorphism, and the addition of higher kinded type constructors (functions from types to types, for example). The lambda cube is generalized
systems.

In computer science and logic, a **dependent type** is a type that depends on a value.

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**.

Can you see a pattern?

- From Data.Word

```
data Word
data Word8
data Word16
data Word32
data Word64
```

- What about UTF8, UTF16, UTF32?

What could possibly go wrong?

```
mallocBytes :: Int -> IO (Ptr a)
```

Source

Allocate a block of memory of the given number of bytes. The block of memory is sufficiently aligned for any of the basic foreign types that fits into a memory block of the allocated size.

The memory may be deallocated using `free` or `finalizerFree` when no longer required.

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
```

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
  - ...

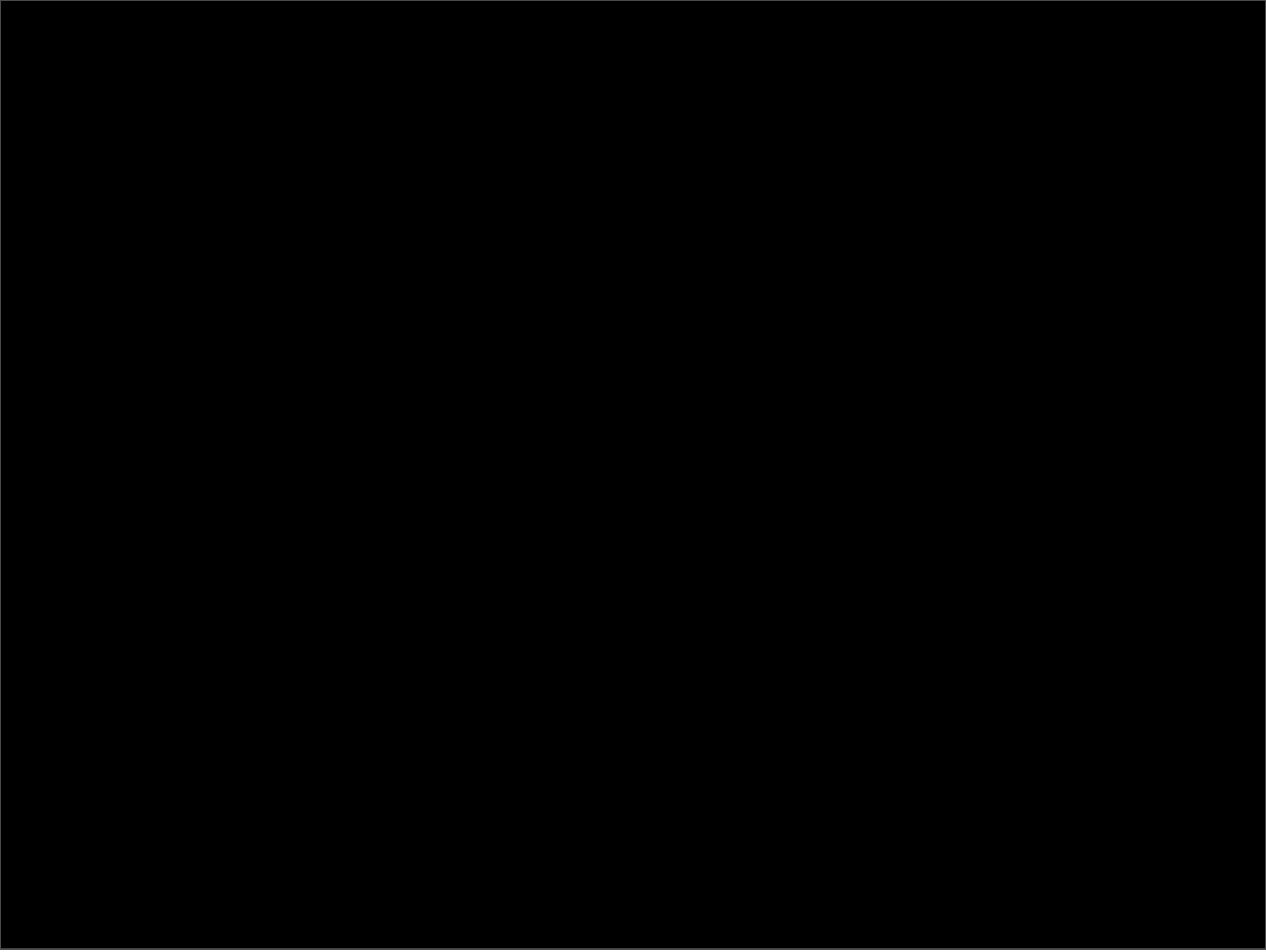
Precise Programming

Curry-Howard Isomorphism

- Dependent types provide a mathematical framework for doing formal 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 : Type) (xs : list a) :
  reverse (reverse xs) = xs.
```

Why should I care about dependent types?



