today

• benign and beneficial effects

• **persistent** vs. **ephemeral** interfaces

• memoizing a function

• graph searching, revisited

imperative vs. functional
fast reversal

fun fastrev (L : 'a list) : 'a list =
  let
    val R = ref [ ]
    fun loop [ ] = !R
      | loop (x::xs) = (R := x :: (!R); loop xs)
  in
    loop L
  end
fast reversal

- For all types `t` and all values `L : t list`, `fastrev L = rev L`
- `fastrev L` causes no state change

\[
\text{fastrev} \\
\text{is extensionally equal to} \\
\text{the pure function } \text{rev}
\]

\[
\text{work/span for fastrev } L \\
\text{is } O(\text{length } L)
\]
benign effects

- **fastrev** uses effects in a *benign* way
  - only affecting *local* state
  - the local state isn’t visible to users
  - **fastrev L** always returns the *same* result and has no side effect

- \((\text{fastrev } [1,2], \text{fastrev } [1,2])\);
  \[\text{val it} = ([2,1], [2,1]) : \text{int list * int list}\]
benign effects
in parallel

fun revALL (S: 'a list seq) : 'a list seq
  = Seq.map fastrev S

\[
\text{revALL } \langle L_1, ..., L_n \rangle = \langle \text{fastrev}(L_1), ..., \text{fastrev}(L_n) \rangle \\
= \langle \text{rev}(L_1), ..., \text{rev}(L_n) \rangle
\]

work $O(\text{length } L_1 + ... + \text{length } L_n)$
span $O(\max\{\text{length } L_1, ..., \text{length } L_n\})$

Evaluations of $\text{fastrev}(L_1) ... \text{fastrev}(L_n)$ don't interfere
beneficial

• Using an effect to ensure correctness, e.g. to avoid non-termination

• Example: graph reachability

```ml
type 'a graph = 'a -> 'a list;

fun G 1 = [2, 3]
  | G 2 = [1, 3]
  | G 3 = [4]
  | G _ = []
```

Nodes reachable from 1: 1, 2, 3, 4
Nodes reachable from 3: 3, 4
fun reachable (G:"a graph) (x:"a, y:"a) : bool =
let
  fun dfs (n:"a) : bool =
    (n=y) orelse (exists dfs (G n))
in
  dfs x
end;

reachable G (1, 2) = true
reachable G (1, 4) = ??? loops forever, should be true
reachable G (1, 5) = ??? loops forever, should be false
fun exists (p : 'a -> bool) (L : 'a list) : bool =
  case L of
    [] => false
  | (x::R) => (p x) orelse (exists p R)

exists p [x_1, ..., x_n] = true
if and only if
p x_i = true for some i
lesson

• Must keep track of the nodes already visited

  Don’t search any further from a node already visited... avoid cycles

• Calls to dfs should maintain a mutable list of the nodes already visited

• Design code so that the effects of recursive calls to dfs help prevent looping
fun reachable (G: "a graph) (x: "a, y: "a) : bool =
  let
    val visited = ref []
    fun dfs (n: "a) : bool =
      (n = y) orelse
      let
        val V = !visited
      in
        if mem n V then false else
        (visited := n::V; exists dfs (G n))
      end
    in
    dfs x
  end
fun reachable (G:"a graph) (x:"a, y:"a) : bool =
  let
    val visited = ref [ ]
    fun dfs (n:"a) : bool =
      (n = y) orelse
      let
        val V = !visited
      in
        if mem n V then false else
        (visited := n::V; exists dfs (G n))
      end
  in
dfs x
end

reachable G (x, y)
creates a fresh visited cell
and a local function dfs that
knows about G, y, visited
then calls
dfs x

dfs n
- if n=y, stop and return true; otherwise
- if n is in !visited, stop and return false; otherwise
- insert n into visited list then
call dfs sequentially on list G n of successors

updates of recursive calls affect later calls
results

fun G 1 = [2,3]
  | G 2 = [3,1]
  | G 3 = [4]
  | G _ = []

reachable G (1,2) = true
reachable G (1,4) = true
reachable G (1,5) = false
specification

