Structural Induction & Asymptotic Analysis

Structural engineering

15-150 M21

Lecture 0607
07 June 2021

0 Struct Induct

Type aliases and invariants

SML allows us to declare *type aliases*

```
type fivetuple = int * int * int * int * int
```

We can use this to help define types with invariants

```
(* INVARIANT: for all n:nat, n>=0 *)
type nat = int
```

```
(* fact : nat -> nat
 * REQUIRES: true
 * ENSURES: fact n == n!
 *)
fun fact 0 = 1
  | fact (n:nat):nat = n * fact(n-1)
```

A useful fantasy

In reality, `nat` is just `int`, so all the constructors of type `int` are constructors of type `nat`. But we pretend that `nat` is given by two constructors:

- 0
- Successor: if `n : nat`, then `n+1 : nat`

Obtaining the `nat` recursion & induction principles

Pretending that all nats are constructed from zero and successor matches our recursion & induction principle.

- **Recursion**: To define `f : nat -> t`, must provide `f(0)` and `f(n)`, using the value of `f(n-1)`.
- **Induction**: To prove $P(\texttt{n})$ for all `n:nat`, must prove $P(0)$ and $P(\texttt{n})$, using the assumption of $P(\texttt{n-1})$

Key Point

The recursion and induction principles for a (recursive) type exactly match the constructors of that type

Lists are recursively-defined type

Recall that the type `t list` has two constructors:

- `[] : t list`
- `(x::xs) : t list` if `x:t` and `xs : t list`

Recursion Principle: To define `f : t1 list -> t2`, must provide `f([]) : t2` and `f(x::xs) : t2`, using the values `x:t1` and `f(xs) : t2`

Induction Principle: To prove $P($`L`$)$ for all `L : t1 list`, must prove $P($`[]`$)$ and $P($`x::xs`$)$ for arbitrary `x:t1`, assuming $P($`xs`$)$.

Demonstration: @ totality proof

```
infix @
fun [] @ L = L
  | (x::xs) @ L = x::(xs @ L)
```

Check Your Understanding

- Prove that `len : t list -> int` is total
- Prove that `rev : t list -> t list` is total
- Formulate the necessary lemmas and prove

$$\texttt{len(L1 @ L2)} \cong \texttt{len(L1) + len(L2)}$$

  for all appropriately-typed values `L1,L2`
- Formulate the necessary lemmas and prove

$$\texttt{len(rev L)} \cong \texttt{len L}$$

  for all appropriately-typed values `L`

5-minute break?

1 Tail Recursion

Demonstration: `rev traces`

Today's slogan I:

Sometimes the best way to make your life easier is to make your life harder

```
trev : string list * string list -> string list
```
REQUIRES: true
ENSURES: `trev(L,acc)` $\cong$ `(rev L)@acc`

This is "harder" than `rev`: this function has an extra parameter, and the behavior of `rev` is just one special case (`acc = []`):

```
val rev = fn L => trev(L,[])
```

Demonstration: `trev`

live-coding

Prop. For all types `t` and all values `L`, `acc : t list`,

`trev(L,acc)` $\cong$ `(rev L)@acc`.

Proof by structural induction on `L`.

BC: `L=[]`.

`trev([],acc)`	$\cong$	`acc`	(defn. `trev`)
	$\cong$	`[] @ acc`	(defn. `@`)
	$\cong$	`(rev []) @ acc`	(defn. `rev`)

Prop. For all types `t` and all values `L,acc : t list`,

`trev(L,acc) ≅ (rev L)@acc.`

IH: Assume `trev(xs,acc') ≅ (rev xs)@acc'` for all `acc'`

Pick arbitrary `x:t` and `acc : t list`. WTS:

`trev(x::xs,acc) ≅ (rev (x::xs))@acc`

	`trev(x::xs,acc)`	
≅	`trev(xs,x::acc)`	(defn. `trev`)
≅	`(rev xs)@(x::acc)`	IH
≅	`((rev xs)@[x])@acc`	(Lemma, totality of `rev`)
≅	`(rev (x::xs))@acc`	(defn. `rev`)

Demonstration: `trev traces`

`trev` is an example of a *tail recursive* function.

