CS6202: Advanced Topics in Programming Languages and Systems

Lecture 7 : **Universal/Existential Types**

- Motivation for Universal Types
- System F & Examples
- Properties
- Type Reconstruction & Parametricity
- Existential Types

Motivation for Universal Types

Lack of code reuse!

Example:

doubleNat = λ f: Nat $\rightarrow$ Nat. λ x:Nat. f (f (x))

doubleBool = λ f: Bool $\rightarrow$ Bool. λ x:Bool. f (f (x))

doubleFun = λ f: (Nat $\rightarrow$ Nat) $\rightarrow$ (Nat $\rightarrow$ Nat).
λ x: Nat $\rightarrow$ Nat. f (f (x))

Properties of Let Polymorphism

- allows for easy type reconstruction
- restricted to the let construct
- problems with references
  (restricted to values on the RHS of =).

Idea of Universal Types

Abstract over type!

double = λ X. λ f: X $\rightarrow$ X. λ x:X. f (f (x))
> *double : $\forall$X. (X $\rightarrow$ X) $\rightarrow$ X $\rightarrow$ X*

double [Nat]
> *<fun> : (Nat $\rightarrow$ Nat) $\rightarrow$ Nat $\rightarrow$ Nat*

double [Bool]
> *<fun> : (Bool $\rightarrow$ Bool) $\rightarrow$ Bool $\rightarrow$ Bool*

Evaluation (Type Abstraction/Application)

$$\frac{t_1 \to t_1'}{t_1[T_2] \to t_1'\,[T_2]} \qquad \text{(E-TApp)}$$

$$(\lambda\, X.\ t)\ [T] \to [X \mapsto T]\ t \qquad \text{(E-TAppTAbs)}$$

Typing (Type Abstraction/Application)

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \lambda\, X\,.\,t : \forall\, X.\ T} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall\, X.\ T_1}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]\ T_1} \qquad \text{(T-TApp)}$$

Example : Identity Function

Creating the polymorphic identity function:

id = λ X. λ x:X. x
> *id : ∀X. X → X*

Using the polymorphic identity function:

id [Nat]
> *<fun> : Nat → Nat*

id [Bool] true
> *true : Bool*

Example : Self-Application

We can apply a function to itself (in System F). Define

selfApp = λ x: ∀ X. X → X . (x [∀ X. X → X]) x
> *selfApp : (∀X. X → X) → (∀X. X → X)*

One use of self-application is:

quadruple = selfApp double
> *quadruple : ∀X. (X → X) → (X → X)*

Exercise : show that above is the same as:
quadruple = λ X. double [X → X] (double [X])

Example : Polymorphic List

Assume that type constructor List is given with the following primitives:

nil	:	$\forall$ X. List X
cons	:	$\forall$ X. X $\rightarrow$ List X $\rightarrow$ List X
isnil	:	$\forall$ X. List X $\rightarrow$ Bool
head	:	$\forall$ X. List X $\rightarrow$ X
tail	:	$\forall$ X. List X $\rightarrow$ List X

Example : Map

With the help of fix, we can write a polymorphic map operation, as follows:

```
map = λ X. λ Y. λ f: X → Y.
            fix (λ m: (List X) → (List Y).
               λ l:List X.
                  if isnil [X] l then nil [Y]
                  else cons [Y] (f (head [X] l))
                                      (m (tail [X] l)))  )
```

> *map : ∀X. ∀Y. (X → Y) → List X → List Y*

System F : Soundness Properties

Preservation Theorem:

If $\Gamma \vdash t : T$ and $t \rightarrow t'$
then $\Gamma \vdash t' : T$

Progress Theorem

If t is a closed well-typed term, then either t is a value or else there is some t' with $t \rightarrow t'$.

System F : Normalisation

A term is normalizing if there is no infinite evaluation

$$t \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$$

Well-typed System F terms (without the fix-point operator) are normalizing.