Assuming $G$ is **finite**,

- **reachable** $G(x, y)$ returns **true** if there is a path in $G$ from $x$ to $y$
- **reachable** $G(x, y)$ returns **false** otherwise
- **reachable** $G(x, y)$ has no side-effect

$G : t \rightarrow t$ list is **finite** iff $G$ is total &

\{$n : t \mid G(n) \neq \[ \]$\} is finite
specification

assuming \( G : t \rightarrow t\) \(\) list \(\) is finite

For the \(dfs\) function used by \(\) reachable \(G (x, y)\)

- For all values \(V : t\) \(\) list \(\) and \(n : t\)

  \(dfs\) \(n\) from a state with \(\) !visited = \(V\)

  returns \(true\)

  if there is a path in \(G\) from \(n\) to \(y\) avoiding \(V\)

  returns \(false\)

  otherwise

  and updates \(!visited\)\ldots
reachable $G(1, 4)$

$\Rightarrow^*$ dfs 1 $\{!\text{visited} = [ ]\}$

$\Rightarrow^*$ true $\{!\text{visited} = [3,2,1]\}$

$\{!\text{visited} = [ ]\}$ dfs 1 $\Rightarrow^*$ exists dfs [2,3] $\{!\text{visited} = [1]\}$

$\{!\text{visited} = [1]\}$ dfs 2 $\Rightarrow^*$ exists dfs [1,3] $\{!\text{visited} = [2,1]\}$

$\{!\text{visited} = [2,1]\}$ dfs 1 $\Rightarrow^*$ false $\{!\text{visited} = [2,1]\}$

$\{!\text{visited} = [2,1]\}$ dfs 3 $\Rightarrow^*$ true $\{!\text{visited} = [3,2,1]\}$
Although visited is private, so not “visible”, this is what happens.

reachable G (1,2) =>* true
reachable G (1,4) =>* true
reachable G (1,5) =>* false
work and span

• Let $G$ be a finite graph with $n$ nodes, $e$ edges
• Give a big-O estimate of the work for reachable $G(x, y)$

  dfs $x$

How about span?
parallel?

Let $G \times = [y_1, ..., y_k]$

- $dfs\ x$ calls exists $dfs\ [y_1, ..., y_k]$, which calls $dfs\ y_1, dfs\ y_2, ...$ \textit{sequentially}

- Not possible to make these calls in \textit{parallel}

  - we’d get \textit{race conditions} with multiple simultaneous updates to \textit{visited}

  - $dfs\ 2$ interferes with $dfs\ 3$
In our \textit{purely functional} implementation of the GAME signature, data is \textit{persistent}.

\item \texttt{step(s, m)} builds a \textit{new} game state and the old state \texttt{s persists}

\begin{align*}
\text{step : state} \times \text{move} & \rightarrow \text{state} \\
\text{val } s_1 &= 15 \\
\text{val } s_2 &= \text{step}(s_1, 3) \\
& (\star s_1 = 15, s_2 = 12 \star)
\end{align*}
• Using imperative features we can build implementations of GAME in which the game state is mutable, or ephemeral.

   \text{step : state } \times \text{move } \rightarrow \text{unit}

• \text{step(s, m) updates the state s} and destroys the old contents

\begin{align*}
\{!s = 15\} & \xrightarrow{\text{step(s, 3)}} \* (\) \{!s = 12\}
\end{align*}

Nim
contrast

• In a **persistent** implementation *multiple* versions of data may be accessible *simultaneously*
  
  • *updates* don’t destroy, they build new versions, and the old ones can still be used

• In an **ephemeral** implementation *a single version* of the data is *available* at each stage
  
