Lecture 14: Recursive Types

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages
Motivation

- Lists, so far
  - Introduce a type constructor $List \, T$
  - Values are either nil or cons $(e_{hd}, e_{tl})$
  - List have arbitrary size, but regular structure

- Similarly, queues, binary trees, labeled trees, ASTs, etc

- It is impractical to extend the language with each as an additional primitive type!

- Solution: recursive types
Example

- Lists of numbers:
  
  \[ NatList = \langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, NatList\} \rangle \]

- This equation defines an infinite tree
- To change into a definition, use abstraction

  \[ NatList = \mu X. \langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, X\} \rangle \]

- \( \mu \) is the explicit recursion operator for types
- Intuitively: “\( NatList \) is the type that satisfies the equation \( X = \langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, X\} \rangle \)”
Example: Lists

- Lists
  - nil = \langle nil = () \rangle \text{ as } NatList
  - cons = \lambda x: Nat. \lambda l: NatList. \langle cons = \{x, l\} \rangle \text{ as } NatList
  - isnil = \lambda l: NatList. \text{case } l \text{ of } \text{nil}(_\_ \_ ) \Rightarrow true \mid cons(_\_ \_ ) \Rightarrow false
  - hd = \lambda l: NatList. \text{case } l \text{ of } \text{nil}(_\_ \_ ) \Rightarrow 0 \mid cons(p) \Rightarrow p.1
  - tl = \lambda l: NatList. \text{case } l \text{ of } \text{nil}(_\_ \_ ) \Rightarrow l \mid cons(p) \Rightarrow p.2
  - sum = \text{fix } \lambda f: NatList \rightarrow Nat. \lambda l: NatList.
    \text{case } l \text{ of } \text{nil}(_\_ \_ ) \Rightarrow 0 \mid cons(p) \Rightarrow p.1 + (f \ p.2)
Hungry functions

- A function that can always take more:

\[ \text{hungry} = \mu X. \text{Nat} \rightarrow X \]

- Such a function is a fixpoint (recursive function):

\[ f = \text{fix} (\lambda f : \text{Nat} \rightarrow \text{hungry}. \lambda n : \text{Nat}. f) \]

- What is the type of \( f \ 1 \ 2 \ 3 \ 4 \ 5 \)?
Streams

- A stream is a function that can return an arbitrary number of values.
- Each time it consumes a unit, returns a new value.

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

- We can use it like an infinite list.
  - Next item \( \text{hd} = \lambda s : \text{Stream}.(s().1) \)
  - Rest of stream \( \text{tl} = \lambda s : \text{Stream}.(s().2) \)

- The stream of all natural numbers:

\[
\text{fix } (\lambda f : \text{Nat} \rightarrow \text{Stream}.) \lambda n : \text{Nat}. \lambda _ : \text{Unit}. \{ n, f(\text{succ } n) \})0
\]
Objects

- Objects can also be recursive types

\[ \text{Counter} = \mu C. \{ \text{get} : \text{Nat}, \text{inc} : \text{Unit} \to C \} \]

- Unlike last time, this is a functional object: \( \text{inc} \) returns the new object
  
  - Java strings are immutable
Recursive type of fixpoint

- Using recursive types we can type the fixpoint operator

  \[
  \text{fix}_T = \lambda f : T \rightarrow T. \\
  (\lambda x : (\mu X. X \rightarrow T). f (x x)) (\lambda x : (\mu X. X \rightarrow T). f (x x))
  \]

- Without types this is the fixpoint combinator of untyped calculus
- Allows programs to diverge: not strongly normalizing
- A term that doesn’t terminate can have any type \( T \)!
- By Curry-Howard:
  - All propositions are proved, including false!
  - The corresponding logic is inconsistent
Type system

- Two ways to treat recursive types
- Depending on the relation between folded/unfolded type
  - e.g: `NatList` and `⟨nil : Unit, cons : {Nat, NatList}⟩`
- Implicit fold/unfold, the above types are equal in all contexts
  - Transparent to the programmer
  - More complex to write typechecker
  - All proofs remain the same (except induction on type expressions)
- Explicit fold/unfold using language primitives
  - Programmer must write fold/unfold primitives to help typechecker
  - Easier to typecheck
  - Requires extra proof cases for soundness: fold/unfold
Type system (cont’d)

- Syntax:
  
  \[ e ::= \ldots \mid \text{fold } [T] \ e \mid \text{unfold } [T] \ e \]
  
  \[ \nu ::= \ldots \mid \text{fold } [T] \ \nu \]
  
  \[ T ::= \ldots \mid X \mid \mu X. T \]

- Typing

  \[ \begin{array}{c}
  \frac{U = \mu X. T \quad \Gamma \vdash e : T[U/X]}{
  \Gamma \vdash \text{fold } [U] \ e : U}
  \\
  \frac{U = \mu X. T \quad \Gamma \vdash e : U}{
  \Gamma \vdash \text{unfold } [U] \ e : T[U/X]}
  \end{array} \]
Semantics

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

\[
e \rightarrow e'
\]

\[
\text{fold } [T] e \rightarrow \text{fold } [T] e'
\]

\[
e \rightarrow e'
\]

\[
\text{unfold } [T] e \rightarrow \text{unfold } [T] e'
\]