More abstraction.

DTP Seminar

- The seminar will run from week 17–26.
- We will convene twice a week:
  - Tuesday 13:15–15:00
  - Thursday 09:00–10:45.
- You can earn 7.5 ECTCS.
- Watch the website for updates.

What will you learn?

- What are dependent types?
- What is the Curry Howard isomorphism?
- How can we use dependent types to verify software? Or write more precise code?
- How to use both Coq and Agda.
- What research is done in this area?

Course outline - I

- In the first week, I want to cover two papers:
  - Constructive mathematics and computer programming;
  - A tutorial implementation of a dependently typed lambda calculus;
- You should have some understanding of type theory when we complete these.

Course outline - II

- Next, I want to study (the first chapters of) the Software Foundations course. You will learn the basics of the Coq proof assistant:
  - how to write functional programs in Coq;
  - how to write specifications in Coq;
  - how to prove a program meets its spec in Coq.

Course outline - III

- In the last part of the lectures, I want to cover the programming language Agda.
  - how to design *indexed data types* capturing invariants in your code;
  - how to program using these types, exploiting *dependent pattern matching*.

Course outline – IV

- **You** will teach the last part of the course.
- In the last weeks of the course, each student must present a research paper or more advanced topic.

Grading

- Your grade will depend on four factors:
  - Presentation of research paper 30%
  - ‘Weekly’ individual exercises 30%
  - Formalization project & report 30%
  - Participation in seminar 10%

Presentations

- I’ve put up a list of suggested papers.
- Please have a look after this session and start thinking about your choice.
- The sooner you choose, the more time you have to prepare.
- I’m open to suggestions!

Projects

- I would like everyone to (try to) verify a non-trivial functional program using Coq;
- Work in pairs.

Verification in Coq

- Verification is **hard**.
- You may not completely finish.
- But you need to try.
- I've suggested a few topics from Richard Bird's *Pearls of Functional Algorithm* design.

Report

- I'd like to see a final report about your project.
  - What problem did you work on?
  - What was the spec?
  - What did you finish? What remains to be done?
- Sources will be pooled in a github repository

Questions?

A tutorial implementation

- Paper available on the wiki;
- Source code available from:

www.andres-loeh.de/LambdaPi/

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

Goal

Explain dependently typed lambda calculus by presenting a ‘simple’ implementation in Haskell.

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

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

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 bound variables.
- "The variable *k* is bound *k* lambdas up"
- Examples:

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

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

Locally nameless

- We use **de Bruijn indices** to represent bound variables.
- Variables that are not bound, i.e. **free variables**, will also be represented by a number.
- Our implementation is careful to distinguish between when a variable is free or bound.

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              -- free variables
 | 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: $\lambda x.x$
$x(\lambda y.yz)$

Evaluation – specification

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

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

How would you implement an evaluator?

Evaluation – implementation

- We will keep track of an **environment** containing a list of values for the variables that we have encountered so far.

Evaluation – examples

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

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

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

Values – implementation

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

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

- We need to write cases for:
  - Bound variables;
  - Free variables;
  - Application;
  - Lambdas.

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

Types – implementation

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

Type checking – examples

$$\lambda x : \sigma . x : \sigma \to \sigma$$

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

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?

Curry-Howard isomorphism

Haskell types

- Consider the “language” of Haskell types:
  - `forall a . a -> a`
  - `forall a b . a -> b -> a`
  - ...

Propositional logic

- Consider the language of first-order propositional logic:
  - `p → p`
  - `p → q → p`
  - `...`
- See the similarity?

What about type rules?

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

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

What about type rules?

Simple observation

- A type looks just like a propositional formula;
- Type rules look just like logical inference rules;
- A program “encodes” a proof derivation.

The correspondence

- functions
- pairs
- either
- application
- abstraction
- quantification?

- implication
- conjunction
- disjunction
- modus ponens
- implication introduction
- ...

What goes wrong?

- So why don't we use Haskell as a (first-order) proof assistant?
- The type `forall a . a` corresponds to False;
- In any sensible proof system, False should not be true;
- Yet in Haskell we have
  `undefined :: forall a . a`

Totality

- If you take this idea seriously, you need to care about when functions are **total**;
- Haskell cares about **purity** (when does a function have side-effects);
- Coq and Agda care about **totality** (when is a function guaranteed to compute an answer in finite time).

Consequence

- To make sure their type system corresponds to a sound logic, Coq and Agda place some restrictions on the programs you may write:
  - no missing case branches, `head []`
  - no general recursion (`iterate`, `repeat`), only folds over finite data.

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!

Next time

- Read Per Martin-Löf's *Constructive mathematics and computer programming.*