Recursive Types

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong
Peking University, Spring Term, 2015
Review: what have we learned so far?

• \(\lambda\)-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
  
  ```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;
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 $\mu$
  • $\text{NatList} = \mu X. \langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, X\}\rangle$
  • Meaning: $X = \langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, X\}\rangle$.

• Constructors of $\text{NatList}$
  
  \[
  \text{nil} = \langle \text{nil}=\text{unit}\rangle \text{ as } \text{NatList};
  \]

  \[\text{cons} = \lambda n: \text{Nat}. \lambda l: \text{NatList}. \langle \text{cons} = \{n, l\}\rangle \text{ as } \text{NatList};\]

  \[\text{cons} : \text{Nat} \rightarrow \text{NatList} \rightarrow \text{NatList}\]
NatList Functions

\[
\text{isnil} = \lambda l : \text{NatList}. \ \text{case } l \ \text{of}
\]
\[
\begin{align*}
\text{<nil}=u & \Rightarrow \text{true} \\
\mid \text{<cons}=p & \Rightarrow \text{false};
\end{align*}
\]

\text{- isnil} : \text{NatList} \rightarrow \text{Bool}

\[
\text{hd} = \lambda l : \text{NatList}. \ \text{case } l \ \text{of}
\]
\[
\begin{align*}
\text{<nil}=u & \Rightarrow 0 \\
\mid \text{<cons}=p & \Rightarrow p.1;
\end{align*}
\]

\text{- hd} : \text{NatList} \rightarrow \text{Nat}

\[
\text{tl} = \lambda l : \text{NatList}. \ \text{case } l \ \text{of}
\]
\[
\begin{align*}
\text{<nil}=u & \Rightarrow 1 \\
\mid \text{<cons}=p & \Rightarrow p.2;
\end{align*}
\]

\text{- tl} : \text{NatList} \rightarrow \text{NatList}
Can we define an infinite list in NatList?

- 1, 2, 1, 2, 1, 2, 1, 2, ...
- infList = fix (\f. cons 1 (cons 2 f))
- hd (tl (tl infList)) //get the 3rd element
- Unfortunately, will diverge
  - why?
Review: Reduction Order

- 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 (\f. 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 \( i \)th element in Fibonacci sequence is smaller than \( C \)

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

  **Haskell version:**
  ```haskell
  fib = 0 : scanl (+) 1 fib
  length takeWhile (< C) fib
  ```
Recursive Functional Types

• What is this function type about?

\[ \text{Stream} = \mu A. \text{Unit} \rightarrow \{ \text{Nat}, A \}; \]

• Returning elements in an infinite sequence one by one
  • Continuation

• Java counterpart: iterator
  • With a mutable state
A Fibonacci stream

Stream = \mu X. \text{Unit} \rightarrow \{\text{Nat}, X\};

fibonacci =
let fib = fix (\lambda f:\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Stream}.
  \lambda x:\text{Nat}. \lambda y:\text{Nat}.
  \lambda_:\text{Unit}. \{x, f y (\text{plus} x y)\})
  in
  fib 0 1;

• Why not diverge?
Exercises

• 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 fullquirec
Is this a correct implementation?

- 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 l));
Review: General Recursions

- Introduce “fix” operator: $\text{fix } f = f(\text{fix } f)$

New syntactic forms:

- $t ::= \ldots$
- $\text{fix } t$

Fixed point of $t$

New typing rules:

- $\Gamma \vdash t : T$

New evaluation rules:

- $\text{fix } (\lambda x : T_1 . t_2) \rightarrow [x \mapsto (\text{fix } (\lambda x : T_1 . t_2))] t_2$
- $t \rightarrow t'$

(E-FixBeta)

New derived forms:

- $\text{letrec } x : T_1 = t_1 \text{ in } t_2$
- $\text{def } = \text{let } x = \text{fix } (\lambda x : T_1 . t_1) \text{ in } t_2$

(E-Fix)
Correction

• 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. 1 unit)));
Hungry Function

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

  • Hungry = \( \mu A. \text{Nat} \rightarrow A \);

  • \( f = \text{fix} (\lambda f: \text{Hungry}. \lambda n:\text{Nat}. \ f) \);
Representing Objects

• Can we represent the following immutable counter?

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

• Not with recursive type:

```
• Counter = {get: Unit → Nat, inc: Unit → Counter, dec: Unit → Counter}
```
Functional Objects

Counter = \mu C. \{get: Nat, inc: Unit \to C, dec: Unit \to C\};

\begin{align*}
c = \text{let create} &= \text{fix } (\lambda f: \{x: \text{Nat}\} \to \text{Counter}. \lambda s: \{x: \text{Nat}\}. \{ \\
&\text{get} = s.x, \\
&\text{inc} = \lambda _: \text{Unit}. f \{x=\text{succ}(s.x)\}, \\
&\text{dec} = \lambda _: \text{Unit}. f \{x=\text{pred}(s.x)\} \} ) \\
\text{in create} \{x=0\};
\end{align*}

- c : Counter

\begin{align*}
c1 &= c.\text{inc } \text{unit}; \\
c2 &= c1.\text{inc } \text{unit}; \\
c2.\text{get};
\end{align*}

- 2 : Nat
**Review: fixed-point combinator**

- **Law:** $\text{fix } f = f (\text{fix } f)$
- **Y Combinator**
  \[
  Y = \lambda f. \ (\lambda x. \ f \ (x \ x)) \ (\lambda x. \ f \ (x \ x))
  \]
- **Use of Y Combinator:** calculating $\Sigma_{i=0}^{n} i$
  \[
  f = \lambda f. \ \lambda n. \\
  \quad \text{if (iszero } n \text{) then } 0 \\
  \quad \text{else } n + f (n - 1)
  \]
  \[
  Y \ f
  \]
Review: fixed-point combinator

\[ Y = \lambda f. (\lambda x. f (x \, x)) \, (\lambda x. f (x \, x)) \]

\[ \text{fix} = \lambda f. (\lambda x. f (\lambda y. x \, x \, y)) \, (\lambda x. f (\lambda y. x \, x \, y)) \]

• Why fix is used instead of \( Y \)?
Answer

\[
\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y))\ (\lambda x. f (\lambda y. x x y))
\]

\[
Y = \lambda f. (\lambda x. f (x x))\ (\lambda x. f (x x))
\]

- Under full beta-reduction: Let \( f : T \to T \)
  - When \( T \) is a function type
    - Fix and \( Y \) are equal: \((\lambda y (x x) y)\ v = (x x)\ v = (\text{fix} f)\ v\)
  - Else
    - (Fix \( f \)) will stuck, while (\( Y \ f \)) will diverge

- Not under call-by-value because
  - \((x x)\) is not a value
  - while \((\lambda y. x x y)\) is
  - \( Y \) will diverge for any \( f \)
Review: fixed-point combinator

\[ \text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) \]

\[ Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

- Can we define \( Y \) in simple typed \( \lambda \)-calculus?
  - No
  - \( x \) has a recursive type
  - \( Y \) was defined as a special language primitive
Defining \texttt{fix} using recursive types

\[ Y_T = \lambda f : T \rightarrow T. \ (\lambda x : (\mu A. A \rightarrow T). \ f \ (x \ x)) \ (\lambda x : (\mu A. A \rightarrow T). \ f \ (x \ x)) \]

\[ Y_T : (T \rightarrow T) \rightarrow T \]

• 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 = \( \mu A. \text{Nat} \rightarrow A \);
• \( h = \text{fix} \ (\lambda f: \text{Nat} \rightarrow \text{Hungry}. \ \lambda n: \text{Nat}. \ f) \);

• What is the type of \( h \)?
  • Hungry?
  • \( \text{Nat} \rightarrow \text{Hungry} \)?
  • \( \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Hungry} \)?
Simple but Effective Solution

• Every term has one type
• Use fold/unfold to convert between types
• $h = \text{fix} \ (\lambda f:\text{Nat} \to \text{Hungry}. \ \lambda n:\text{Nat}. \ f)$
  • $h: \text{Nat} \to \text{Hungry}$
  • fold[Hungry] $h$: Hungry
  • unfold[Hungry] (h 1): Nat$\to$Hungry
Iso-recursive Types

$\rightarrow \mu$

$t ::= \ldots$

\[ \text{fold}[T] t \]

\[ \text{unfold}[T] t \]

$v ::= \ldots$

\[ \text{fold}[T] v \]

$T ::= \ldots$

\[ X \]

\[ \mu X. T \]

Extended $\lambda$- (9-1)

```
t_1 \rightarrow t'_1
fold[T] t_1 \rightarrow \text{fold}[T] t'_1
```

(E-FLD)

```
t_1 \rightarrow t'_1
\text{unfold}[T] t_1 \rightarrow \text{unfold}[T] t'_1
```

(E-UNFLD)

New typing rules

```
\Gamma \vdash t : T
```

(T-FLD)

```
U = \mu X. T_1 \quad \Gamma \vdash t_1 : [X \rightarrow U]T_1
```

\[ \Gamma \vdash \text{fold}[U] t_1 : U \]

(T-UNFLD)

New evaluation rules

```
\text{unfold}[S] (\text{fold}[T] v_1) \rightarrow v'_1
```

(E-UNFLD FLD)

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

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

• NatList = \( \mu X. \langle \text{nil:Unit, cons:}\{\text{Nat,X}\} \rangle \)
• NLBody = \( \langle \text{nil:Unit, cons:}\{\text{Nat,NatList}\} \rangle \)
• nil = \text{fold [NatList]}(\langle \text{nil=unit} \rangle \text{ as NLBody} ;
• cons = \( \lambda n:\text{Nat. } \lambda l:\text{NatList. } \text{fold[NatList]} \langle \text{cons=}\{n,l\} \rangle \text{ as NLBody} \)
Example

\[\text{isnil} = \lambda l: \text{NatList}.\]
\[
\text{case unfold } [\text{NatList}] l \text{ of }
\]
\[
<\text{nil}=u> \Rightarrow \text{true}
\]
\[
| <\text{cons}=p> \Rightarrow \text{false};
\]
\[\text{hd} = \lambda l: \text{NatList}.\]
\[
\text{case unfold } [\text{NatList}] l \text{ of }
\]
\[
<\text{nil}=u> \Rightarrow 0
\]
\[
| <\text{cons}=p> \Rightarrow p.1;
\]
\[\text{tl} = \lambda l: \text{NatList}.\]
\[
\text{case unfold } [\text{NatList}] l \text{ of }
\]
\[
<\text{nil}=u> \Rightarrow 1
\]
\[
| <\text{cons}=p> \Rightarrow p.2;\]
Implementation Problem 2

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

• What is the subtype relation between A and B?
  • A <: B?
  • B <: A?
  • No relation?
Subtyping by assumption

\[ \Sigma, X <: Y \vdash S <: T \]
\[ \Sigma \vdash \mu X . S <: \mu Y . T \]

**Example:**

- Even <: Nat
- \( A = \mu X . \text{Nat} \rightarrow (\text{Even} \times X) \)
- \( B = \mu Y . \text{Even} \rightarrow (\text{Nat} \times Y) \)

- Assuming \( X <: Y \)
- We have \( \text{Nat} \rightarrow (\text{Even} \times X) <: \text{Even} \rightarrow (\text{Nat} \times Y) \)
- 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