MFCS

Rectypes

Klaus Sutner
Carnegie Mellon University
Fall 2022

1 **Rectypes (aka Inductive Sets)**

2 **Rectype List**

Building Things Bottom-Up

A lot of important objects are constructed in an inductive/recursive manner. The general framework looks like so: we are given

- one or several atoms
- one or several constructors.

We are interested in the objects produced by repeated application of constructors to atoms. For us, repeated always means finitely often.

We call the atoms are primitive, everything is compound.

We will write constructors with square brackets to distinguish them from ordinary functions. So $C[a, b]$ is a constructor with two arguments.

Recursive Data Types

Given atoms and constructors, we can define a recursive data type (rectype for short) or inductively defined set[†] by systematically applying the constructors over and over again to all atoms.

More precisely, call a set X inductive (wrto our atoms and constructors) if X contains all the atoms, and is closed under the constructors: if $a_1, a_2, \ldots, a_k \in X$ then $C[a_1, a_2, \ldots, a_k] \in X$ where C is a constructor requiring k inputs (usually $k = 1, 2$).

Then the rectype defined by these atoms and constructors is

$$D = \bigcap \{\, X \mid X \text{ inductive} \,\}$$

[†]Rectype is a neologism that we have stolen from T. Forster in Cambridge; it is somewhat nonstandard, but it's too good not to use.

Deconstruction

For our purposes, the most interesting scenario is when all objects in D are either primitive or compound, but never both.

Also, for a compound object b such that

$$b = C[a_1, a_2, \ldots, a_k]$$

there is no other way of deconstructing b.

The a_i might again be compound and can be taken apart in a unique way, over and over, until we hit rock bottom and wind up with atoms.

The Killer App

We can think of the natural numbers as being constructed from

- the single atom $\underline{0}$,
- the single constructor $\underline{S}$.

Thus we obtain $\underline{0}$, $\underline{S}[\underline{0}]$, $\underline{S}[\underline{S}[\underline{0}]]$, $\underline{S}[\underline{S}[\underline{S}[\underline{0}]]]$, ...

These simply represent the intuitive naturals 0, 1, 2, 3, ..., without using any particular digit-based notation system for the naturals, so-called numeration systems.

Why would we do this? Because there are many possible numeration systems, and it is a bad idea to bake these into the fundamental definitions. They come into play when we start to develop efficient algorithms operating on the naturals.

Naturals, Formalized

Again, the rectype Nat is defined by the atom $\underline{0}$ and the constructor $\underline{S}$.

That's it, nothing else.

If you are a friend of set theory you will want to think of $\underline{S}$ as some kind of function. Since we insist on unique deconstruction, we need to have

$$\underline{S}[x] \neq \underline{0}$$

$$\underline{S}[x] = \underline{S}[y] \Rightarrow x = y$$

In other words, $\underline{S}$ must be injective and $\underline{0}$ cannot lie in its range. By injectivity, the decomposition is unique.

This is exactly what the Dedekind-Peano axioms say, nothing new here, really.

Wordprocessing

Distinctly more scary is to think of $\underline{S}$ as just a symbol, with no particular meaning, so that our formal "natural numbers" are just the terms $\underline{0}, \underline{S}[\underline{0}], \underline{S}[\underline{S}[\underline{0}]], \ldots$ We are manipulating formal expressions, that's all.

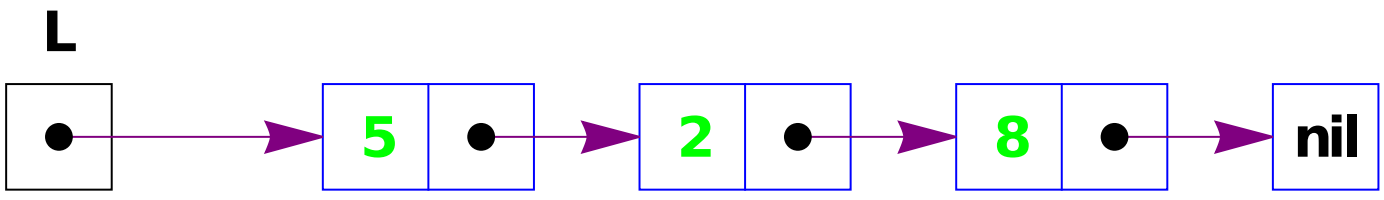
Most people seem to hate this approach, we like to think about semantics, not pushing symbols around on paper. That's fine, but the symbolic interpretation is just as valid[†].

