Recursive Types

Programming Languages Reading Group, Indiana University

Kartik Sabharwal

2022-02-01

Types as Sets of Values

Additions after initial presentation are colored teal.

$$
\begin{aligned}
[\![\text{Integer}]\!] &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![A \cup B]\!] &= [\![A]\!] \cup [\![B]\!] \\
[\![A \cap B]\!] &= [\![A]\!] \cap [\![B]\!] \\
[\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![A + B]\!] &= \{\text{inj}_1(a) \mid a \in [\![A]\!]\} \cup \{\text{inj}_2(b) \mid b \in [\![B]\!]\} \\
[\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} \text{ (the set of functions from } A \text{ to } B\text{)}
\end{aligned}
$$

(The last definition is good for building intuition but we need to be careful that our model doesn't raise a cardinality problem)

Trivia. If X and Y are finite sets,

$$|X \times Y| = |X| \times |Y|, \qquad |X + Y| = |X| + |Y|, \qquad |Y^X| = |Y|^{|X|}$$

Types as Sets of Values – Examples

```
#lang typed/racket/base

(struct (S) inj1 ([s : S]))
(struct (T) inj2 ([t : T]))
(define-type (Sum S T) (U (inj1 S) (inj2 T)))

(define-type A (U False 1 2))
(define-type B Boolean)

(define ex1 : (U A B) 2)
(define ex2 : (Intersection A B) #f)
(define ex3 : (Pair A B) (cons 1 #t))
(define ex4 : (Sum A B) (inj1 (ann 2 A)))
(define ex5 : (-> A B) (lambda ([a : A]) (if a #t a)))
```

Recursive Types – Motivation 1

A lot of objects that we want to model using programming languages have a self-referential structure. Natural numbers, lists, binary trees and abstract syntax trees are popular examples.

```
(define-type Natural^ (Sum Null Natural^))

(define ex6 : Natural^ (inj2 (inj2 (inj1 null))))
;; * * * * * * * * * * * * * * * * * * * * * * * * *
(define-type IntList (U Null (Pair Integer IntList)))

(define ex7 : IntList null)
(define ex8 : IntList (cons 1 (cons 5 (cons 4 null))))
```

Recursive Types – Motivation 2

```
(struct Leaf ())
(struct (T) Node ([left : T]
                  [value : Integer]
                  [right : T]))
(define-type Tree (U Leaf (Node Tree)))

(define ex9 : Tree (Node (Leaf) 1 (Leaf)))
(define ex10 : Tree (Node ex9 0 ex9))
;; * * * * * * * * * * * * * * * * * * * * * * * * *
(define-type IntFun (U Integer (-> Integer IntFun)))

(define ex11 : IntFun
  (lambda ([x : Integer])
    (if (> x 0) x ex11)))
```

Recursive Types – Motivation 3

```
(struct Lit ([b : Boolean]))
(struct Var ([s : Symbol]))
(struct (BE) And ([beL : BE] [beR : BE]))
(struct (BE) Or ([beL : BE] [beR : BE]))
(struct (BE) Not ([be : BE]))
(define-type BoolExp
  (U Lit Var (And BoolExp) (Or BoolExp) (Not BoolExp)))

(define ex12 : BoolExp (Not (And (Lit #t) (Var 'x))))
```

We ought to be able to visualize these types as sets when we want to prove properties about them.

Recursive Type → Generating Function

We defined a recursive type intended to represent the natural numbers via an equation.

$$\text{Natural} \;=\; \text{Null} + \text{Natural}$$

We can confine the "unknown" in the equation to the left-hand side using a μ variable binder.

$$\text{Natural} \;=\; \mu X.\ \text{Null} + X$$

TAPL defines $[\![\text{Natural}]\!]$ as the "greatest fixed-point" of the following function on sets of values. We can call it the "generating function" for Natural.

$$F(X) = \{\text{inj1}(\text{null})\} \cup \{\text{inj2}(x) \mid x \in X\}$$

Fixed-Point

A "fixed-point" of a function is a value that doesn't change when we apply the function to it.

Any set X such that $F(X) = X$ is a fixed-point of F

If F has two fixed-points X_1 and X_2, $X_1 \subseteq X_2 \implies X_1 \leq X_2$.

Least Fixed Point

Kleene's Fixed-Point Theorem shows us how to compute the least fixed-point of F – repeatedly apply F to $\varnothing$ and perform a union over all elements in this sequence.

$$\begin{aligned}
F^1(\varnothing) &= \{\text{inj1}(\text{null})\} \\
F^2(\varnothing) &= \{\text{inj1}(\text{null}),\ \text{inj2}(\text{inj1}(\text{null}))\} \\
&\dots \\
F^n(\varnothing) &= \{\text{inj1}(\text{null}), \dots, \text{inj2}^{(n-1)}(\text{inj1}(\text{null}))\}
\end{aligned}$$

Applying F to the set $\cup_{i=0}^{\infty} F^i(\varnothing)$ doesn't add or remove any elements from it. We have our least fixed-point.

The least fixed-point is already an infinite set! How are we going to construct a larger fixed-point?

Greatest Fixed Point

Let $[\![\text{Any}]\!]$ be the set of all values in our language.

To crack a guess at what the greatest fixed point is let's iteratively apply F to $[\![\text{Any}]\!]$ and see which values are eventually killed off.

Observe that any value in the LFP will survive intact. We don't need to worry about those.

I think it's fair to say that any value of the form $\text{inj2}^n(\text{x})$, where $n \geq 0$, n is as large as possible and x is not inj1(null), won't be in $F^{(n+1)}([\![\text{Any}]\!])$

Suppose, for a moment, that we're allowed to have "infinite" values, like inj2 . . .. This value is special because $\text{inj2}(\text{inj2}^\infty) = \text{inj2}^\infty$

$\text{LFP} \cup \{\text{inj2}^\infty\}$ is the greatest fixed-point of F

Lazy vs Strict – 1

Infinite values aren't really a stretch when we're talking about programming languages. All we need is the ability to evaluate code lazily. Here's inj2$^\infty$ in Haskell, where's it's recognized as a Natural. (Given this, the type should more accurately be called "Conatural")

```
module RecursiveTypes where

data Natural = Inj1 () | Inj2 Natural

fix :: (t -> t) -> t
fix f = f (fix f)

inj2_infty :: Natural
inj2_infty = fix Inj2
```

Lazy vs Strict – 2

Typed Racket uses strict semantics so we can't pull the same trick twice. Even though the recursive type Natural theoretically admits infinite values, we can't actually create one.

We're aware that we can delay evaluation using thunks. Let's define a new type that supports delayed evaluation.

```
(define-type Conatural (-> Null (Sum Null Conatural)))
```

Note that the generating function for this recursive type is non-trivially different from that of Natural, but I'm certain the information content is the same as Haskell's Natural type. We can now represent infinity.

```
(define inj2_infty : Conatural
  (lambda ([_ : Null]) (inj2 inj2_infty)))
```

References

Chapters 20 and 21 of "Types and Programming Languages" by Benjamin C. Pierce

A Note on Recursive Types and Fixed Points by Aaron Stump

Cornell CS 4110 – Denotational Semantics Examples

Recommended during talk: Calculating Functional Programs by Jeremy Gibbons