Defn. A recursive function is said to be **tail recursive** if it does not perform any computation on the result of a recursive call

```
tfact : int * int -> int
```
REQUIRES: n $\geq$ 0
ENSURES: `tfact(n,acc)` $\cong$ `acc * (fact n)`

```
texp : int * int -> int
```
REQUIRES: n $\geq$ 0
ENSURES: `exp(n,acc)` $\cong$ `acc * (exp n)`

Check Your Understanding

- Prove `trev` is total
- Write `tfact`, `texp`, etc. so that they satisfy their specs
- Write a tail-recursive accumulator version of the following function

```
fun sum ([] : int list):int = 0
  | sum (x::xs) = x + sum(xs)
```

5-minute break?

2 Sequential Runtime Analysis

Today's slogan II:

Think big (and think long-term)

What we want

For a given function `f`, we want to know how long `(f v)` takes to evaluate to a value, for each `v` such that `(f v)` is valuable.

$$\texttt{v1} \quad \texttt{f v1} \Longrightarrow^{13} \texttt{v1'}$$

$$\texttt{v2} \quad \texttt{f v1} \Longrightarrow^{1} \texttt{v2'}$$

$$\texttt{v3} \quad \texttt{f v1} \Longrightarrow^{96000} \texttt{v3'}$$

$$\texttt{v4} \quad \texttt{f v1} \Longrightarrow^{115} \texttt{v4'}$$

In general, for each `v` such that `(f v)` is valuable, we want to know the least n such that

$$\texttt{f v} \Longrightarrow^{n} \texttt{v'}$$

for some value `v'`.

A few problems with this...

- Internal representations?

$$\texttt{6+6} \stackrel{?}{\Longrightarrow} \texttt{12}$$

- Hardware dependence: want algorithm analysis to be the same for both our computers, even if yours is twice as fast as mine.
- For small inputs, the costs of initiating the process and storing variables is much higher relative to the costs of actual computation

Recall: `exp versus pow`

We instead generally assume *large* inputs, and instead seek to classify what inpact doubling, tripling, etc. the size of the input has on the computation time.

$$\begin{aligned} W_{\texttt{f}} &: \mathbb{N} \to \quad \mathbb{N} \\ &: n \mapsto \text{(the number of steps it takes to evaluate} \\ &\qquad\quad \texttt{(f v)} \text{ if } \texttt{v} \text{ is some input of size } n\text{)} \end{aligned}$$

This assumes we have a well-defined notion of size defined on the input type of `f`, such that `(f v)` and `(f v')` take the same number of steps to evaluate whenever `v` and `v'` are of the same size.

We can classify such functions with big-O

The big-O complexity of a (mathematical) function $W : \mathbb{N} \to \mathbb{N}$ tells us how fast W grows, proportionally to its input:
Rough idea: Let $W, g : \mathbb{N} \to \mathbb{N}$. We say W is $O(g)$ if there's some constant $c > 0$ such that

$$W(n) \leq cg(n) \qquad \text{for sufficiently large } n$$

A bound is *tight* if there's no tighter bound which suffices. For the examples encountered in this class, it should be clear whether a bound is tight or not.
Example: $W(n) = 3n^2 + 4n + 2$. This function is $O(n^4)$ and $O(n^3)$, but a tight bound is $O(n^2)$.

- If W is $O(\log n)$, then quadrupling the size of the input adds 2 (units of time) to the runtime.

$$W(4n) \approx W(n) + 2$$

- If W is $O(n)$, then quadrupling the size of the input approximately quadruples the runtime:

$$W(4n) \approx 4W(n)$$

- If W is $O(n \log n)$, then quadrupling the size of the input scales and increments the runtime:

$$W(4n) \approx 4(W(n) + 2)$$

- If W is $O(n^2)$, then quadrupling the input size multiplies the runtime by 16.

$$W(3n) \approx 9W(n) \quad W(4n) \approx 16W(n) \quad W(5n) \approx 25W(n)$$

- If W is $O(2^n)$, then doubling the input size squares the output size, and tripling cubes the runtime.

$$W(2n) \approx (W(n))^2 \quad W(3n) \approx (W(n))^3$$

0. How you're quantifying input size
1. Recurrence
2. Description of work tree
3. Measurements of work tree (height, and width at each level)
4. Summation
5. Big-O

Worked example: `exp` analysis

Worked example: `pow` analysis

Worked example: @ analysis

Worked example: `rev` analysis

Summary

- We prove the behavior of structurally recursive functions using structural induction
- Accumulator arguments can facilitate efficient solutions to computational problems
- We can make use of the formalism of recurrences and asymptotic complexity to precisely articulate the runtime of recursive functional functions/algorithms

Next Time

- Analysis of Multi-Step Algorithms
- Sorting
- Parallel Runtime Analysis

Thank you!