Seminar on Dependently Typed Programming

Wouter Swierstra

2-05-12

Last time:
implementing the
simply typed lambda
calculus

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

```
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
```

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

```
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])
```

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

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

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

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

On to dependent types

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

Type checking needs to perform evaluation!

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

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

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

What needs to change?

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

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

$$(a : *) \to a \to a \ : \ *$$

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)
    -- VPi domain range
    -- note the dependency!
```

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

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

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

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

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.

What is still missing?

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.

General pattern

- When is X a valid type?
- How are elements of X created? (constructors)
- What is the type of the fold over X? How does this fold compute?
- (When are two X types equal? When are two inhabitants of X equal?)

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
```

How can we make this more general?

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.

Extending values

```
data Val =
   VNat | VZero | VSucc Val |
   VNatElim Val Val Val Val ...
```

We need to add a new constructors to the value type.

Evaluation

```
evalInfer (NatElim m mz ms k) e =
 let mzVal = evalCheck mz e
     msVal = evalCheck ms e
     rec kVal = case kval of
       VZero -> mzVal
       VSucc v ->
         msVal `vapp` v `vapp` rec v
 -- and a branch for neutral elim
in rec (evalCheck k e)
```

Type checking - I

```
inferType i g Nat = return VStar
inferType i g Zero = return VNat
inferType i g (Succ k) = do
    checkType i g k VNat
    return VNat
```

Type checking - II

```
inferType i g (NatElim m mz ms k) = do
  checkType i g m (VPi VNat (const VStar))
  let mVal = eval m []
  checkType i g mz (mVal `vapp` VZero)
  checkType i g ms VPi VNat (\k ->
     VPi (mVal `vapp` k) (\_ ->
     mVal `vapp` (VSucc k)))
  checkType i g k VNat
  let kVal = eval k []
  return (mVal `vapp` kVal)
```

The paper & code

- There are a lot more examples of how to add new types to the system:
  - vectors;
  - finite types *Fin n*;
  - equality;
  - ...

Software Foundations

Benjamin C. Pierce
Chris Casinghino
Michael Greenberg
Vilhelm Sjöberg
Brent Yorgey

with Andrew W. Appel, Arthur Charguéraud, Anthony Cowley, Jeffrey Foster, Michael Hicks, Ranjit Jhala, Greg Morrisett, Chung-chieh Shan, Leonid Spesivtsev, and Andrew Tolmach

Coq

The Coq Proof Assistant

Welcome !

What is Coq ?

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the formalization of programming languages semantics (e.g. the CompCert compiler certification project or Java Card EAL7 certification in industrial context), the formalization of mathematics (e.g. the full formalization of the 4 color theorem or constructive mathematics at Nijmegen) and teaching.

More about Coq

How to get it ?

The stable version of Coq is version 8.3. Released in October 2010, it comes with many improvements of existing features, especially regarding the tactics, the module system, extraction, the type classes, the program command, libraries, coqdoc. It also includes a new tactic and a few new libraries.

The next version 8.4 is currently in beta testing phase.

Get Coq

Documentation

The reference documentation for Coq are the Reference Manual and the documentation of the Standard Library. Other useful documents (tutorials, faq, ...) are available from the documentation page.

Reference Manual Standard Library All documents

The current version: Coq 8.3

This version comes with many improvements of existing features, especially regarding the tactics, the module system, extraction, the type classes, the program command, libraries, coqdoc. It also includes a new tactic (`nsatz`, standing for Hilbert's NullStellensatz, that extends `ring` to systems of polynomial equations) and a few new libraries (a certification of mergesort, a new library of finite sets with computational and logical contents separated). For a full log of changes, see the file CHANGES.

Coq 8.3 is not entirely upward compatible with 8.2. The major incompatibilities can be easily treated by using the new `-compat 8.2` option or by setting/unsetting adequate options. See the file COMPATIBILITY for details and migration recommendations.

Sources		
	coq-8.3pl4.tar.gz	3.7 MB
Binaries		
Windows	coq-8.3pl3-win-0.exe (bundled with CoqIDE)	49 MB
MacOS	coq-8.3pl4-macosx.dmg coqide-8.3pl4-macosx.dmg (Coq bundled with CoqIDE interface) *The packages require MacOS ≥ 10.5*	61 MB 78 MB
Documentation		
	Tutorial.pdf Reference-Manual.pdf	0.2 MB 1.5 MB

On Linux, BSD or MacOS package-based distributions, Coq is released by the corresponding maintainers. For reports on the Windows package and the MacOS `dmg` packages, use the Coq bug tracker.

Reference Manual

Version 8.3[1]

The Coq Development Team

The Coq Standard Library

Here is a short description of the Coq standard library, which is distributed with the system. It provides a set of modules directly available through the `Require Import` command.

The standard library is composed of the following subdirectories:

Init: The core library (automatically loaded when starting Coq)
Notations Datatypes Logic Logic_Type Peano Specif Tactics Wf (Prelude)

Logic: Classical logic and dependent equality
SetIsType Classical_Pred_Set Classical_Pred_Type Classical_Prop Classical_Type (Classical) ClassicalFacts Decidable Eqdep_dec EqdepFacts Eqdep JMeq ChoiceFacts RelationalChoice ClassicalChoice ClassicalDescription ClassicalEpsilon ClassicalUniqueChoice Berardi Diaconescu Hurkens ProofIrrelevance ProofIrrelevanceFacts ConstructiveEpsilon Description Epsilon IndefiniteDescription FunctionalExtensionality

Structures: Algebraic structures (types with equality, with order, ...). DecidableType* and OrderedType* are there only for compatibility.
Equalities EqualitiesFacts Orders OrdersTac OrdersAlt OrdersEx OrdersFacts OrdersLists GenericMinMax DecidableType DecidableTypeEx OrderedType OrderedTypeAlt OrderedTypeEx

Bool: Booleans (basic functions and results)
Bool BoolEq DecBool IfProp Sumbool Zerob Bvector

Arith: Basic Peano arithmetic

What is Coq ?

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the formalization of programming languages semantics (e.g. the CompCert compiler certification project or Java Card EAL7 certification in industrial context), the formalization of mathematics (e.g. the full formalization of the 4 color theorem or constructive mathematics at Nijmegen) and teaching.

More about Coq

What is Coq?

- A total functional programming language...
- ... with dependent types
- ... and a *tactic* language to help write proofs.

Coq IDE

- Most binaries come with CoqIDE, which allows you to interactively complete Coq proofs;
- Emacs users may prefer ProofGeneral.
- **Homework:** be sure to have a working Coq setup before the next lecture!

Software foundations

- HTML and Coq code available from:

  www.cis.upenn.edu/~bcpierce/sf

Coq Demo

Homework

- Install Coq and try to write a simple lambda calculus expressions for S, K, and I.
- Have a look at the suggested projects. Email me about your choice of partner and at least two topics;
- Study suggested papers. Email me with your preferences.