Programming Language Concepts: Lecture 21

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 21, 08 April 2009

Type inference with shallow types

Consider

```
applypair f x y = (f x,f y)
```

Type inference with shallow types

Consider

```
applypair f x y = (f x,f y)
```

Is the following expression well typed, where `id z = z`?

```
applypair id 7 'c' = (id 7, id 'c') = (7,'c')
```

Type inference with shallow types

Consider

```
applypair f x y = (f x,f y)
```

Is the following expression well typed, where `id z = z`?

```
applypair id 7 'c' = (id 7, id 'c') = (7,'c')
```

We have to unify the following set of constraints

```
id  :: a -> a
7   :: Int
'c' :: Char
a = Int              (from id 7)
a = Char             (from id 'c')
```

Type inference with shallow types

Consider

```
applypair f x y = (f x,f y)
```

Is the following expression well typed, where `id z = z`?

```
applypair id 7 'c' = (id 7, id 'c') = (7,'c')
```

We have to unify the following set of constraints

```
id  :: a -> a
7   :: Int
'c' :: Char
a = Int             (from id 7)
a = Char            (from id 'c')
```

Not possible! Haskell compiler says

```
applypair :: (a -> b) -> b -> b -> (b,b)}
```

Type inference with shallow types

In the λ-calculus, we have

$$\lambda fxy.pair\ (fx)(fy), \text{ where } pair \equiv \lambda xyz.(zxy)$$

Type inference with shallow types

In the λ-calculus, we have

$$\lambda fxy.pair\ (fx)(fy), \text{ where } pair \equiv \lambda xyz.(zxy)$$

When we pass a value for f, it has to unify with types of both x and y

Type inference with shallow types

In the λ-calculus, we have

$$\lambda fxy.pair\ (fx)(fy), \text{ where } pair \equiv \lambda xyz.(zxy)$$

When we pass a value for f, it has to unify with types of both x and y

Suppose, we write, instead

```
applypair x y = (f x,f y) where f z = z
```

Type inference with shallow types

In the λ-calculus, we have

$$\lambda fxy.pair\ (fx)(fy), \text{ where } pair \equiv \lambda xyz.(zxy)$$

When we pass a value for f, it has to unify with types of both x and y

Suppose, we write, instead

```
applypair x y = (f x,f y) where f z = z
```

Now, we have

```
applypair :: a -> b -> (a,b)
```

Type inference with shallow types

In the λ-calculus, we have

$$\lambda fxy.pair\ (fx)(fy), \text{ where } pair \equiv \lambda xyz.(zxy)$$

When we pass a value for f, it has to unify with types of both x and y

Suppose, we write, instead

```
applypair x y = (f x,f y) where f z = z
```

Now, we have

```
applypair :: a -> b -> (a,b)
```

What's going on?

Type inference with shallow types

Extend λ-calculus with "local" definitions, like `where`

$$\Lambda = C_i \mid x \mid \lambda x.M \mid MN \mid \text{let } f = e \text{ in } M$$

Type inference with shallow types

Extend λ-calculus with "local" definitions, like `where`

$$\Lambda = C_i \mid x \mid \lambda x.M \mid MN \mid \text{let } f = e \text{ in } M$$

Here is the λ-term for the second version of `applypair`

$$\text{let } f = \lambda z.z \text{ in } \lambda xy.pair\ (fx)(fy)$$

Type inference with shallow types

Extend λ-calculus with "local" definitions, like `where`

$$\Lambda = C_i \mid x \mid \lambda x.M \mid MN \mid \text{let } f = e \text{ in } M$$

Here is the λ-term for the second version of `applypair`

$$\text{let } f = \lambda z.z \text{ in } \lambda xy.pair\ (fx)(fy)$$

In fact, Haskell allows both

```
let f z = z in applypair x y = (f x,f y)
```

and

```
applypair x y = (f x,f y) where f z = z
```

Type inference with shallow types

- $\text{let } f = e \text{ in } \lambda x.M$ and $(\lambda f x.M)e$ are equivalent with respect to β-reduction

Type inference with shallow types

- $\text{let } f = e \text{ in } \lambda x.M$ and $(\lambda fx.M)e$ are equivalent with respect to β-reduction
- . . . but type inference works differently for the two

Type inference with shallow types

- $\text{let } f = e \text{ in } \lambda x.M$ and $(\lambda f x.M)e$ are equivalent with respect to β-reduction
- ... but type inference works differently for the two
- One may be typeable while the other is not
  - $(\lambda I.(II))(\lambda x.x)$
  - $\text{let } I = \lambda x.x \text{ in } (II)$

Type inference with shallow types

Type inference for $M = \text{let } f = e \text{ in } M'$

Type inference with shallow types

Type inference for $M = \text{let } f = e \text{ in } M'$

First attempt

- Assume $f :: t$ where $\alpha, \beta, \ldots$ are type variables occurring in t
- Make a separate copy of type variables for each instance of f in M'

