Concepts of programming languages

Lecture 6

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Last time

- What is the λ-calculus?
- How can we define its semantics?
- How can we define its type system?

This time

- Relating evaluation and types
- How to handle variable binding in embedded languages?
- Standalone or embedded? Pros and cons of each.

Evaluation and typing

Last time we saw two key relations defining the *static* and *dynamic* semantics of the typed lambda calculus:

1. A typing relation $\Gamma \vdash t : \tau$, describing when a term is well-typed;
2. A reduction relation $t \to t'$, describing a single reduction step.

What can we prove about these relations?

Properties - I

- Reduction is deterministic:

if $t \to t_1$ and $t \to t_2$ then $t_1 = t_2$

- There is only ever a single choice of type assignment:

if $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \sigma$ then $\tau = \sigma$

Properties - II

- **Type preservation:**

if $\Gamma \vdash t : \tau$ and $t \to t'$ then $\Gamma \vdash t' : \tau$

- **Progress:**

for all terms t either t is a value or there exists a term t' such that $t \to t'$

Type soundness

Progress and preservation together give what is known as **type soundness**

Why are progress and preservation so important?

Together they guarantee that **any** program of type τ can be evaluated to a value of type τ.

Robin Milner: *Well-typed programs don't go wrong!*

Proofs?

The proofs of these properties are not too complicated – you can find them in Pierce's *Types and programming languages* if you're interested.

They all follow the same pattern: rule induction over the typing or reduction relation.

The proofs are all 'mathematically boring' – they don't require any particularly deep insight, but rather rely on careful bookkeeping.

This is A Good Thing: it means that our definitions are right.

Example: uniqueness types (Var)

Suppose that $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \sigma$. We show, by induction on a derivation of $\Gamma \vdash t : \tau$ that $\sigma = \tau$.

We'll cover the case for variables.

Suppose the first derivation is an application of the variable rule, then t must be a variable x.

The only rule that can be used to prove $\Gamma \vdash x : \sigma$ is the same rule for variables.

From $x : \sigma \in \Gamma$ and $x : \tau \in \Gamma$, we conclude $\sigma = \tau$.

Example: uniqueness types (App)

Suppose that $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \sigma$. We show, by induction on a derivation of $\Gamma \vdash t : \tau$ that $\sigma = \tau$.

Suppose the first derivation is an application of the App rule, then t must be an application $f\ x$.

Since there is only one rule for applications, the second derivation must also be built using the App rule.

Our induction hypotheses guarantee that both subtrees of the applications have equal types. In particular, the subtrees for f must have equal types the form $\rho \to \sigma$ and $\rho \to \tau$. Hence we conclude $\sigma = \tau$.

Other cases & other proofs

Similar proofs can be done for other branches.

Similar proofs can be done for other properties.

There is quite an active field of research into *automating* these proofs, or at least using computers to check their validity.

Check out Pierce's more recent work, *Software Foundations*, that uses the Coq proof assistant to formalize the metatheory covered by *Types and semantics*.

Normalisation

The only interesting proof about the typed lambda calculus is the proof that:

for all t for which we can find a typing derivation $\vdash t : \tau$, there exists a value v such that $t \to^* v$.

This structure of this proof relies on the type of t – unsurprisingly, as this property does not hold for the untyped lambda calculus.

The underlying proof techniques (logical relations) go beyond the scope of this course.

Back to variable binding

In the SVG example, we saw a simple approach to handling variable binding in an embedded language.

what about embedding more complex (programming) languages?

How can we handle **variable binding** in our object language?

This question pops up again and again when studying programming languages.

The problem of variable binding

To better study this problem, we'll focus on embedding the (untyped) lambda calculus.

I want to go through a variety of techniques for embedding this and handling name binding.

Of course, it's kind of strange to embed a lambda calculus into a language like Haskell:

- you want to inspect the binding structure of object language;
- you want to make the recursive structure or sharing of the object language *observable*;
- …

Named variables – aka the obvious thing

The simplest approach to name binding is to use strings for variable names:

```
type Name = String

data Term =
    Lambda Name Term
  | App Term Term
  | Var Name

idTerm : Term
idTerm = Lambda "x" (Var "x")
```

Named variables – canonicity

This representation is not **canonical** – two alpha equivalent terms have different representations:

```
idTerm' : Term
idTerm' = Lambda "y" (Var "y")
```

Despite being alpha equivalent, `idTerm` and `idTerm'` are not equal.