Why bother? Because wordprocessing is essentially the only thing a computer can do. More precisely, computers inherently perform symbolic computations.

[†]Hilbert made a big fuss about this.

Linked Lists

A hugely important data structure that seem quaint today is a linked list. Linked lists were invented in 1953 by H./.P/.Luhn, and implemented by Newell, Shaw and Simon in 1956 in their work on artificial intelligence. A little later, McCarthy's Lisp used linked lists as a foundational concept.



How do we reason about these lists?

Axioms for Lists

Here is a simplified problem: we only consider lists of urelements (rather than nested lists). Fix some ground set A of potential list elements. Here is the rectype $\mathrm{List}(A)$ of all lists over A.

- single atom nil,
- constructors $\mathrm{prep}[a, L]$ for all $a \in A$.

nil represents the empty list, and $\mathrm{prep}[a, L]$ stands for the prepend operation.

To lighten notation we usually write $a :: L$ instead of $\mathrm{prep}[a, L]$.

Why? Because this sort of algebraic notation is better for humans.

Building Lists

What is ordinarily written as the list $(a_1, a_2, \ldots, a_n)$ is now represented by the composite object

$$\mathsf{prep}[a_1, \mathsf{prep}[a_2, \ldots, \mathsf{prep}[a_n, \mathsf{nil}], \ldots]]$$

which is much easier on the eye when written as

$$a_1 :: a_2 :: \ldots :: a_n :: \mathsf{nil}$$

This approach is taken directly from Lisp (one of the most important programming languages ever, maybe the most important one). It translates easily into a pointer-based data structure.

Decomposition

The corresponding destructors are

$$\begin{aligned} \mathsf{head} &: \mathsf{List}(A) \to A \\ \mathsf{tail} &: \mathsf{List}(A) \to \mathsf{List}(A) \end{aligned}$$

such that

$$K = a :: L \quad \text{implies} \quad a = \mathsf{head}(K), L = \mathsf{tail}(K)$$

Note that, as written, both destructors are undefined for nil. For any non-empty list $L = a_1 :: a_2 :: \ldots :: a_n :: \mathsf{nil}$ we have

$$\begin{aligned} \mathsf{head}(L) &= a_1 \\ \mathsf{tail}(L) &= a_2 :: \ldots :: a_n :: \mathsf{nil} \end{aligned}$$

The Rectype List

One usually does not bother to spell this out, but as before for the rectype Nat, we want List(A) to be the least set that

- contains the atom nil
- contains $a :: L$ for any $a \in A$ whenever it contains L.

This minimality condition excludes weird and unintended "lists" containing lists as elements, containing elements not in A, or, heaven forfend, infinite lists.

Structural Induction on Lists

Theorem (Induction for Lists)

Suppose $\mathcal{X} \subseteq \mathsf{List}(A)$ *where*

- $\mathsf{nil} \in \mathcal{X}$ *and*
- $a :: L \in \mathcal{X}$ *for all* $a \in A, L \in \mathcal{X}$.

Then $\mathcal{X} = \mathsf{List}(A)$.

Informally: any property that nil has, and that is inherited by $a :: L$ from L must already hold for all lists.

So the rectype $\mathsf{List}(A)$ is defined by induction, and we can use this theorem to prove properties of recursively defined operations on lists.

Justification

Why is this reasoning admissible?