Type inference with shallow types

Type inference for $M = \text{let } f = e \text{ in } M'$

First attempt

- Assume $f :: t$ where $\alpha, \beta, \ldots$ are type variables occurring in t
- Make a separate copy of type variables for each instance of f in M'

Example

- $\text{let } f = \lambda z.z \text{ in } \lambda xy.pair\ (fx)(fy)$
- First instance of f has type $\alpha_1 \rightarrow \beta_1$
- Second instance of f has type $\alpha_2 \rightarrow \beta_2$

Type inference with shallow types

A subtle problem

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag      = pair w
    pair s t = (s,t)
```

Type inference with shallow types

A subtle problem

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag        = pair w
    pair s t = (s,t)
```

- `applypair2 w x y` $\rightsquigarrow$ `((w,x),(w,y))`
- Type should be
  `applypair2 ::  a -> b -> c -> ((a,b),(a,c))`

Type inference with shallow types

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag        = pair w
    pair s t = (s,t)
```

Type inference

```
applypair2 :: a -> b -> c -> (d,e)
pair       :: f -> g -> (f,g)
tag        :: h -> (i,h)
```

Type inference with shallow types

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag      = pair w
    pair s t = (s,t)
```

Type inference

```
applypair2 :: a -> b -> c -> (d,e)
pair       :: f -> g -> (f,g)
tag        :: h -> (i,h)
```

- `a = i` because `tag` uses input `w` from `applypair2`

Type inference with shallow types

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag       = pair w
    pair s t = (s,t)
```

Type inference

```
applypair2 :: a -> b -> c -> (d,e)
pair       :: f -> g -> (f,g)
tag        :: h -> (i,h)
```

- `a = i` because `tag` uses input `w` from `applypair2`
- Using `let` rule, two instances of `tag` get different types
  - `d = h1 -> (i1,h1)`
  - `e = h2 -> (i2,h2)`

Type inference with shallow types

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag      = pair w
    pair s t = (s,t)
```

Type inference

```
applypair2 :: a -> b -> c -> (d,e)
pair       :: f -> g -> (f,g)
tag        :: h -> (i,h)
```

- `a = i` because `tag` uses input `w` from `applypair2`
- Using `let` rule, two instances of `tag` get different types
  - `d = h1 -> (i1,h1)`
  - `e = h2 -> (i2,h2)`
- End up with
  `applypair2 ::  a -> b -> c -> ((i1,b),(i2,c))`
  - The connection `a = i = i1 = i2` is lost!

Type inference with shallow types

- In `tag :: h -> (i,h)`
  - `h` is local to `tag`
  - `i` is unified with type passed directly to main function

Type inference with shallow types

- In `tag :: h -> (i,h)`
  - `h` is local to `tag`
  - `i` is unified with type passed directly to main function
- `h` is called a generic variable
  - Should not make copies of non-generic variables!

Type inference with shallow types

- In `tag :: h -> (i,h)`
  - `h` is local to `tag`
  - `i` is unified with type passed directly to main function

- `h` is called a generic variable
  - Should not make copies of non-generic variables!

Correct type inference rule for $M = \text{let } f = e \text{ in } M'$

- Assume $f :: t$ where $\alpha, \beta, \ldots$ are generic type variables occurring in t
- Make a separate copy of these generic type variables for each instance of f in M'
- Non-generic variables retain their name across all copies of f

Logic programming

- Programming with relations . . .
  - . . . as opposed to programming with functions

Logic programming

- Programming with relations ...
  - ... as opposed to programming with functions
- Function f with n arguments defines a relation R_f with $n+1$ arguments

$$f(x_1, x_2, \ldots, x_n) = y \text{ iff } (x_1, x_2, \ldots, x_n, y) \in R_f$$

Logic programming

- Programming with relations ...
  - ... as opposed to programming with functions
- Function f with n arguments defines a relation R_f with $n+1$ arguments

$$f(x_1, x_2, \ldots, x_n) = y \text{ iff } (x_1, x_2, \ldots, x_n, y) \in R_f$$

- Functional programs compute y given $(x_1, x_2, \ldots, x_n)$

Logic programming

- Programming with relations ...
  - ... as opposed to programming with functions
- Function f with n arguments defines a relation R_f with $n+1$ arguments

$$f(x_1, x_2, \ldots, x_n) = y \text{ iff } (x_1, x_2, \ldots, x_n, y) \in R_f$$

- Functional programs compute y given $(x_1, x_2, \ldots, x_n)$
- Logic programming allows computation of more general relations

Logic programming

- Programming with relations …
  - … as opposed to programming with functions
- Function f with n arguments defines a relation R_f with $n+1$ arguments

$$f(x_1, x_2, \ldots, x_n) = y \text{ iff } (x_1, x_2, \ldots, x_n, y) \in R_f$$

- Functional programs compute y given $(x_1, x_2, \ldots, x_n)$
- Logic programming allows computation of more general relations
- Will follow `Prolog` syntax

Variables and constants

Two kinds of entities

- Variables
  - Names starting with a capital letter
  - `X`, `Y`, `Name`, ...