Named variables – ill-scoped terms

Typically, we may want to enforce **statically** that we can only write **closed** lambda terms.

In the same way that we only consider Haskell programs valid if all variables are defined, we want to ensure our lambda terms are closed.

```
notClosed : Term
notClosed = Var "free!"

ok : Term
ok = Lam "free!" notClosed
```

In this setting, such checks become **dynamic**…

Named variables – computing free variables

We can compute the free variables of such expressions easily enough:

```
freeVars :: Term -> [Name]
freeVars (Lam nm body)  = freeVars body \\ [nm]
freeVars (Var nm)       = [nm]
freeVars (App f x)      =
  freeVars f `union` freeVars x
```

```
isClosed :: Term -> Bool
isClosed t = null (freeVars t)
```

But this still does not prevent us from writing ‘bad’ terms.

Named variables – substitution

Writing scope preserving substition is not very easy: to avoid accidental name capture you sometimes need to rename variables.

```
subst :: Name -> Term -> Term
subst x e t = sub t
  where
  sub e@(Var i) = if i == v then x else e
  sub (App f a) = App (sub f) (sub a)
  sub (Lam i e) = ...
```

The case for lambda's is a bit messy…

Capture avoiding substitutions (lambda)

If we want to substitute the term `t` for for the variable `x` in the term `Lam y b` we need to check:

- if `x` is equal to `y`, the substitution should have no effect;
- if `y` occurs freely in `t`, we need to rename `y` to avoid capture;
- otherwise, we can safely substitute in the body of the lambda `b`

How messy? (From Lennart Augustsson's blog)

```
sub (Lam i e) =
  if v == i then
      Lam i e
  else if i `elem` fvx then
      let i' = fresh e i
          e' = substVar i i' e
      in  Lam i' (sub e')
  else
      Lam i (sub e)
fvx = freeVars x
fresh e i = ...
```

The `fresh` function generates a name that is not yet used.

Addressing these limitations

What other options do we have?

Addressing these limitations

What other options do we have?

Choose a representation of lambda terms that is **canonical** – ensuring that two alpha equivalent terms have an equal representation.

De Bruijn indices

Instead of giving variables a *name*, represent variables using a *number*.

This number represents the number of lambda's to the binding occurrence.

So 0 refers to the most recently bound variable; 1 refers to the variable before that; etc.

Example:

$$\lambda\, \lambda\, 1$$

De Bruijn indices

Instead of giving variables a *name*, represent variables using a *number*.

This number represents the number of lambda's to the binding occurrence.

So 0 refers to the most recently bound variable; 1 refers to the variable before that; etc.

Example:

$$\lambda\, \lambda\, 1$$

Corresponds to Haskell's `const` function `\x y -> x`

De Bruijn indices

This idea is easy enough to implement:

```
type Name = Int
data Term =
  Lambda Term
  | App Term Term
  | Var Name
```

Note: Lambda terms no longer name the bound variables.

Two alpha equivalent lambda terms have **exactly** the same representation.

De Bruijn indices – free variables

To decide if a term is closed or not is easy enough: does every variable refer to a binding lambda?

```
isClosed : Term -> Bool
isClosed t = go 0 t
  where
  go : Int -> Term -> Bool
  go n (Var i) = i < n
  go n (App f x) = go n f && go n x
  go n (Lam b) = go (n + 1) b
```

Similarly, defining a capture avoiding substitution is fiddly, but feasible.

De Bruijn indices – writing terms

Although using De Bruijn indices fixes some of the drawbacks associated with named variables, we still have two major problems:

- we can still write non-closed terms;
- writing any non-trivial term is a pain.

We'll fix the first problem first…

Variable binding

As a first approximation, lets abstract over the type used to represent names.

```
data Term name =
  Lambda (Term name)
  | App (Term name) (Term name)
  | Var name
```

We can now choose to instantiate the `name` variable differently…

Getting there…

For example, we can choose a fixed set of variables:

```
data MyVars = X | Y | Z
type MyTerm = Term MyVar
```

Now we can only use `X`, `Y` and `Z` for our variables – nice!

But there is no connection with the binding structure of our language. In particular, lambdas don't increase the set of available variables.

A solution, Maybe?

The body of the lambda should have **one more variable** – to do this we modify the type of the lambda body:

```
data Term name =
  Lambda (Term (Maybe name))
  | App (Term name) (Term name)
  | Var name
```

In our example type, `Term MyVars`, we have three variables (`X`,`Y` and `Z`). After going under a lambda, we represent variables using `Maybe MyVars`:

- the `Nothing` constructor refers to the most recently bound variable;
- for any `v : MyVars`, we can use `Just v` to refer to an existing variable.