System F : Historical Background

- Discovered by Jean-Yves Girard in 1972 for proof theory.
- Independently developed by John Reynolds 1974 as *polymorphic lambda calculus*.
- Normalization : quite innovative inductive proof technique due to Tait (1968) and Girard (1972).
- Type reconstruction : was open problem until 1994!

Undecidability of Type Reconstruction for System F.

Wells 1994 : it is *undecidable* when given a closed term m of the untyped lambda calculus, if there is some well-typed term t in System F such that erase(t)=m.

The type erasure operation is defined as:

$\text{erase}(x)$	$= x$
$\text{erase}(\lambda\, x{:}T.\ t)$	$= \lambda\, x.\ \text{erase}(t)$
$\text{erase}(t_1\ t_2)$	$= \text{erase}(t_1)\ \text{erase}(t_2)$
$\text{erase}(\lambda\, x{:}X.\ t)$	$= \text{erase}(t)$
$\text{erase}(t\ [T]\ )$	$= \text{erase}(t)$

What to do with Undecidability?

Restrict the language :

let polymorphism of ML, rank-2 polymorphism, etc.

Partial Type Reconstruction:

correct but incomplete approaches such as local type inference, greedy type inference, etc.

Parametricity

Polymorphic programs operate uniformly over any input, independently of their type.

Language implementations benefit from this *parametricity* by generating only one machine code version for polymorphic functions. Also, certain theorems come for free.

At runtime, type application does not result in any computation. This is exemplified by OCaml’s let polymorphism, where no type application is needed.

Motivation for Existential Type

We emphasize the operational reading, supported by the notation:

$\{\exists X, T\}$

Terms of such type have the form:

$\{*S, t\}$

We call such terms "modules" with the *hidden* type S and the term component t.

Example

The term

$\{*Nat, \{a=5, f=\lambda\ x{:}Nat.\ succ(x)\}\}$

has type

$\{\exists X, \{a{:}X, f{:}X \to X\}$

but it may also have type:

$\{\exists X, \{a{:}X, f{:}X \to Nat\}$

Solution : use ascription to force a unique type for module.

$\{*Nat, \{a=5, f=\lambda\ x{:}Nat.\ succ(x)\}\}$ as $\{\exists X, \{a{:}X, f{:}X \to X\}$

Elements of Existential Types

The hidden type of different elements can be different.

p4 = $\{*Nat, \{a=5, f=\lambda\ x{:}Nat.\ succ(x)\}\}$ as $\{\exists X, \{a{:}X, f{:}X \to Nat\}$
> *p4 : $\{\exists X, \{a{:}X, f{:}X \to Nat\}$*

p5 = $\{*Bool, \{a=true, f=\lambda\ x{:}Bool.\ 0\}\}$ as $\{\exists X, \{a{:}X, f{:}X \to Nat\}$
> *p5 : $\{\exists X, \{a{:}X, f{:}X \to Nat\}$*

In effect, the module type is *parameterised* over the internal type. Elements of existential types use internal types, but these are not visible where the elements are used.

Violations of Abstraction

We must not make assumption about internal type, nor could it be exposed to a location out of its scope.

let {X,x}=p4 in succ(x.a)
> *Error : argument of succ is not a number.*

let {X,x}=p4 in x.a
> *Error : scoping error!*

where:

p4 = $\{*Nat, \{a=5, f=\lambda\ x{:}Nat.\ succ(x)\}\}$ as $\{\exists X, \{a{:}X, f{:}X \to Nat\}$

Syntax of Existential Types

• t ::=	…	terms
	{*T,t} as T	packing
	let {X,x}=t in t	unpacking
• v ::=	…	values
	{*T,v) as T	package value
• T ::=	…	values
	{∃X,T}	existential type