Variables and constants

Two kinds of entities

- Variables
  - Names starting with a capital letter
  - `X`, `Y`, `Name`, . . .
- Constants
  - Names starting with a small letter
  - `ball`, `node`, `graph`, `a`, `b`, ...

Variables and constants

Two kinds of entities

- Variables
  - Names starting with a capital letter
  - `X`, `Y`, `Name`, ...
- Constants
  - Names starting with a small letter
  - `ball`, `node`, `graph`, `a`, `b`, ...
  - Uninterpreted — no types like `Char`, `Bool` etc!

Variables and constants

Two kinds of entities

- Variables
  - Names starting with a capital letter
  - `X`, `Y`, `Name`, ...
- Constants
  - Names starting with a small letter
  - `ball`, `node`, `graph`, `a`, `b`, ...
  - Uninterpreted — no types like `Char`, `Bool` etc!
  - Exception: natural numbers, some arithmetic

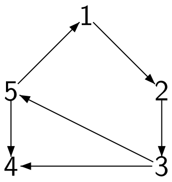
Defining relations

A Prolog program describes a relation

Defining relations

A Prolog program describes a relation

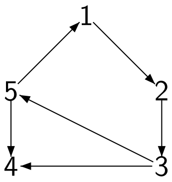
Example: A graph



Defining relations

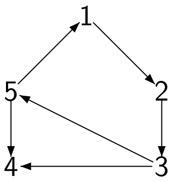
A Prolog program describes a relation

Example: A graph



- Want to define a relation `path(X,Y)`
- `path(X,Y)` holds if there is a path from `X` to `Y`

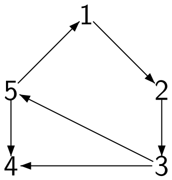
Facts and rules



Represent edge relation using the following facts.

```
edge(3,4).
edge(5,4).
edge(5,1).
edge(1,2).
edge(3,5).
edge(2,3).
```

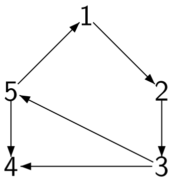
Facts and rules . . .



Define path using the following rules.

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Facts and rules . . .



Define path using the following rules.

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Read the rules read as follows:

Rule 1 For all X,Y, (X,Y) $\in$ path if X is same as (i.e., unifies with) Y.

Rule 2 For all X,Y, (X,Y) $\in$ path if there exists Z such that (X,Z) $\in$ edge and (Z,Y) $\in$ path.

Facts and rules . . .

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 . . . and Premise_n

Facts and rules . . .

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 . . . and Premise_n

  - if is written `:-`
  - and is written `,`

Facts and rules . . .

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 . . . and Premise_n

  - if is written `:-`
  - and is written `,`
  - This type of logical formula is called a Horn Clause

Facts and rules . . .

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 . . . and Premise_n

  - if is written `:-`
  - and is written `,`
  - This type of logical formula is called a Horn Clause

- Quantification of variables

Facts and rules . . .

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 . . . and Premise_n

  - if is written `:-`
  - and is written `,`
  - This type of logical formula is called a Horn Clause
- Quantification of variables
  - Variables in goal are universally quantified
    - `X`, `Y` above

Facts and rules ...

```
path(X,Y) :- X = Y.
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- Each rule is of the form

  Conclusion if Premise_1 and Premise_2 ... and Premise_n

  - if is written `:-`
  - and is written `,`
  - This type of logical formula is called a Horn Clause
- Quantification of variables
  - Variables in goal are universally quantified
    - `X`, `Y` above
  - Variables in premise are existentially quantified
    - `Z` above

Computing in Prolog

- Ask a question (a query)

```
?- path(3,1).
```

Computing in Prolog

- Ask a question (a query)

```
?- path(3,1).
```

- Prolog scans facts and rules top-to-bottom
  - 3 cannot be unified with 1, skip Rule 1.

Computing in Prolog

- Ask a question (a query)

```
?- path(3,1).
```

- Prolog scans facts and rules top-to-bottom
  - 3 cannot be unified with 1, skip Rule 1.
  - Rule 2 generates two subgoals. Find Z such that
    - (3,Z) $\in$ edge and
    - (Z,1) $\in$ path.

Computing in Prolog

- Ask a question (a query)