  • *updates* destroy the old version
signature GAME =
  sig
    type state
    type move
    val start : state
    val step : state * move -> state
  ...
end

persistence is implied by type of step
Nim
(persistent implementation)

structure Nim : GAME =
  struct
    type state = int
    type move = int
    fun step (s, n) =
      if (s >= n) then s - n
      else ...
    ...
  end
functor MaxiMe (G : GAME) : PLAYER =
struct
    structure Game = G;
    open Game;
    ...
    fun F (s : state) : int = 
        let
            val M = moves s
        in
            if (null M) then (score s) else
                reduce1 Int.max
                (Seq.map (fn m => (........)) M)
        end
    and G (s : state) : int = 
        ...
end
minimax Nim
(persistent version)

\[ F_3 = \text{reduce1 \: Int.max} \: \langle G_0, G_1, G_2 \rangle \]

\[ G_0 = \sim 1 \]
\[ G_1 = 1 \]
\[ G_2 = \sim 1 \]

- calls to \( G \) are independent
- can be done in parallel
- order doesn’t affect result of \( F_3 \)

\[ F_3 = 1 \]
signature EPH_GAME =
  sig
    type state
    type move
    val start : state
    val step : state * move -> unit
  ...
end

ephemerality is implied by type of step
ephemeral Nim

structure Eph_Nim : EPH_GAME =
struct
  type state = int ref
  type move = int
  fun step (s, n) =
    if (!s >= n) then s := !s - n
    else ...
end
functor Eph_MaxiMe (Game : EPH_GAME) : EPH_PLAYER =
struct
  structure Game = Game;
  open Game;
...
fun F (s : state) : int =
  let
    val M = moves s
  in
    if (null M) then (score s) else
      reduce1 Int.max
      (Seq.map (fn m => (..........)) M)
  end;

and G (s : state) : int =
...
end;

OOPS!!
step(s, m) changes !s, so later calls to G don’t see the original value
minimax Nim
(ephemeral version)

\{!s = 3\}

F s =>*
reduce1 Int.max
\langle(s := !s - 3; G s), (s := !s - 2; G s), (s := !s - 1; G s)\rangle

- calls to G *interfere*... they all read and write \( s \)
- the overall *effect* depends on *execution order*

This isn’t a good way to implement minimax!
signature EPH_GAME =
  sig
    type state
    type move
    val start : state
    val step : state * move -> unit
    val undo_step : state * move -> unit
    ...
  end

for pragmatic reasons we add undo_step
ephemeral Nim

structure Eph_Nim : EPH_GAME =

struct
  type state = int ref
  type move = int
  fun step (s, n) =
    if (!s >= n) then s := !s - n
    else ...
  fun undo_step (s, n) =
    s := !s + n
  ...
end
functor Eph_MaxiMe (Game : EPH_GAME) : EPH_PLAYER =

struct
    structure Game = Game;
    open Game;
    ...
    fun F (s : state) : int =
        let
            val M = moves s
        in
            if (null M) then (score s) else
                reduce1 Int.max
                (Seq.map (fn m => (...........)) M)
        end

    and G (s : state) : int =
        ...
end;

step(s,m);
let
    val v = G(s)
in
    undo_step(s,m);
    v
end
sequential vs. parallel

- Even with these "improvements" it’s not safe to execute the calls to \( G \) in parallel...
  - possible race conditions
  - for correct move assessment, each thread should finish its "undo" code before another thread accesses the state

some effectful code is safe only in \texttt{sequential} contexts
functor Eph_MaxiMe (Game : EPH_GAME) : EPH_PLAYER =
struct
  structure Game = Game;
  open Game;
...
  fun F (s : state) : int = let
    val M = toList(moves s) in
    if (List.null M) then (score s) else
      foldr Int.max (List.map (fn m => (........)) M)
    end
  and G (s : state) : int = ...
end;
results

(sequential version)

structure P = Eph_MaxiMe(Eph_Nim)
val s = ref 15

- P.player s;
val it = 2 : move  (picks the true best move)

- !s;
val it = 15 : int  (and resets the state)
And now...

