Sums and Variants
General Recursion
Lists

Advanced Compiler Construction and Program Analysis

Lecture 4

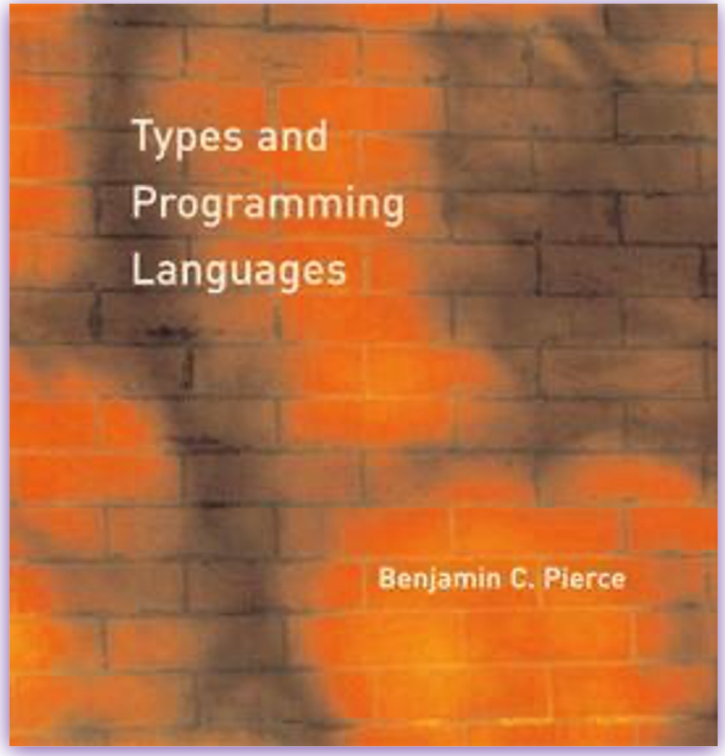
Innopolis University, Spring 2022

The topics of this lecture are covered in detail in…

Benjamin C. Pierce.

Types and Programming Languages

MIT Press 2002



Sum Types in Haskell

```haskell
data Result
     = Success Response
     | Failure ErrorMessage

processResult :: Result -> IO ()
processResult result =
  case result of
    Success response -> print response
    Failure err      -> print err
```

Sum Types in Haskell

```haskell
data Result
       = Success Response
       | Failure ErrorMessage
```

```
Result = Response + ErrorMessage

Success :: Response -> Result
Failure :: ErrorMessage -> Result
```

```haskell
processResult :: Result -> IO ()
processResult result =
  case result of
    Success response -> print response
    Failure err      -> print err
```

Sum Types: syntax

```
t ::= …                                  terms
    inl t                                tagging (left injection)
    inr t                                tagging (right injection)
    case t of inl x⇒t | inr x⇒t           case

v ::= …                                  values
    inl v                                tagged value (left)
    inr v                                tagged values (right)

T ::= …                                  types
      T + T                              sum type
```

Sum Types: evaluation

$$\texttt{case inl } v \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 \longrightarrow [x_1 \mapsto v]t_1$$

$$\texttt{case inr } v \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 \longrightarrow [x_2 \mapsto v]t_2$$

$$\frac{t \longrightarrow u}{\texttt{case } t \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 \longrightarrow \texttt{case } u \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2}$$

$$\frac{t \longrightarrow u}{\texttt{inl } t \longrightarrow \texttt{inl } u} \qquad \frac{t \longrightarrow u}{\texttt{inr } t \longrightarrow \texttt{inr } u}$$

Sum Types: typing

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{inl}\ t : A+B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \texttt{inr}\ t : A+B}$$

$$\frac{\Gamma \vdash t : A+B \qquad \Gamma, x_1 : A \vdash t_1 : C \qquad \Gamma, x_2 : B \vdash t_2 : C}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 : C}$$

Sum Types: example

Exercise 4.1. Evaluate the following term:

```
let
   toBool = λx:Nat+Bool.
      case x of
        inl n ⇒ if iszero n then false else true
      | inr b ⇒ b
in toBool (inl 3 as Nat+Bool)
```

Sums and Uniqueness of Types

What is the type of the term `inl 0` ?