A solution

This ensures **statically** that for any term of type `Term names` may only uses free variables drawn from `names`.

In particular, if `names` has no variables, the term is closed:

```
data Empty
type Closed = Term Empty
```

The `Empty` type has no inhabitants; hence, `Closed` terms have no variables, except for those bound by a lambda.

Example

To define the `const` function we can now write:

```
const : Term
const = Lambda (Lambda (Just Nothing))
```

This is not particularly easy to read.

A solution?

Manipulating such terms is even more painful than De Bruijn indices…

We lose a lot of efficiency – checking if two variables are equal is no longer just comparing two integers.

And writing such terms is even worse than writing terms using De Bruijn indices.

There is a clear price to pay.

Yet another option

The problem we're facing is that specifying variable binding of our **object language** (the lambda calculus) is hard in our **host language** (Haskell).

But Haskell already has its own notion of lambdas and variables – why not re-use those?

Higher-order abstract syntax (HOAS)

We can define the following `Term` data type:

```
data Term name =
  Lambda (Term -> Term)
  | App (Term name) (Term name)
```

- Note that the `Lambda` constructor stores a **function** – our abstract syntax tree is now **higher-order**;
- We no longer have a constructor for variables. Instead, we will piggyback on Haskell's variables for the object language.

HOAS – example

```
idTerm :: Term
idTerm = Lambda (\x -> x)

constTerm :: Term
constTerm = Lambda (\x -> Lambda (\y -> x))

weird :: Term
weird = App (Lambda (\x -> x x)) (Lambda (\x -> x x))
```

Evaluating with HOAS

If all we care about is evaluation, however, HOAS is perfect:

```
eval :: Term -> Term
eval (Lambda t) = Lambda t
eval (App f x) = eval f `app` eval x
  where
  app (Lambda f) x = f x
```

We can re-use Haskell's evaluation mechanism!

Working with HOAS

When defining other interpretations of such ‘higher-order’ abstract syntax trees, we run into problems.

For example, trying to count the number of lambdas in a term:

```
countLambdas :: Term -> Int
countLambdas (App f x) =
  countLambdas f + countLambdas x
countLambdas (Lambda f) = 1 + ...
```

HOAS revisited

Using this simple HOAS term type, we cannot easily define intepretations beyond evaluation.

- Can we provide a HOAS-interface to another representation?
- Can we define a variation of HOAS that does allow other interpretations?

de Bruijn to HOAS

Fortunately, we can add a HOAS interface on top of an underlying De Bruijn representation:

```
type Name = Int
data Term =
  Lambda Term
  | App Term Term
  | Var Name

lambda :: (Term -> Term) -> Term
lambda f = ...
```

de Bruijn to HOAS

Fortunately, we can add a HOAS interface on top of an underlying De Bruijn representation:

```
type Name = Int
data Term =
  Lambda Term
  | App Term Term
  | Var Name

lambda :: (Term -> Term) -> Term
lambda f = Lambda (f (Var 0))
```

Unfortunately, this does not always work.

De Bruijn to HOAS

Simply passing `Var 0` will work for some terms.

Take the identity function, for example:

```
lambda :: (Term -> Term) -> Term
lambda f = Lambda (f (Var 0))

id :: Term
id = lambda (\x -> x)
   = Lamda ((\x -> x) (Var 0))
   = Lambda (Var 0)
```

de Bruijn to HOAS

But simply applying the argument function to `Var 0` may result in accidental capture of variables:

```
constF :: Term -> Term -> Term
constF x y = lambda (\x -> lambda (\y -> x))
           = Lambda (lambda (\y -> Var 0))
           = Lambda (Lambda Var 0)
```

We were hoping to get `Lambda (Lambda (Var 1))`...

de Bruijn to HOAS

What went wrong?

Our host language doesn't know about our object language's binding structure.

Using our host languages substitution mechanism (through function application) is not *capture avoiding*.

We need to maintain information about our object level binders in the translation.

de Bruijn to HOAS

To resolve this, we keep track of the number of binders we are currently under:

```
type DB = Int -> Term

lambda :: (DB -> DB) -> DB
lambda f = \i -> let v = \j -> Var (j-(i+1)) in
                 Lambda (f v (i+1))
```

- the new variable lives at level 0 initially `i+1 - (i+1)`;
- as we go under more binders, `j` increases, so we avoid accidental capture;
- we can close any `DB` by applying it to 0.

