Recursive Types

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong
Peking University, Spring Term, 2014
Review: Why learn type theories?

• Art vs. Knowledge
  • Art cannot be taught, while knowledge can
  • What people have invented
  • How to interpret them abstractly
  • How to reason their properties formally

• Why formal reasoning important
  • Poorly designed languages widely used
    • Java array flaw
    • JavaScript: google “JavaScript sucks”
    • PHP: you know it
  • Well designed language needs strictly reasoning
    • Devils in details
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

• Non-nominal 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;
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 $\mu$
  • NatList = $\mu X$. <nil:Unit, cons:{Nat,X}>
  • Meaning: X = <nil:Unit, cons:{Nat,X}>.

• Constructors of NatList

  nil = <nil=unit> as NatList;

  • nil : NatList

  cons = $\lambda n$:Nat. $\lambda l$:NatList. <cons={n,l}> as NatList;

  • cons : Nat $\rightarrow$ NatList $\rightarrow$ NatList
NatList Functions

\[
\text{isnil} = \lambda l: \text{NatList. case } l \text{ of } \begin{cases} 
\langle \text{nil}=u \rangle & \Rightarrow \text{true} \\
| \langle \text{cons}=p \rangle & \Rightarrow \text{false};
\end{cases}
\]

\[\text{hd} = \lambda l: \text{NatList. case } l \text{ of } \langle \text{nil}=u \rangle \Rightarrow 0 | \langle \text{cons}=p \rangle \Rightarrow p.1;\]

\[\text{tl} = \lambda l: \text{NatList. case } l \text{ of } \langle \text{nil}=u \rangle \Rightarrow 1 | \langle \text{cons}=p \rangle \Rightarrow p.2;\]

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

• 1, 2, 1, 2, 1, 2, 1, 2, ...
• \( \text{infList} = \text{fix (}\lambda f. \text{cons 1 (cons 2 f)}) \)
• \( \text{hd (tl (tl infList))} \) //get the 3\(^{rd}\) 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
  • Normal Order + No reduction inside abstractions

• Call by value (used in the book)
  • Only outmost redexes are reduced
  • Parameters need to be values

• \( \text{infList} = \text{fix (} \lambda f. \text{ cons 1 (cons 2 f)} \text{)} \)

• \( \text{hd (} \text{tl (tl infList))} //\text{get the 3}^{\text{rd}} \text{ element} \)
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\{X, \text{Nat}\};

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}. \{f \ y \ (\text{plus} \ x \ y), \ x\})
  in
  fib 0 1;

• Why not diverge?
Exercies

• Change Stream to represent both finite and infinite list
• Two functions “nil” and “cons” for list constructions
• Construct the following two list in your implementation
  • 01
  • 1212121212…
List with infinite members

- InfList = Rec X.
  <infNil:Unit, infCons:{Nat, Unit->X}>;

- infNil = <infNil=unit> as InfList;

- infCons =
  lambda n:Nat. lambda l:Unit->InfList.
  <infCons={n,l}> as InfList;

- infList = fix (λf.
  infCons 1
  (λ_:Unit.(infCons 2 (λ_:Unit.f))))
Hungry Function

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

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

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

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

\[c = \text{let create} = \text{fix} (\lambda f: \{x: \text{Nat}\} \to \text{Counter}. \lambda s: \{x: \text{Nat}\}. \{\text{get} = s.x, \text{inc} = \lambda _: \text{Unit}. f \{x = \text{succ}(s.x)\}, \text{dec} = \lambda _: \text{Unit}. f \{x = \text{pred}(s.x)\}\})\]

\[\text{in create} \{x = 0\};\]

- \(c: \text{Counter}\)

\(c1 = c.\text{inc} \\text{unit};\)
\(c2 = c1.\text{inc} \\text{unit};\)
\(c2.\text{get};\)

- \(2: \text{Nat}\)
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 \( f : T \rightarrow T \)
  • 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
    • \((\text{Fix} \, f)\) 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