Sums and Uniqueness of Types

What is the type of the term `inl 0` ?

```
inl 0 : Nat + Nat
inl 0 : Nat + Bool
inl 0 : Nat + (Nat → Bool)
```

Sums and Uniqueness of Types

What is the type of the term `inl 0` ?

```
inl 0 : Nat + Nat
inl 0 : Nat + Bool
inl 0 : Nat + (Nat → Bool)
```

Since types are not unique, type inference is not simple.
We can solve it in (at least) three ways:

1. Use a smarter type checking algorithm (infer from context).
2. Introduce some kind of polymorphism (e.g. use subtyping).
3. Ask the programmer to write down types explicitly.

Sum Types: syntax with explicit types

```
t ::= …
    inl t as T
    inr t as T
    case t of inl x⇒t | inr x⇒t

v ::= …
    inl v as T
    inr v as T

T ::= …
        T + T
```

Sum Types: typing with explicit types

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{inl}\ t\ \texttt{as}\ A+B : A+B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \texttt{inr}\ t\ \texttt{as}\ A+B : A+B}$$

$$\frac{\Gamma \vdash t : A+B \qquad \Gamma, x_1 : A \vdash t_1 : C \qquad \Gamma, x_2 : B \vdash t_2 : C}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 : C}$$

Variants: syntax with explicit types

t ::= …	***terms***
<l = t> as T	*tagging*
case t of <l_1=x_1>⇒t_1 | … | <l_n=x_n>⇒t_n	*case*
v ::= …	***values***
<l = v> as T	*tagged value*
T ::= …	***types***
<l_1:T_1, …, l_n=T_n>	*variant type*

Variants: evaluation

$$\texttt{case } \langle l = v \rangle \texttt{ as } T \texttt{ of } \ldots \mid \langle l = x_1 \rangle \Rightarrow t_1 \mid \ldots \longrightarrow [x_1 \mapsto v] t_1$$

$$\frac{t \longrightarrow u}{\texttt{case } t \texttt{ of } \langle l_1 = x_1 \rangle \Rightarrow t_1 \mid \ldots \mid \langle l_n = x_n \rangle \Rightarrow t_n \longrightarrow \texttt{case } u \texttt{ of } \langle l_1 = x_1 \rangle \Rightarrow t_1 \mid \ldots \mid \langle l_n = x_n \rangle \Rightarrow t_n}$$

$$\frac{t \longrightarrow u}{\langle l = t \rangle \texttt{ as } T \longrightarrow \langle l = u \rangle \texttt{ as } T}$$

Variants: typing

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash \langle l_i = t \rangle \text{ as } \langle l_1 : T_1, \ldots, l_n = T_n \rangle : \langle l_1 : T_1, \ldots, l_n = T_n \rangle}$$

$$\frac{\Gamma \vdash t : \langle l_1 : T_1, \ldots, l_n = T_n \rangle \quad \textit{for each } i \textit{ from 1 to } n \; \Gamma, x_i : T_i \vdash t_i : C}{\Gamma \vdash \text{case } t \text{ of } \langle l_1 = x_1 \rangle \Rightarrow t_1 \mid \ldots \mid \langle l_n = x_n \rangle \Rightarrow t_n : C}$$

Option type

```
Option[A]  := <just : A, nothing : Unit>


let
   positive = λx:Nat.
      if iszero x
        then <nothing = unit> as Option[Nat]
        else <just = x> as Option[Nat]
in positive (succ (succ (succ 0)))
```

Enumerations

```
Color := <red : Unit, green : Unit, blue :
Unit>

let
   colorCode = λx:Color.
      case x of
        <red   = _> ⇒ 1
      | <green = _> ⇒ 2
      | <blue  = _> ⇒ 3
in colorCode (<green = unit> as Color)
```

Wrapper types as single-field variants

```
Seconds := <seconds : Nat>
Minutes := <minutes : Nat>

let
   minutesToSeconds = λx:Minutes.
      case x of
        <minutes = n>
           ⇒ <seconds = 60 * n> as Seconds
in minutesToSeconds (<minutes = 15> as Minutes)
```

General Recursion

t ::= …	***terms***
fix t	*fixed point*

$$\textbf{fix}\ (\lambda x{:}T.t) \longrightarrow [x \mapsto \textbf{fix}\ (\lambda x{:}T.t)]t$$

$$\frac{t \longrightarrow u}{\textbf{fix}\ t \longrightarrow \textbf{fix}\ u}$$

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \textbf{fix}\ t : T}$$

General Recursion: `letrec` derived form

$$\texttt{letrec } x = t_1 \texttt{ in } t_2 \quad := \quad \texttt{let } x = \texttt{fix } (\lambda x{:}T.t_1) \texttt{ in } t_2$$

General Recursion: exercise

Exercise 4.2. Define `equals` using `fix`.