  ```
  ?- path(3,1).
  ```

- Prolog scans facts and rules top-to-bottom
  - 3 cannot be unified with 1, skip Rule 1.
  - Rule 2 generates two subgoals. Find `Z` such that
    - `(3,Z)` ∈ `edge` and
    - `(Z,1)` ∈ `path`.
- Sub goals are tried depth-first
  - `(3,Z)` ∈ `edge`?
    - `(3,4)` ∈ `edge`, set `Z = 4`

Computing in Prolog

- Ask a question (a query)

  `?- path(3,1).`

- Prolog scans facts and rules top-to-bottom
  - 3 cannot be unified with 1, skip Rule 1.
  - Rule 2 generates two subgoals. Find `Z` such that
    - `(3,Z)` $\in$ `edge` and
    - `(Z,1)` $\in$ `path`.
- Sub goals are tried depth-first
  - `(3,Z)` $\in$ `edge`?
    - `(3,4)` $\in$ `edge`, set `Z = 4`
  - `(4,1)` $\in$ `path`? 4 cannot be unifed with 1, two subgoals, new `Z'`
    - `(4,Z')` $\in$ `edge`
    - `(Z',1)` $\in$ `path`

Computing in Prolog

- Ask a question (a query)

  ```
  ?- path(3,1).
  ```

- Prolog scans facts and rules top-to-bottom
  - `3` cannot be unified with `1`, skip Rule 1.
  - Rule 2 generates two subgoals. Find `Z` such that
    - `(3,Z)` $\in$ `edge` and
    - `(Z,1)` $\in$ `path`.
- Sub goals are tried depth-first
  - `(3,Z)` $\in$ `edge`?
    - `(3,4)` $\in$ `edge`, set `Z = 4`
  - `(4,1)` $\in$ `path`? `4` cannot be unifed with 1, two subgoals, new `Z'`
    - `(4,Z')` $\in$ `edge`
    - `(Z',1)` $\in$ `path`
  - Cannot find `Z'` such that `(4,Z')` $\in$ `edge`!

Backtracking

- `(3,Z)` $\in$ `edge`?
  - `(3,4)` $\in$ `edge`, set `Z = 4`
- `(4,1)` $\in$ `path`? `4` cannot be unified with `1`, two subgoals, new `Z'`
  - `(4,Z')` $\in$ `edge`
  - `(Z',1)` $\in$ `path`
- No `Z'` such that `(4,Z')` $\in$ `edge`

Backtracking

- `(3,Z)` $\in$ `edge`?
  - `(3,4)` $\in$ `edge`, set `Z = 4`
- `(4,1)` $\in$ `path`? `4` cannot be unified with `1`, two subgoals, new `Z'`
  - `(4,Z')` $\in$ `edge`
  - `(Z',1)` $\in$ `path`
- No `Z'` such that `(4,Z')` $\in$ `edge`
- Backtrack and try another value for `Z`
  - `edge(3,5)` $\in$ `edge`, set `Z = 5`

Backtracking

- `(3,Z)` $\in$ `edge`?
  - `(3,4)` $\in$ `edge`, set `Z = 4`
- `(4,1)` $\in$ `path`? `4` cannot be unified with `1`, two subgoals, new `Z'`
  - `(4,Z')` $\in$ `edge`
  - `(Z',1)` $\in$ `path`
- No `Z'` such that `(4,Z')` $\in$ `edge`
- Backtrack and try another value for `Z`
  - `edge(3,5)` $\in$ `edge`, set `Z = 5`
- `(5,1)` $\in$ `path`? `(5,1)` $\in$ `edge`, `path(1,1)`, $\surd$

Backtracking

- `(3,Z)` $\in$ `edge`?
  - `(3,4)` $\in$ `edge`, set `Z = 4`
- `(4,1)` $\in$ `path`? `4` cannot be unified with `1`, two subgoals, new `Z'`
  - `(4,Z')` $\in$ `edge`
  - `(Z',1)` $\in$ `path`
- No `Z'` such that `(4,Z')` $\in$ `edge`
- Backtrack and try another value for `Z`
  - `edge(3,5)` $\in$ `edge`, set `Z = 5`
- `(5,1)` $\in$ `path`? `(5,1)` $\in$ `edge`, `path(1,1)`, $\checkmark$

Backtracking is sensitive to order of facts

- We had put `edge(3,4)` before `edge(3,5)`

Reversing the question

- Consider the question

```
?- edge(3,X).
```

Reversing the question

- Consider the question

```
?- edge(3,X).
```

- Find all `X` such that `(3,X)` $\in$ `edge`

Reversing the question

- Consider the question

```
?- edge(3,X).
```

- Find all `X` such that `(3,X)` $\in$ `edge`
- Prolog lists out all satisfying values, one by one

```
X=4;
X=5;
X=2;
No.
```