15-150 Fall 2013

Stephen Brookes

Lecture 12

change

- Generalizing the subset-sum problem
- Developing specs and code together
- A lesson in program design

the problem

- Given an integer n, a list of coins L, and a constraint p, is there a sublist of L that adds up to n and satisfies p?

background

```
fun sublists [ ] = [ [ ] ]
|     sublists (x::R) =
        let val S = sublists R in S @ map (fn L => x::L) S end

fun exists p [ ] = false
|     exists p (x::R) = p(x) orelse exists p R

fun sum L = foldr (op +) 0 L
```

change

A non-recursive function that returns a **boolean**

```
fun slowchange (n, L) p =
      exists (fn A => (sum A = n andalso p A)) (sublists L)
```

slowchange : int * int list -> (int list -> bool) -> bool

REQUIRES p is total

ENSURES slowchange (n, L) p = **true**
iff there is a sublist A of L with
sum A = n and p A = **true**

critique

change (325, [1,2,3,...,25]) (**fn** _ => **true**)

- Very slow!
- Not recursive
- Generate-and-test, brute force

a better spec

change : int * int list -> (int list -> bool) -> bool

REQUIRES p is total, $n \geq 0$, L a list of positive integers

ENSURES change (n, L) p = **true**

if there is a sublist A of L with
sum A = n and p A = **true**

change (n, L) p = **false**, otherwise

a better strategy

- Avoid building the list of sublists
  - *accumulate* a suitable sublist *implicitly*
- Deal with special cases first
  - n=0
  - n > 0, L = [ ]
- For n > 0, L = x::R, use ***recursion***...

the spec suggests this might be possible!

change

A *recursive* function that returns a **boolean**

```
fun change (0, L) p = p [ ]
|    change (n, [ ]) p = false
|   change (n, x::R) p =
        if x <= n
        then (change (n-x, R) (fn A => p(x::A))
                   orelse change (n, R) p)
        else change (n, R) p
```

correctness?

- Use induction on length of L

For all positive integer lists L, all n $\geq$ 0,
and all total functions p : int list -> bool,

change (n, L) p = **true**
if there is a sublist A of L with
sum A = n and p A = **true**

change (n, L) p = **false**, otherwise

examples

change (10, [5,2,5]) (**fn** _ => **true**)
= **true**

change (210, [1,2,3,...,20]) (**fn** _ => **true**)
=>* **true** ***(FAST!)***

change (10, [10,5,2,5]) (**fn** A => length(A)>1)
= **true**

change (10, [10,5,2]) (**fn** A => length(A)>1)
= **false**

boolean blindness

- By returning only a *truth value* we only get a small amount of information (**true** or **false**)
- From the context, we know this tells us "if it is *possible* to make change..."
- But what if we want *more* information?
    - "a *way* to make change, if there is one"

mkchange

A recursive function that returns an **(int list) option**

mkchange : int * int list -> (int list -> bool) -> int list option

REQUIRES n >= 0, L a list of positive integers, p is total

ENSURES mkchange (n, L) p = SOME A,
where A is a sublist of L
with sum A = n and p A = **true**,
if there is such a sublist;

mkchange (n, L) p = NONE, otherwise

mkchange

```
fun mkchange (0, L) p =
          if p [ ] then SOME [ ] else NONE
|    mkchange (n, [ ]) p = NONE
|    mkchange (n, x::R) p =
       if x <= n
       then
          case mkchange (n-x, R) (fn A => p(x::A)) of
              SOME A => SOME (x::A)
             | NONE    => mkchange (n, R) p
       else
          mkchange (n, R) p
```

correctness?

- Use induction on length of L

For all positive integer lists L, all n $\geq$ 0,
and all total functions p : int list -> bool,

mkchange (n, L) p = SOME A
where A is a sublist A L with
sum A = n and p A = **true**,
if there is one

mkchange (n, L) p = NONE, otherwise

more generally

- We can easily generalize the spec
  and design a very *flexible* function
      let's go *polymorphic*...

- Instead of returning an ***option*** value,
  and pattern-matching on options,
  assume some type of *answers*
  and function parameters s and k that can
  be used to produce answers in *successful*
  and *unsuccessful* cases

more general spec

```
mkchange2 : int * int list -> (int list -> bool)
                    -> (int list -> 'a) -> (unit -> 'a) -> 'a
```

REQUIRES n>=0, L a list of positive integers, p total

ENSURES mkchange2 (n, L) p s k = s A
where A is a sublist of L such that
sum A = n and p A = **true**,
if there is one

mkchange2 (n, L) p s k = k ( )
otherwise

mkchange2

```
fun mkchange2 (0, L) p s k =
          if p [ ] then s [ ] else k( )
|    mkchange2 (n, [ ]) p s k = k( )
|    mkchange2 (n, x::R) p s k =
       if x <= n
       then
          mkchange2 (n-x, R)
           (fn A => p(x::A))
             (fn A => s(x::A))
               (fn ( ) => mkchange2 (n, R) p s k)
       else
           mkchange2 (n, R) p s k
```

correctness?

- Use induction on length of L

For all positive integer lists L, all n $\geq$ 0,
all total functions p : int list -> bool,
all types t and all s : int list -> t, k : unit -> t

mkchange2 (n, L) p s k = s A
where A is a sublist A L with
sum A = n and p A = **true**,
if there is one

mkchange2 (n, L) p s k = k( ), otherwise

utility

```
fun change (n, L) p =
   mkchange2 (n, L) p (fn _ => true) (fn ( ) => false)
```

```
fun mkchange (n, L) p =
   mkchange2 (n, L) p SOME (fn ( ) => NONE)
```

comments

mkchange2 (n, L) p s k

- The type of mkchange2 is

```
int * int list -> (int list -> bool) -> (int list -> 'a) -> (unit -> 'a) -> 'a
 n       L                p                       s                   k
```

- s is a *success continuation*

- k is a *failure continuation*

coming soon

- Using *continuation-style* programs to solve problems that require ***backtracking***