General Recursion: exercise

Exercise 4.3. Define `plus`, `times`, `factorial` using `fix`.

General Recursion: every type is inhabited

```
diverge^T := λ_:Unit. fix (λx:T.x)
```

General Recursion: every type is inhabited

```
diverge[T] := λ_:Unit. fix (λx:T.x)

undefined[T] := diverge[T] unit
```

Lists: syntax

```
t ::= …                          terms
    nil[T]                       empty list
    cons[T] t t                  list constructor
    isnil[T] t                   empty list test
    head[T] t                    head of a list
    tail[T] t                    tail of a list

v ::= …                          values
    nil[T]                       empty list value
    cons[T] v v                  list value constructor

T ::= …                          types
       List[T]                   list type
```

Lists: evaluation (computation rules)

$$\textbf{isnil}[T_1]\ \textbf{nil}[T_2] \longrightarrow \texttt{true}$$

$$\textbf{isnil}[T_1]\ (\textbf{cons}[T_2]\ v_1\ v_2) \longrightarrow \texttt{false}$$

$$\textbf{head}[T_1]\ (\textbf{cons}[T_2]\ v_1\ v_2) \longrightarrow v_1$$

$$\textbf{tail}[T_1]\ (\textbf{cons}[T_2]\ v_1\ v_2) \longrightarrow v_2$$

Lists: evaluation (congruence rules)

$$\frac{t \longrightarrow u}{\texttt{isnil[T]}\ t \longrightarrow \texttt{isnil[T]}\ u}$$

$$\frac{t_1 \longrightarrow u_1}{\texttt{cons[T]}\ t_1\ t_2 \longrightarrow \texttt{cons[T]}\ u_1\ t_2}$$

$$\frac{t_2 \longrightarrow u_2}{\texttt{cons[T]}\ v_1\ t_2 \longrightarrow \texttt{cons[T]}\ v_1\ t_2}$$

$$\frac{t \longrightarrow u}{\texttt{head[T]}\ t \longrightarrow \texttt{head[T]}\ u}$$

$$\frac{t \longrightarrow u}{\texttt{tail[T]}\ t \longrightarrow \texttt{tail[T]}\ u}$$

Lists: typing

$$\Gamma \vdash \texttt{nil[T]} : \texttt{List[T]}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : \texttt{List[T]}}{\Gamma \vdash \texttt{cons[T]}\ t_1\ t_2 : \texttt{List[T]}}$$

$$\frac{\Gamma \vdash t : \texttt{List[T]}}{\Gamma \vdash \texttt{isnil[T]}\ t : \texttt{Bool}}$$

$$\frac{\Gamma \vdash t : \texttt{List[T]}}{\Gamma \vdash \texttt{head[T]}\ t : T}$$

$$\frac{\Gamma \vdash t : \texttt{List[T]}}{\Gamma \vdash \texttt{tail[T]}\ t : \texttt{List[T]}}$$

Lists: example

```
letrec
   length = λx:List[Nat].
      if isnil[Nat] x
        then 0
        else succ (length (tail[T] x))
in let
   list = cons[Nat] 0 (cons[Nat] 1 nil[Nat])
in length list
```

Lists: type annotations

Exercise 4.4.
Some type annotations in the syntax of lists can be removed. Is it possible to remove all type annotations?

Lists: progress and preservation

Exercise 4.5.
Verify that progress and preservation still hold for simply typed lambda calculus with general recursion and lists.

Lists: exercise

Exercise 4.6.
Implement a `map` and `fold` for lists.

Lists: exercise

Exercise 4.7.
Reformulate lists, using **`case`** instead of **`head`** and **`tail`**.

Summary

- ❏ Sum types and Variants
- ❏ General Recursion
- ❏ Lists

Summary

- ❏ Sum types and Variants
- ❏ General Recursion
- ❏ Lists

See you next time!