Odds and Ends

Principles of Programming Languages

CAS CS 320

Lecture 20

Practice Problem

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'}$$

$$\cdot \vdash \lambda f^{\top \to \top} . \lambda x^{\top} . fx : (\top \to \top) \to \top \to \top$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

Give a derivation for the above judgment.

Answer

$$\cdot \vdash \lambda f^{\top \to \top} . \lambda x^{\top} . fx : (\top \to \top) \to \top \to \top$$

Outline

Demo an implementation of the STLC

Briefly discuss concrete syntax and desugaring

Demo an implementation of desugaring

Bonus Topic: Parametric polymorphism and System F

Demo (I)

Demo (Syntax)

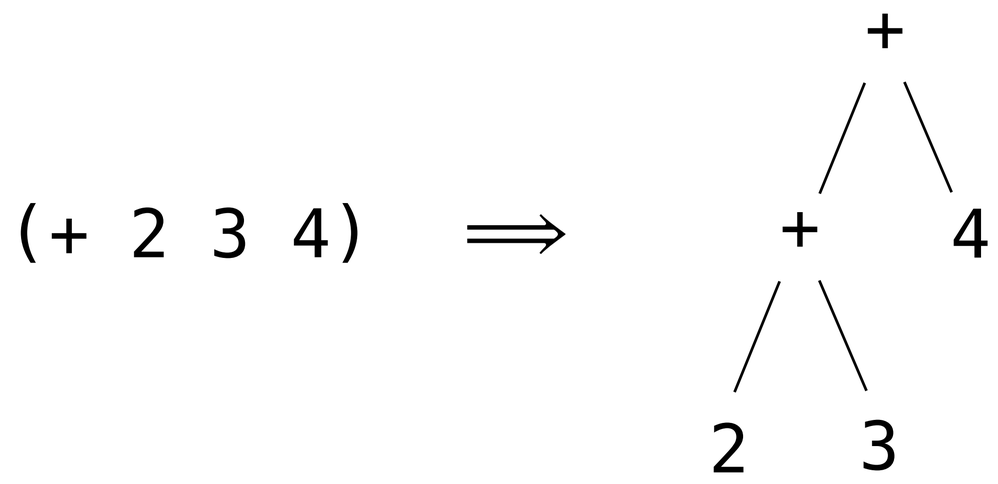
```
<e>  ::= () | <v> | <e> <e> | fun ( <v> : <ty> ) -> <e>
       | let <v> : <ty> = <e> in <e>
       | let rec <v> ( <v> : <ty> ) : <ty> = <e> in <e>
       | if <e> then <e> else <e>
       | <e> + <e> | <e> - <e> | <e> * <e> | <e> = <e>
<ty> ::= unit | int | bool | <ty> -> <ty>
<v>  ::= ...
```

This is an extension of our demo from last lecture

(It would be good practice to write down the typing rules for this language)

Desugaring

Surface-Level vs. Abstract Syntax



Surface Level (Concrete) syntax refers to how a program looks to a human programmer. It's usually described by a BNF grammar

Abstract syntax refers to how a program look to a type-checker and evaluator, it's what actually gets *run*

Both can be represented in code. Abstract syntax is almost always represented as an ADT in OCaml.

Desugaring

```
fun x y -> x + y
```

becomes

```
fun x -> fun y -> x + y
```

Desugaring is the process of transforming surface-level syntax into abstract syntax

This is **not** a terribly interesting aspect of PL design, but it makes our languages more pleasant to use

The Project

We will do three kinds of translations in mini-project 2:

1. **top-level lets**

2. **lets with arguments**

3. **multi-argument functions**

These three things will give us a surface level syntax very close to what we're used to in OCaml

Demo (II)

System F (Bonus Topic)

Parametric Polymorphism

```
let k = fun x y -> x

let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

In OCaml we can write **polymorphic** functions which are *agnostic* in some way to their input type

They are *parametrized* by the type of the input, in that they must behave exactly the same way on all inputs, independent of the type itself.

In particular, we can't "dispatch" according to type

Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The parameters in our types are called **type variables**

The variables are *instantiated* when we apply our function to an argument

<u>The implications:</u>

» We need to define **substitution** for types
» We need to determine *when* to substitute

Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

Just like with expression variables, we don't like *unbound* type variables (if they're unbound it means we don't know what they refer to)

In reality, types variables in OCaml are *quantified* (and the syntax is hidden)

We read this "`id` has type `t -> t` for any type `t`"

(This means we might have to deal with capture-avoiding substitution in types!)

Second-Order Lambda Calculus

```
let id_int : int -> int = fun x -> x
let id : 'a . 'a -> 'a =
  fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

System F or the **second-order lambda calculus** were introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

As usual the motivations for introducing this systems were quite different from our ideas about polymorphism now

The basic idea: One way of allowing polymorphism is to allow the introduction of types into the language which we can *pass as arguments to functions*

Warning

```
let id_int : int -> int = fun x -> x
let id : 'a . 'a -> 'a =
  fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

Not valid OCaml

This is *not* what OCaml does

This is *not* what we'll be implementing in the project

There are *very few* languages that implement this kind of polymorphism

This is just to see another way to deal with polymorphism *(and for your general intellectual betterment)*

The Syntax

$$e ::= \bullet \mid x \mid \lambda x^{\tau} . e \mid ee \mid \Lambda\alpha . e \mid e\tau$$

$$\tau ::= \top \mid \tau \to \tau \mid \alpha \mid \forall\alpha . \tau$$

$$x ::= \textit{variables}$$

$$\alpha ::= \textit{type variables}$$

The syntax for SOLC is the same as the that of STLC but with

» constructs for abstracting over and applying to types

» constructs for quantifying (or generalizing) over type variables

Typing

$$\frac{}{\Gamma \vdash \bullet : \top} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad (*)}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . \tau} \qquad \frac{\Gamma \vdash e : \forall \alpha . \tau \qquad \tau' \texttt{ is a type}}{\Gamma \vdash e\tau' : [\tau'/\alpha]\tau}$$

*** α must be not appear free in Γ**

We add <u>two new rules</u> to STLC to deal with our new constructs for polymorphism:

1. We can generalize over a type variable if our context doesn't depend on it
2. We can apply an expression e to a type τ, but we have to *substitute* the type into the type of e

Example

$$\cdot \vdash (\Lambda\alpha\,.\,\lambda x^{\alpha} x)(\ \top \to \top\ )\lambda x^{\top}\,.\,x : \top \to \top$$