Lecture 14: Recursive Types

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Static Analysis
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:

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

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

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

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

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

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

- 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

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

- 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:

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

- Objects can also be recursive types

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

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

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

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

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

\[
\begin{array}{c}
e \rightarrow e' \\
\text{fold } [T] e \rightarrow \text{fold } [T] e'
\end{array}
\]

\[
\begin{array}{c}
e \rightarrow e' \\
\text{unfold } [T] e \rightarrow \text{unfold } [T] e'
\end{array}
\]