Extensional Code Design

15-150 M21

Lecture 0604
04 June 2021

Today's slogan:

Make it faster

0 The Power of Strong Induction

Recall exp

exp : int -> int
REQUIRES: n $\geq$ 0
ENSURES: exp(n) $\cong$ 2^n

0604.0 (pow.sml)

```
fun exp (0:int):int = 1
  | exp n = 2 * exp(n-1)
```

Analysis: `exp` Code Trace

If n is even, then

$$2^{\mathtt{n}} = \left(2^{\mathtt{n\ div\ 2}}\right)^2$$

```
pow : int -> int
```
REQUIRES: $\mathtt{n} \geq 0$
ENSURES: $\mathtt{pow(n)} \cong \mathtt{exp(n)}$

By with a little help from my friends

0604.1 (pow.sml)

```
fun square (x:int):int = x * x
```

0604.2 (pow.sml)

```
fun even (x:int):bool = (x mod 2)=0
```

pow definition

```
pow : int -> int
```

REQUIRES: `n` $\geq$ `0`

ENSURES: `pow(n)` $\cong$ `exp(n)`

0604.3 (pow.sml)

```
fun pow (0:int):int = 1
  | pow n =
      case (even n) of
        true => square(pow(n div 2))
      | false => 2 * pow(n-1)
```

Analysis: `pow` Code Trace

Equivalence

Thm. For all values `n : int` where `n>=0`,

$$\texttt{exp(n)} \cong \texttt{pow(n)}.$$

Proof

```
square(exp(n div 2))
≅
(exp(n div 2))* (exp(n div 2))
```

```
(fn x => x * x)(exp(n div 2))
≅
(exp(n div 2))* (exp(n div 2))
```

Lemma 5? Prop. 1?

Key Point:
Valuable Stepping

Valuable-Stepping Principle

Principle If `e2` is a valuable expression, then

$$\texttt{(fn x => e1) e2} \quad \cong \quad \texttt{[e2/x] e1}$$

Notes:

- It's $\cong$, **not** $\Longrightarrow$! This is only an evaluation step if `e2` is a *value* (eagerness).
- This equivalence often holds even if `e2` is *not* valuable, but that requires careful analysis of `e1`. Sometimes it doesn't hold, though. Consider

  e1 : `(exp ~1, x)`

  e2 : `1 div 0`

- This equivalence can also be broken (or complicated) if shadowing is taking place, or if `e1` or `e2` is impure. So only use it when those are not an issue.

```
square(exp(n div 2))
```

$$\cong$$

```
(exp(n div 2)) * (exp(n div 2))
```

- Defn of `square`
- Lemma 5: `n div 2` is valuable and nonnegative
- Prop. 1: if `e` valuable and nonnegative, `exp(e)` valuable
- Valuable-Stepping Principle: can substitute valuable expressions into function body as if they were values, and obtain the same thing (up to $\cong$)

5-minute break

1 Faster List Functions

Review: Lists

A lengthy function

`len : int list -> int`
REQUIRES: true
ENSURES: `len L` evaluates to the length of `L`

0604.4 (lists.sml)

```
fun len ([] : int list):int = 0
  | len (x::xs) = 1 + len xs

val 5 = len [1,2,3,4,5]
val 2 = len [~5000,19]
val 0 = len []
```

```
(op @) : int list * int list -> int list
```

REQUIRES: true

ENSURES: If `L1` is a list of length *m* and `L2` is a lsit of length *n*, then `L1@L2` evaluates to a list of length $m + n$ whose first *m* elements are the elements of `L1` (in the same order they appear in `L1`) and whose last *n* elements are the elements of `L2` (in the same order they appear in `L2`)

0604.5 (lists.sml)

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

Ti esrever dna ti pilf

```
rev : int list -> int list
```

REQUIRES: true

ENSURES: `rev L` $\Longrightarrow$ `L'`, where `L'` contains the same elements as `L`, in the opposite order.

0604.6 (lists.sml)

```
fun rev ([]:int list):int list = []
  | rev (x::xs) = (rev xs)@[x]
```

0604.6 (lists.sml)

```
val [] = rev []
val [4,3,2,1] = rev [1,2,3,4]
```

Demonstration: `@` and `rev`

traces

Today's second slogan:

Sometimes the best way to make your life easier is to make your life harder `trev : int list * int list -> int list`

REQUIRES: true

ENSURES: `trev(L,acc)` $\cong$ `L'@acc`, where `L'` contains the same elements as `L`, in the opposite order.

0604.7 (lists.sml)

```
fun trev ([]:int list,acc:int list) = acc
  | trev (x::xs,acc):int list = trev(xs,x::acc)
```

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

0604.7 (lists.sml)

```
fun trev ([]:int list,acc:int list) = acc
  | trev (x::xs,acc):int list = trev(xs,x::acc)
```

Summary

- More complex recursion patterns can be proven correct using strong induction
- Referential transparency means we can swap out code with better implementations
- We can reason about the runtime of functions
- Adding accumulator arguments can facilitate writing better code

Next Time

- Proving stuff about lists
- Precisely reasoning about runtime
- More tail recursion

Thank you!