HOAS revisited

Using this simple HOAS term type, we cannot easily define inteprertations beyond evalaution.

- Can we provide a HOAS-interface to another representation?
- Can we define a variation of HOAS that does allow other interpretations?

HOAS problems

We cannot easily define alternative interpretations once we fix the type being abstracted over in the higher-order abstract syntax tree:

But what if we try to keep this abstract?

```
data Term a =
    App (Term a) (Term a)
  | Lambda (a -> Term a)
```

This almost works…

HOAS

If we try to define the identity function:

```
id : Term a
id = Lambda (\x -> ...)
```

We have no way to turn the variable `x : a` back into a term.

We need to introduce an explicit variable constructor.

PHOAS

The resulting term data type becomes:

```
data Term a =
    App (Term a) (Term a)
  | Lambda (a -> Term a)
  | Var a

idTerm : Term
idTerm = Lambda (\x -> Var x)
```

This is sometimes known as *parametric higher order abstract syntax* or PHOAS.

PHOAS

By quantifying over all types `a` as follows:

```
data Term a =
    App (Term a) (Term a)
  | Lambda (a -> Term a)
  | Var a
type ClosedTerm = forall a . Term a
```

We know that a value of type `ClosedTerm` must indeed correspond to closed lambda terms.

Why? Try using the `Var` constructor – you don't know which type to pass it, so you cannot find a suitable argument.

```
failure :: ClosedTerm
failure = Var ...
```

Alternative interpretations

One appealing aspect of PHOAS is that it *is* possible to define alternative interpretations of terms:

```
showTerm :: ClosedTerm -> String
showTerm c = showT 0 c
  where
  showT :: Int -> Term String -> String
  showT nm (Var i) = i
  showT nm (App f x) =
    "(" ++ showT nm f ++ ")(" ++ showT nm x ++ ")"
  showT nm (Lambda f) =
    let x = "x" ++ show nm in
    showT (nm + 1) (f x)
```

The `Var` case is always the same; the `Lambda` case chooses how to handle variables.

Handling binding

- Strings
- De Bruijn variables
- Well-scoped terms using Maybe
- HOAS
- PHOAS

Each approach has its own advantages and disadvantages.

But most are interchangeable…

DSLs: approaches

A stand-alone DSL typically has *limited expressivity* and requires writing a parser/interpreter from scratch.

An embedded DSL can re-use the host language's features, but is constrained by the host language's syntax and semantics.

Challenge

Suppose we want to have a variant of Markdown where we can have both computation and formatting.

```
# Fibonacci

The first 5 fibonacci numbers
are @bulletList (map show (fibs 5))@
```

Stand-alone

If we try to define our own Markdown flavour that allows you to mix computation in the layout…

Stand-alone

If we try to define our own Markdown flavour that allows you to mix computation in the layout…

We end up needing to implement our own programming language.

Embedded

We can define a simple enough Haskell data type for representing Markdown:

```
data MDElt =
  Title Depth String
  | Bullets [MD]
  | Text String
  ...
type MD = [MDElt]
```

Embedded

But working with this language is awkward.

We need *some* computation, but we're mostly interested in writing strings and formatting them.

```
example =
  [Title 3 "Fibonacci numbers"
  , Text "The first five fibonacci numbers are"
  , Bullets (fibMD 5)]
  where
  fibMD :: Int -> MD
```

This really isn't user-friendly – we're mostly interested in **data** (strings), rather than **computations** (Haskell code).

What can we do?

Next time, we'll start studying **reflection** in the context of Racket and Haskell.

This will enable us to define new languages, mixed in with our host language.

Recap

- Relating evaluation and types
- How to handle variable binding in embedded languages?
- Deep vs shallow embeddings revisited
- Standalone or embedded: challenges

Further reading

- Types and programming languages by Benjamin Pierce – gives a solid account of the semantics of simple programming languages, including formal proofs of type soundness and implementations in ML.
- Unembedding Domain-Specific Languages by Robert Atkey, Sam Lindley, and Jeremy Yallop – shows how to convert between different binding representations
- The lambda calculus cooked four ways by Lennart Augustsson – compares four different approaches to embedding the lambda calculus in Haskell, including performance and ease of use.
- Abstract Syntax Graphs for Domain Specific Languages by Bruno C. d. S. Oliveira and Andres Löh – gives an overview of PHOAS