Design Principles of Programming Languages

Recursive Types

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong
Peking University, Spring Term

Review: what have we learned so far?

- λ-calculus: function and data can be treated the same
- Types: annotations for preventing bugs
  - All terms can be typed: functions, statements, etc.
  - Safety=Progress+Preservation
- Structural types: can we do better than Java?
- Subtypes: what if a term has more than one type?

What in the latter half of the course?

- Recursive types
  - from finite world to infinite world
  - theory of induction and coinduction
- Type Inference
- Polymorphism
  - theoretical base for generics
  - System F: an important system for academic study

- Do come to class
  - Will be much harder than the first half!
  - The book is not perfect.
  - Class performance will be part of your final score

Defining a linked list

- Implementing in Java

```
class ListNode {
  int value;
  ListNode next;
}
```

- Implementing in fullSimple
  - `NatList = <nil:Unit, cons:{Nat,NatList}>;`
  - `nil = <nil=unit> as NatList;`
  - `cons = lambda n:Nat. lambda l:NatList. <cons={n,l}> as NatList;`

Compiling

- natlist.f

```
NatList = <nil:Unit, cons:{Nat,NatList}>;
nil = <nil=unit> as NatList;
cons = lambda n:Nat. lambda l:NatList.
<cons={n,l}> as NatList;
```

```
/cygdrive/d/Kuaipan/Courses/2014 Design Principles of Programming Languag...

Yingfei@Yingfei-Laptop /cygdrive/d/Kuaipan/Courses/2014 Design Principles of P
rogramming Languages/src/fullsimple
$ ./f natlist.f
NatList :: *
nil : NatList
D:\Kuaipan\Courses\2014 Design Principles of Programming Languages\src\fullsimple\
natlist.f:3.46:
field does not have expected type
```

Why?

- Source of Parser.mly

```
AType :
  …
  | UCID
       { fun ctx ->
           if isnamebound ctx $1.v then
             TyVar(name2index $1.i ctx $1.v, ctxlength ctx)
           else
             TyId($1.v) }
  …
```

- Second NatList is parsed as a new TyId
  - `NatList = <nil:Unit, cons:{Nat,NatList}>;`

Recursive Types

- Useful in defining complex types
- Need special mechanism to support

- This course is about
  - How useful recursive types are
  - How to support recursive types

Defining Recursive Types

- Using operator μ
  - `NatList = μX. <nil:Unit, cons:{Nat,X}>`
  - Meaning: `X = <nil:Unit, cons:{Nat,X}>.`

- Constructors of NatList

```
nil = <nil=unit> as NatList;
▸ nil : NatList

cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList;
▸ cons : Nat → NatList → NatList
```

NatList Functions

```
isnil = λl:NatList. case l of
                        <nil=u> ⇒ true
                      | <cons=p> ⇒ false;
```

▸ isnil : NatList → Bool

```
hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1;
```

▸ hd : NatList → Nat

```
tl = λl:NatList. case l of <nil=u> ⇒ l | <cons=p> ⇒ p.2;
```

▸ tl : NatList → NatList

Can we define an infinite list in NatList?

- 1, 2, 1, 2, 1, 2, 1, 2, …
- `infList = fix (λf. cons 1 (cons 2 f))`
- `hd (tl (tl infList))` //get the 3rd element
- Unfortunately, will diverge
  - why?

Review: Reduction Order (page57)

- Full beta-reduction
  - any redex may be reduced at any time
- Normal Order
  - leftmost, outmost redex is reduced first
- Call by name (used in lazy evaluation languages)
  - Normal Order + No reduction inside abstractions
- Call by value (used in the book)
  - Call by name + Parameters need to be values
- `infList = fix (λf. cons 1 (cons 2 f))`
- `hd (tl (tl infList))` //get the 3rd element

Interlude: Why do we need infinite lists?

- Computers can only perform finite computations
- Answer
  - Because we can
  - Because it is cool
  - Because it could be more structural and reusable
- Example: find the largest i where ith element in Fibonacci sequence is smaller than C

Java version:

```
int index = 0, v1=0, v2=1;
while (v1 < C) {
  int t = v1+v2;
  v1=v2;
  v2=t;
  index++;
}
return index;
```

Haskell version:

```
fib = 0 : scanl (+) 1 fib
length takeWhile (< C) fib
```

Recursive Functional Types

- What is this function type about?

```
Stream = μA. Unit→{Nat,A};
```

- Returning elements in an infinite sequence one by one
  - Continuation
- Java counterpart: iterator
  - With a mutable state

A Fibonacci stream

```
Stream = μX. Unit->{Nat, X};

fibonacci =
  let fib = fix (λf:Nat->Nat->Stream.
      λx:Nat. λy:Nat.
      λ_:Unit. {x, f y (plus x y)})
  in
  fib 0 1;
```

- Why not diverge?