Typing Rules

$$\frac{\Gamma \vdash t : [X \mapsto U]\, T}{\Gamma \vdash \{{*}U,\, t\} \text{ as } \{\exists\, X,\, T\} : \{\exists\, X,\, T\}} \quad \text{(T-Pack)}$$

$$\frac{\Gamma \vdash t_1 : \{\exists\, X,\, T_1\} \qquad \Gamma, X, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X,x\}{=}t_1 \text{ in } t_2 : T_2} \quad \text{(T-Unpack)}$$

Evaluation Rules

$$\frac{t \to t'}{\{{*}U,\, t\} \text{ as } T \to \{{*}U,\, t'\} \text{ as } T} \quad \text{(E-Pack)}$$

$$\frac{t_1 \to t_1'}{\text{let } \{X,x\}{=}t_1 \text{ in } t_2 \to \text{let } \{X,x\}{=}t_1' \text{ in } t_2} \quad \text{(E-UnPack)}$$

$$\text{let } \{X,x\}{=}\, \{{*}T,\, v\} \text{ in } t \to [X \mapsto T,\, x \mapsto v]\, t \quad \text{(E-UnpackPack)}$$

Abstract Data Types

ADT counter =
- type Counter
- representation Nat
- signature
  - new : Counter,
  - get : Counter → Nat,
  - inc : Counter → Counter;
- operations
  - new = 1;
  - get = λ i:Nat. i
  - inc = λ i:Nat. succ(i)

Translation using Existential Types

```
counterADT =
      {*Nat,
            {new = 1,
              get = λ i:Nat. i
              inc = λ i:Nat. succ(i) } }
      as {∃ Counter,
            {new : Counter,
              get : Counter → Nat,
              inc : Counter → Counter} }
```

> *counterADT : {∃Counter, {new : Counter,*
> *get : Counter → Counter, inc : Counter → Counter}}*

Using Abstract Data Types

```
let {Counter,ctr} = counterADT in
ctr.get (ctr.inc ctr.new)
```

> *2 : Nat*

```
let {Counter,ctr} = counterADT in
let add3 = λ c:Counter.
             ctr.inc(ctr.inc(ctr.inc c)) in ctr.get(add3 ctr.new)
```

> *4 : Nat*

Structure of Programs using ADTs

Each ADT can use all previously declared ADTs.

```
let {ADT,m1} = <ADT1 package> in

let {ADT,m2} = <ADT2 package> in

      …

let {ADT,mn} = <ADTn package> in

      <main program>
```


Representation Independence

Abstract data type enjoy *representation independence*. They can be replaced by alternative implementations without affecting the rest of the programs, as long as the existential type is not modified. Example:

```
counterADT =
      {* {x:Nat},
        {new = {x=1},
          get = λ i: {x:Nat}. i.x
          inc = λ i: {x:Nat}. {x=succ(i.x)} } }
      as {∃ Counter,
        {new : Counter,
          get : Counter → Nat,
          inc : Counter → Counter} }
```

Motivation for Bounded Quantification

Arises when subtyping is combined with polymorphism.

Consider:

f = λ x: {a:Nat}. x
> *f : {a:Nat} → {a:Nat}*

Now, what is the type of?

f {a=0}
> *{a=0} : {a:Nat}*

f {a=1, b=4}
> *{a=1,b=4} : {a:Nat}*

Problem

Note that below is ill-typed! Why?

let c = f {a=1, b=4} in
c.b

One solution is to use universal type:

f = λ X . λ x: X. x
> *f : ∀X. X → X*

But how to handle:

f = λ x: {a:Nat}. {x.a, x}
> *f : {a:Nat} → {a:Nat}*

Certainly not !

f = λ X . λ x: X. {x.a, x}

Solution : Bounded Quantification

Quantified type may be *bounded* by a subtyping relation:

For example:

f = λ X<:{a:Nat} . λ x: X. {x.a, x}
> *f : ∀X <:{a:Nat} . X → {Nat, X}*

This is the core of System $F_{<:}$. More details in Pierce's book!