Suppose there is a counterexample, some list L not in $\mathcal{X}$.

Choose L to be of minimal length (this works since any non-empyt set of naturals has a least element).

L cannot be nil since we checked that nil $\in \mathcal{X}$.

Hence, by decomposition, $L = a :: K$, $a \in A$, K a list shorter than L. By minimality, $K \in \mathcal{X}$.

We get a contradiction since $K \in \mathcal{X}$ implies $L = a :: K \in L$.

No Loss

Note that when $A = \{\bullet\}$ is a one-element set we are basically dealing again with the natural numbers: a list $\{\bullet, \bullet, \bullet\}$ is just the natural number 3, written in unary.

One can check that induction on these $\bullet$-lists is exactly the same as induction on the naturals.

Left and Right

However, there is one big difference between induction on the naturals and Induction on lists: for lists, we have a choice between induction on the left and on the right.

Standard induction on the left:

- Base case: show $\varphi(\mathsf{nil})$
- Induction step:
  assuming $\varphi(L)$, show $\varphi(a :: L)$

Alternative induction on the right:

- Base case: show $\varphi(\mathsf{nil})$
- Induction step:
  assuming $\varphi(L)$, show $\varphi(L :: a)$

More Operations

Here is a definition of append in our framework.

$$\begin{aligned} \mathsf{app}(a, \mathsf{nil}) &= a :: \mathsf{nil} \\ \mathsf{app}(a, b :: L) &= b :: \mathsf{app}(a, L) \end{aligned}$$

Joining two lists together

$$\begin{aligned} \mathsf{join}(\mathsf{nil}, K) &= K \\ \mathsf{join}(a :: L, K) &= a :: \mathsf{join}(L, K) \end{aligned}$$

For legibility we often write $L :: a$ instead of $\mathsf{app}(a, L)$ and $K :: L$ instead of $\mathsf{join}(K, L)$.

Careful with parens, though. The law for append says $(b :: L) :: a = b :: (L :: a)$.

Yet More Operations

Erasing all occurrences of $a \in A$ from a list

$$\begin{aligned} \mathsf{erase}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{erase}(a :: L) &= \mathsf{erase}(L) \\ \mathsf{erase}(b :: L) &= b :: \mathsf{erase}(L) \qquad a \neq b \in A \end{aligned}$$

Keeping the first occurrence of $a \in A$:

$$\begin{aligned} \mathsf{keep1}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{keep1}(a :: L) &= a :: \mathsf{erase}(L) \\ \mathsf{keep1}(b :: L) &= b :: \mathsf{keep1}(L) \qquad a \neq b \in A \end{aligned}$$

Mixed Operation

The objects involved need not all be lists.

For example, we can define the length of a list as follows:

$$\mathsf{len}(\mathsf{nil}) = 0$$
$$\mathsf{len}(a :: L) = \mathsf{len}(L) + 1$$

We use the naturals informally here, but one could express everything quite easily in terms of the strict rectype definition of Nat.

Claim

$\mathsf{len}(\mathsf{join}(K, L)) = \mathsf{len}(K) + \mathsf{len}(L)$

Proof

Claim: $\mathsf{len}(\mathsf{join}(K, L)) = \mathsf{len}(K) + \mathsf{len}(L)$

Proof.

Base case: $K = \mathsf{nil}$

$$\mathsf{len}(\mathsf{nil} :: L) = \mathsf{len}(L) = 0 + \mathsf{len}(L) = \mathsf{len}(\mathsf{nil}) + \mathsf{len}(L)$$

Induction step: $K = a :: K'$.

$$\begin{aligned}\mathsf{len}((a :: K') :: L) &= \mathsf{len}(a :: (K' :: L)) = 1 + \mathsf{len}(K' :: L) \\ &= 1 + \mathsf{len}(K') + \mathsf{len}(L) = \mathsf{len}(K) + \mathsf{len}(L)\end{aligned}$$

□

Reversal

Here is a definition of the reversal operation on lists:

$$\mathsf{rev}(\mathsf{nil}) = \mathsf{nil},$$
$$\mathsf{rev}(a :: L) = \mathsf{rev}(L) :: a$$

Aside: If you worry about implementation this may look unappealing: append as defined is linear time on singly-linked lists, so this definition would produce a quadratic time reversal.