Exercies

- Use the idea of Stream to fix `infList`
- Two functions “nil” and “cons” for list constructions
- Two functions “hd” and “tl” for returning elements
- Construct the following two lists in your implementation
  - 01
  - 1212121212…
- And return the second element
- Implement in fullequirec

Answer

- `InfList = Rec X. Unit-><infNil:Unit, infCons:{Nat,X}>;`
- `InfBody = <infNil:Unit, infCons:{Nat,InfList}>;`
- `nil = lambda _:Unit. <infNil=unit> as InfBody;`
- `cons = lambda n:Nat. lambda l:InfList. lambda _:Unit. <infCons={n,l}> as InfBody;`

- `zeroOneList = cons 0 (cons 1 nil);`
- `oneTwoList = fix (lambda l:InfList. cons 1 (cons 2 (lambda _:Unit. l unit)));`

Hungry Function

- Stupid yet simple function. Will be used to discuss the properties of recursive types.

  - `Hungry = `μ`A. Nat→A;`
  - `f = fix (`λ`f: Hungry. `λ`n:Nat. f);`

Representing Objects

- Can we represent the following immutable counter?

```
class Counter {
  int get();
  Counter inc();
  Counter dec();
}
```

- Without recursive type:
  - $\text{Counter} = \{\text{get}: \text{Unit} \longrightarrow \text{Nat}, \ \text{inc}: \text{Unit} \longrightarrow \text{Counter}, \text{dec}: \text{Unit} \rightarrow \text{Counter}\}$

Functional Objects

```
Counter = μC. {get:Nat, inc:Unit→C, dec:Unit→C};

  c = let create = fix (λf: {x:Nat}→Counter. λs: {x:Nat}.
                                  {get = s.x,
                                   inc = λ_:Unit. f {x=succ(s.x)},
                                   dec = λ_:Unit. f {x=pred(s.x)} })
      in create {x=0};

► c : Counter

  c1 = c.inc unit;
  c2 = c1.inc unit;
  c2.get;

► 2 : Nat
```

Review: fixed-point combinator

- Law: fix f = f (fix f)
- Y Combinator

$$Y = \lambda f.\ (\lambda x.\ f\ \overbrace{(x\ x)}^{\text{(fix f)}})\ (\lambda x.\ f\ (x\ x))$$

- Use of Y Combinator: calculating $\Sigma_{i=0}^{n} i$

```
f = λf. λn.
  if (iszero n) then 0
  else n + f (n – 1)
Y f
```

Review: fixed-point combinator

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

- Why fix is used instead of Y?

Answer

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

Y = λf. (λx. f (x x)) (λx. f (x x))

- Under full beta-reduction: Let f : $T \to T$
  - When T is a function type
    - Fix and Y are equal: $(\lambda y\ (x\ x)\ y)\ v = (x\ x)\ v = (fix\ f)\ v$
  - Else
    - (Fix f) will stuck, while (Y f) will diverage
- Not under call-by-value because
  - `(x x)` is not a value
  - while `(λy. x x y)` is
  - Y will diverge for any f

Review: fixed-point combinator

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

- Can we define Y in simple typed λ-calculus?
  - No
  - x has a recursive type
  - Y was defined as a special language primitive

Defining `fix` using recursive types

Y_T = λf:T→T. (λx:(μA.A→T). f (x x)) (λx:(μA.A→T). f (x x))

Y_T : (T→T) → T

- T is the type of the recursive function
- Q: Do languages with recursive types have strong normalization property?
  - Strong normalization: well-typed program will terminate
- A: No, because Y_T can be defined

Defining Lambda Calculus

- Read the book

Implementation Problem 1

- `Hungry = μA. Nat→A;`
- `h = fix (λf: Nat→ Hungry. λn:Nat. f);`

- What is the type of h?
  - `Hungry?`
  - `Nat→Hungry?`
  - `Nat→Nat→Hungry?`

Simple but Effective Solution

- Every term has one type
- Use fold/unfold to convert between types
- `h = fix (λf: Nat→ Hungry. λn:Nat. f)`
  - `h: Nat→ Hungry`
  - `fold[Hungry] h: Hungry`
  - `unfold[Hungry] (h 1): Nat→ Hungry`