- Memoizing a function
- Using a lookup table to store and re-use prior results
Tables

define type ('a, 'b) table = ('a * 'b) list;

define exception NotFound;

fun lookup x [ ] = raise NotFound
| lookup x ((a,b)::L) = if x = a then b else lookup x L;

NotFoundException : exn

lookup : "a -> ("a * 'b) table -> 'b
```plaintext
fun memoize f = 
  let
    val table = ref []
  in
    fn x => ( lookup x (!table)
        handle NotFound =>
          let
            val y = f x
          in
            table := (x,y)::(!table);
            y
          end
    end
end
```

`memoize f` evaluates to a function that behaves like `f` extensionally, but stores argument-result combination when called.
memoizing fib

fun fib 0 = 1
| fib 1 = 1
| fib n = fib(n-1) + fib(n-2);

val memfib = memoize fib

- val f = memoize fib;
val f = fn : int -> int

- val g = memoize fib;
val g = fn : int -> int

- f 42; (* SLOW *)
val it = 433494437 : int

- f 42; (* FAST *)
val it = 433494437 : int

- memfib 42; (* SLOW *)
val it = 433494437 : int

- memfib 41; (* SLOW *)
val it = 267914296 : int

- memfib 42; (* FAST *)
val it = 433494437 : int

- g 42; (* SLOW *)
val it = 433494437 : int
issues

• This memoize function is “generic” —
does the same thing regardless of how f is defined
  • calling with x inserts (x, f(x)) in the table
  • a later call with x will get the answer quickly
• If f is recursive (like fib) we should be able to
  memoize the results of recursive calls made by f(x) and avoid recomputing them…
local

val tab = ref [(1,1), (0,1)]
in

fun fastfib n =
  lookup n (!tab)
  handle NotFound =>
  let val m = fastfib(n-1) + fastfib(n-2)
  in
  tab := (n,m)::(!tab);
  m
end

end
examples

- fastfib 43;
  val it = 701408733 : int
- fastfib 43;
  val it = 701408733 : int
- fastfib 42;
  val it = 433494437 : int
  ... FAST
  ... FAST
fun badfib n =
let
  val tab = ref [(1,1), (0,1)]
in
  lookup n (!tab)
  handle NotFound =>
    let val m = badfib(n-1) + badfib(n-2)
in
    tab := (n,m)::(!tab);
  m
end
end

every call to badfib creates a fresh table
examples

- fib 43;
  val it = 701408733 : int

- badfib 43;
  val it = 701408733 : int

- badfib 43;
  val it = 701408733 : int

... SLOW

... SLOW

... SLOW AGAIN!!
story so far

• We specify and prove correctness using equational reasoning and induction

• For pure functional programs, only concerned with value

• For imperative programs, need to consider value and side-effect

• Pay attention to scope!

WARNING

equations that hold for pure ML may fail for imperative programs
back to graphs

- Now let’s contrast *imperative* and *functional* styles
- We talked about (depth-first) graph reachability
- We showed how to deal with cycles (in finite graphs) by using imperative features
fun reachable (G:"a graph) (x:"a, y:"a) : bool =
  let
    fun dfs (n:"a) : bool =
      (n = y) orelse exists dfs (G n)
  in
    dfs x
  end

Purely functional, but loops if there’s a cycle
second try

```ml
fun reachable (G:"a graph) (x:"a, y:"a) : bool = let
  val visited = ref [ ]
  fun dfs (n:"a) : bool = (n = y) orelse
    let
      val V = !visited
    in
      if mem n V then false else
        (visited := n::V; exists dfs (G n))
    end
  in
    dfs x end

Not purely functional,
but OK even if there’s a cycle
```
fun reachable (G:"a graph) (x:"a, y:"a) : bool = 
  let 
    fun dfs (V: "a list) (n:"a) : bool = 
      (n = y) orelse 
      (if mem n V then false else 
       exists (dfs (n::V)) (G n)) 
    in 
    dfs [ ] x 
  end

Purely functional, 
OK even if there’s a cycle
assessment

• *Imperative* style can be natural when we need to *communicate* information between function calls

• But the same results *may* be achievable in pure *functional* style, by using an extra argument to *propagate* information