Solution: change the data structure.

The Message: Don't worry about implementation details too soon, first get the logic right.

A Reversal Proof

Claim

$\mathsf{rev}(L :: a) = a :: \mathsf{rev}(L)$ *for all* L, a.

Proof.

Base case: $L = \mathsf{nil}$

$$\begin{aligned}\mathsf{rev}(\mathsf{nil} :: a) &= \mathsf{rev}(a :: \mathsf{nil}) \\ &= \mathsf{rev}(\mathsf{nil}) :: a \\ &= \mathsf{nil} :: a \\ &= a :: \mathsf{nil} \\ &= a :: \mathsf{rev}(\mathsf{nil})\end{aligned}$$

Make sure to identify exactly which axioms are used at each step.

Step

Induction step: let $L = b :: K$.

$$\begin{aligned}
\mathsf{rev}((b :: K) :: a) &= \mathsf{rev}(b :: (K :: a)) \\
&= \mathsf{rev}(K :: a) :: b \\
&= (a :: \mathsf{rev}(K)) :: b \\
&= a :: (\mathsf{rev}(K) :: b) \\
&= a :: \mathsf{rev}(b :: K)
\end{aligned}$$

Again, make sure to identify the axioms for each step.

□

Proof Fatigue

Note that every single step in this type of proof is really simple: we only need to decide which axiom to use and when to apply the induction hypothesis.

This type of argument is called equational logic and is relatively easy to automate.

Alas, for humans it's not so simple: everyone's eyes glaze over after half a dozen steps. Plus, it's really easy to make silly mistakes.

Exercises

Exercise

$\mathsf{rev}(L :: K) = \mathsf{rev}(K) :: \mathsf{rev}(L)$ *for all* L, K.

Exercise

$\mathsf{rev}(\mathsf{rev}(L)) = L$ *for all* L.

Exercise

Write $\mathsf{rot}(L)$ *for the result of rotating* L *cyclically by one place to the left. Give an inductive definition of* rot *and characterize the lists* L *such that* $\mathsf{rot}(L) = L$.

The Rotation Problem

Problem:	**Rotation**
Instance:	An array of A, a positive integer s.
Solution:	Rotate A by s places.

Of course, the challenge is to do this with minimal resources.

How about linear time and $O(1)$ extra space?

This is surprisingly difficult. Clearly, we can rotate by one place in linear time and $O(1)$ extra space. But we cannot repeat $s = O(n)$ times without violating the linearity constraint.

Alternatively, we can use scratch space $O(s)$ to move the first s elements out of the way, and move everything in linear time, but that violates the space constraint.

The Reversal Trick

A clever and far from obvious trick is to use reversal to implement rotation. The key observation is that

$$\mathsf{rot}(u :: v, s) = \mathsf{rev}(\mathsf{rev}(u) :: \mathsf{rev}(v))$$

where u has length s.

In other words, reverse the initial segment of A of length s, then reverse the remainder, and in one last step reverse the whole array.

Since reversal can clearly be handled in linear time and $O(1)$ extra space, done.

Code


```
// reverse block from lo to hi, inclusive
void  reverse( int lo, int hi ) {
   int i,j, m = (hi-lo)/2;
   for( i=lo,j=hi; i<m; i++,j-- )
                swap( i, j );
}

// rotate left, len length of array
void  rotate_left( int s ) {
        s = s mod len;
        reverse( 0, s-1   );
        reverse( s, len-1 );
        reverse( 0, len-1 );
}
```

Exercise

What happens if we perform the `reverse(0, len-1)` *operation first?*