Iso-recursive Types

$\rightarrow \mu$ *Extends $\lambda_{\rightarrow}$ (9-1)*

$\mathtt{t}$	::=	...	*terms:*
		$\mathtt{fold}\ [\mathtt{T}]\ \mathtt{t}$	*folding*
		$\mathtt{unfold}\ [\mathtt{T}]\ \mathtt{t}$	*unfolding*
$\mathtt{v}$	::=	...	*values:*
		$\mathtt{fold}\ [\mathtt{T}]\ \mathtt{v}$	*folding*
$\mathtt{T}$	::=	...	*types:*
		$\mathtt{X}$	*type variable*
		$\mu\mathtt{X}.\mathtt{T}$	*recursive type*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{unfold}\ [\mathtt{S}]\ (\mathtt{fold}\ [\mathtt{T}]\ \mathtt{v}_1) \longrightarrow \mathtt{v}_1 \qquad \text{(E-UNFLDFLD)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{fold}\ [\mathtt{T}]\ \mathtt{t}_1 \longrightarrow \mathtt{fold}\ [\mathtt{T}]\ \mathtt{t}_1'} \qquad \text{(E-FLD)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{unfold}\ [\mathtt{T}]\ \mathtt{t}_1 \longrightarrow \mathtt{unfold}\ [\mathtt{T}]\ \mathtt{t}_1'} \qquad \text{(E-UNFLD)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{U} = \mu\mathtt{X}.\mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_1 : [\mathtt{X} \mapsto \mathtt{U}]\mathtt{T}_1}{\Gamma \vdash \mathtt{fold}\ [\mathtt{U}]\ \mathtt{t}_1 : \mathtt{U}} \qquad \text{(T-FLD)}$$

$$\frac{\mathtt{U} = \mu\mathtt{X}.\mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{U}}{\Gamma \vdash \mathtt{unfold}\ [\mathtt{U}]\ \mathtt{t}_1 : [\mathtt{X} \mapsto \mathtt{U}]\mathtt{T}_1} \qquad \text{(T-UNFLD)}$$

Figure 20-1: Iso-recursive types ($\lambda\mu$)

Exercise

- Implement (finite) NatList in iso-recursive type
  - implement nil, cons, hd

Example

- `NatList = μX. <nil:Unit, cons:{Nat,X}>`
- `NLBody = <nil:Unit, cons:{Nat,NatList}>`
- `nil = fold [NatList](<nil=unit> as NLBody);`
- `cons = λn:Nat. λl:NatList. fold[NatList] <cons={n,l}> as NLBody`

Example

```
isnil = λl:NatList.
          case unfold [NatList] l of
            <nil=u> ⇒ true
          | <cons=p> ⇒ false;
hd = λl:NatList.
        case unfold [NatList] l of
          <nil=u> ⇒ 0
        | <cons=p> ⇒ p.1;
tl = λl:NatList.
        case unfold [NatList] l of
          <nil=u> ⇒ l
        | <cons=p> ⇒ p.2;
```

Implementation Problem 2

- `Even <: Nat`
- $\texttt{A = } \mu\texttt{X.Nat} \rightarrow \texttt{(Even} \times \texttt{X)}$
- $B = \mu Y.\mathrm{Even} \rightarrow (\mathrm{Nat} \times Y)$

- What is the subtype relation between A and B?
  - A <: B?
  - B <: A?
  - No relation?

Subtyping by assumption

- $\dfrac{\Sigma, X <: Y \vdash S <: T}{\Sigma \vdash \mu X.S <: \mu Y.T}$

- Example:
  - `Even <: Nat`
  - A = μX.Nat→(Even×X)
  - B = μY.Even→(Nat×Y)

  - Assuming X<:Y
  - We have Nat→(Even×X) <: Even→(Nat×Y)
  - Thus A <: B
- Why this works? Principle of safe substitution.
- Its implementing algorithm will be explained in the next course

Recursive Types in Practice

- Recursive data types
  - Most language supports recursive data types by nominal type system
    - Java, C#, …
  - Some languages with structural types try to generate fold/unfold
    - Haskell, OCaml…
- Recursive function types
  - C# supports recursive function types through nominal types
    - "delegate int A()" and "delegate int B()" are different

Homework

- Implement Y combinator in your favorite language except Ocaml
  - Your implementation will be limited by the expressiveness of the language, but should support (fix f) where f:(Nat->Nat)->(Nat->Nat)
  - Your implementation should contain test cases for the teaching assistants to easily verify your implementation
  - Hint: wrap functions in data types, like Java interface
  - Please submit electronically