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

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

• 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
- $h = \text{fix} \left( \lambda f : \text{Nat} \rightarrow \text{Hungry}. \lambda n : \text{Nat}. f \right)$
  - $h : \text{Nat} \rightarrow \text{Hungry}$
  - $\text{fold}[\text{Hungry}] \ h : \text{Hungry}$
  - $\text{unfold}[\text{Hungry}] \ (h \ 1) : \text{Nat} \rightarrow \text{Hungry}$
Iso-recursive Types

\[ t ::= \ldots \]
\[ \text{fold} [T] t \]
\[ \text{unfold} [T] t \]

\[ v ::= \ldots \]
\[ \text{fold} [T] v \]

\[ T ::= \ldots \]
\[ X \]
\[ \mu X . T \]

\text{terms:}

- \[ t \rightarrow t' \]
- \[ \text{fold} [T] t \rightarrow \text{fold} [T] t' \]

\text{values:}

- \[ t \rightarrow t' \]
- \[ \text{unfold} [T] t \rightarrow \text{unfold} [T] t' \]

\text{types:}

- \[ \Gamma \vdash t : T \]

\text{New typing rules:}

- \[ \Gamma = \mu X . T \quad \Gamma \vdash t_1 : [X \rightarrow U] T \quad \Gamma \vdash \text{fold} [U] t_1 : U \]

- \[ \Gamma = \mu X . T \quad \Gamma \vdash t_1 : U \quad \Gamma \vdash \text{unfold} [U] t_1 : [X \rightarrow U] T \]

\text{New evaluation rules:}

- \[ \text{unfold} [S] (\text{fold} [T] v_1) \rightarrow v_1 \]

(E-FLD)

(E-UNFLD)

(T-FLD)

(T-UNFLD)

(E-UNFLDFLD)

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

• NatList = $\mu X. <\text{nil:Unit}, \text{cons:}\{\text{Nat}, X\}>

• NLBody = <\text{nil:Unit}, \text{cons:}\{\text{Nat, NatList}\}>

• nil = fold [NatList](<\text{nil=unit}> as NLBody);

• cons = $\lambda n: \text{Nat}. \lambda l: \text{NatList}. \text{fold}[\text{NatList}] <\text{cons=}\{n, l\}>$ as NLBody
Example

\[
\begin{align*}
isnil &= \lambda l: \text{NatList}. \\
&\quad \text{case unfold } [\text{NatList}] l \text{ of} \\
&\quad \quad <\text{nil}=u> \Rightarrow \text{true} \\
&\quad \quad | <\text{cons}=p> \Rightarrow \text{false}; \\
\text{hd} &= \lambda l: \text{NatList}. \\
&\quad \text{case unfold } [\text{NatList}] l \text{ of} \\
&\quad \quad <\text{nil}=u> \Rightarrow 0 \\
&\quad \quad | <\text{cons}=p> \Rightarrow p.1; \\
\text{tl} &= \lambda l: \text{NatList}. \\
&\quad \text{case unfold } [\text{NatList}] l \text{ of} \\
&\quad \quad <\text{nil}=u> \Rightarrow l \\
&\quad \quad | <\text{cons}=p> \Rightarrow p.2;
\end{align*}
\]
Iso-recursive types

• Used in many languages
  • Java, Haskell, Ocaml, etc.

• Fold/unfold can be omitted by special design
  • Only recursive types over data types, not functions
    • Provide member access functions
      • Java: user declared functions
      • Haskell, Ocaml: the functions in the previous example are automatically generated

• C#: nominal function types.
  • “delegate int A()” and “delegate int B()” are different
Implementation Problem 2

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

• 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 = $\mu X . \text{Nat} \to (\text{Even} \times X)$
  - B = $\mu Y . \text{Even} \to (\text{Nat} \times Y)$

- Assuming X <: Y
- We have Nat $\to (\text{Even} \times X)$ <: Even $\to (\text{Nat} \times Y)$
- Thus A <: B

- Why this works? Principle of safe substitution.
- Its implementing algorithm will be explained in the next course
Homework

• Defining a function f: NatList -> Stream that returns the elements in the NatList one by one, and returns 0 if the list is empty.
  • Implement and test your function both in fullisorec and fullequirec.
  • Please submit an electronic version with several test cases so that the teaching assistants can easily verify your implementation.

• 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