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:

$$NatList = \langle \textit{nil} : Unit, \textit{cons} : \{Nat, NatList\} \rangle$$

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

$$NatList = \mu X. \langle \textit{nil} : Unit, \textit{cons} : \{Nat, X\} \rangle$$

- μ is the explicit recursion operator for types
- Intuitively: "$NatList$ is the type that satisfies the equation $X = \langle \textit{nil} : Unit, \textit{cons} : \{Nat, X\} \rangle$"

Example: Lists

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

Hungry functions

- A function that can always take more:

$$hungry = \mu X.Nat \to X$$

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

$$\mathsf{f} = \mathsf{fix}\ (\lambda f : Nat \to hungry.\lambda n : Nat.f)$$

- 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

$$Stream = \mu X. Unit \to \{Nat, X\}$$

- We can use it like an infinite list
  - Next item hd $= \lambda s : Stream.(s\ ()).1$
  - Rest of stream tl $= \lambda s : Stream.(s\ ()).2$
- The stream of all natural numbers:

$$\text{fix } (\lambda f : Nat \to Stream.\lambda n : Nat.\lambda\_ : Unit.\{n, f(\text{succ } n)\})0$$

Objects

- Objects can also be recursive types

$$\mathit{Counter} = \mu C.\,\{\mathit{get} : Nat, \mathit{inc} : Unit \to C\}$$

- Unlike last time, this is a functional object: *inc* returns the new object
  - Java strings are immutable

Recursive type of fixpoint

- Using recursive types we can type the fixpoint operator

$$\begin{aligned} \mathrm{fix}_T = \lambda f : T \to T. \\ (\lambda x : (\mu X.X \to T).f\,(x\,x))\;(\lambda x : (\mu X.X \to T).f\,(x\,x)) \end{aligned}$$

- 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 $\langle \textit{nil} : Unit, \textit{cons} : \{Nat, NatList\}\rangle$
- 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:

$$\begin{array}{rcl} e & ::= & \ldots \mid \text{fold } [T]\ e \mid \text{unfold } [T]\ e \\ v & ::= & \ldots \mid \text{fold } [T]\ v \\ T & ::= & \ldots \mid X \mid \mu X.T \end{array}$$

- Typing

$$[\text{T-FOLD}]\ \frac{U = \mu X.T \qquad \Gamma \vdash e : T[U/X]}{\Gamma \vdash \text{fold } [U]\ e : U}$$

$$[\text{T-UNFOLD}]\ \frac{U = \mu X.T \qquad \Gamma \vdash e : U}{\Gamma \vdash \text{unfold } [U]\ e : T[U/X]}$$

Semantics

$$\frac{}{\text{unfold } [S] \ (\text{fold } [T] \ v) \rightarrow v}$$

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

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