Imperative Programming

15-150

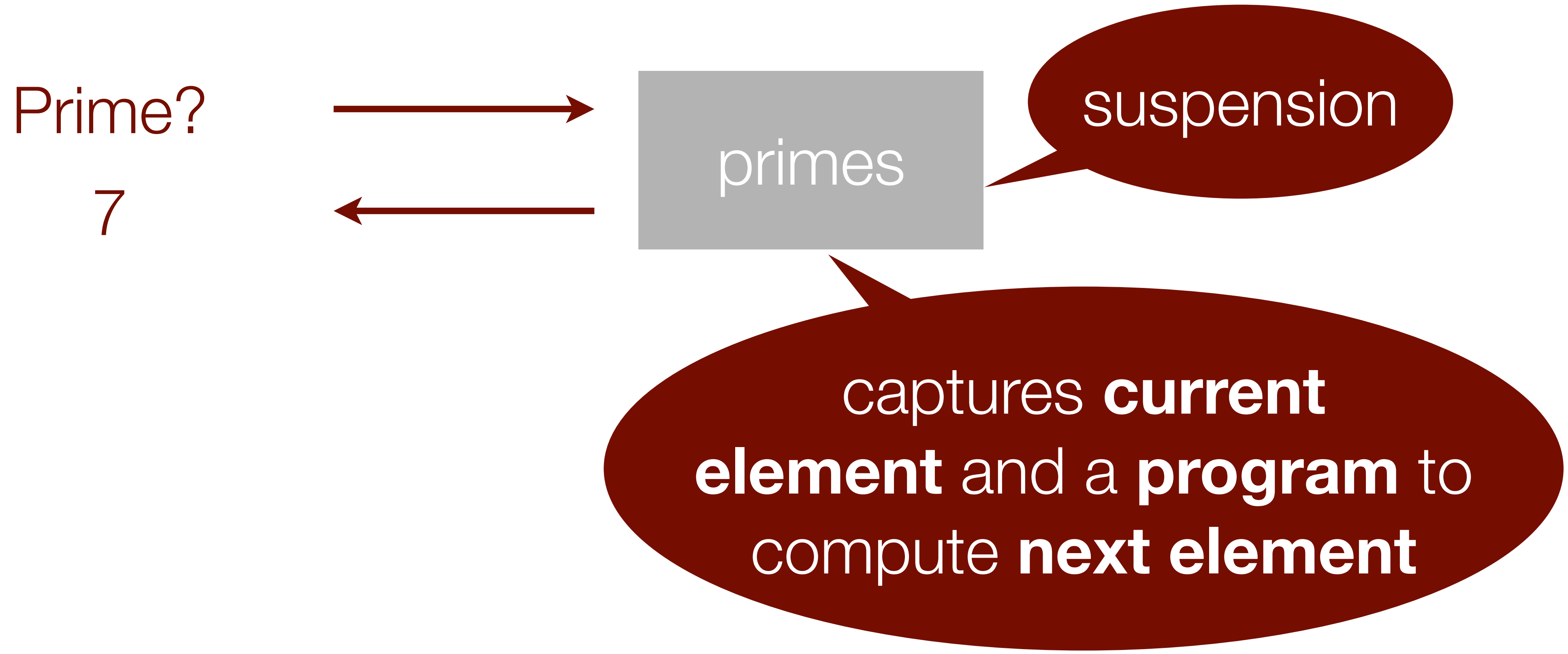
Lecture 21: November 20, 2025

Stephanie Balzer
Carnegie Mellon University

Let's first continue with streams

Streams*

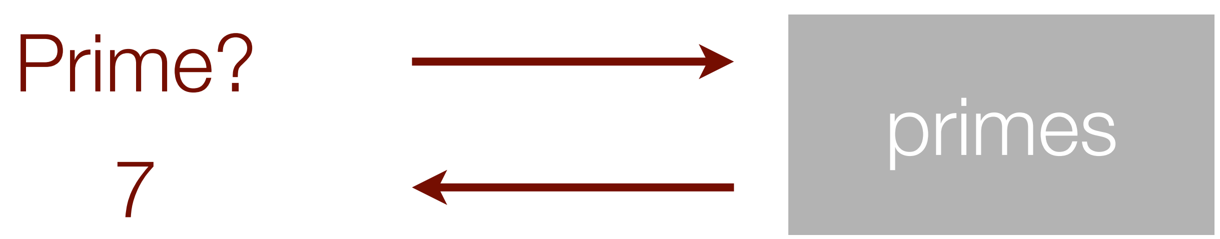
Streams are data structures that are being continuously created, e.g.,



* (Note, different from SML's built-in I/O streams.)

Streams

Streams are data structures that are being continuously created, e.g.,



- We can think of streams as being generated by state machines:
  - only when "kicked" (forcing suspension) they yield element
  - advancing state for computation of next element.
- Streams are defined **coinductively**.

* (Note, different from SML's built-in I/O streams.)

Stream signature

```
signature STREAM =
sig
  type 'a stream                          (* abstract *)

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty               (* concrete *)

  val expose : 'a stream -> 'a front

  val delay : (unit -> 'a front) -> 'a stream

  (* more functions (see accompanying code) *)
end
```

Stream structure

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Cons of 'a * 'a stream | Empty

  (* delay : (unit -> 'a front) -> 'a stream *)
  fun delay (d) = Stream(d)

  (* expose : 'a stream -> 'a front *)
  fun expose (Stream(d)) = d ()

  (* more functions (see accompanying code) *)
end
```

Let's practice: stream of nats

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

Let's implement an infinite stream of all natural numbers:

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))

(* int S.stream *)
val nats = S.delay (nat' 0)
```

initial element

Recall:
```
(* delay : (unit -> 'a front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of nats

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

Let's implement an infinite stream of all natural numbers:

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))

(* int S.stream *)
val nats = S.delay (nat' 0)
```

current element (x)

Recall:
```
(* delay : (unit -> 'a front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of nats

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

➡ **Let's implement an infinite stream of all natural numbers:**

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))

(* int S.stream *)
val nats = S.delay (nat' 0)
```

current element

next element

Recall:

```
(* delay : (unit -> 'a front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

`2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...`

Write down all the natural numbers greater than `1`.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

`2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...`

Find leftmost element (`2` currently).

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

`2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...`

Cross off all multiples of that leftmost element.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

2,3,~~4~~,5,~~6~~,7,~~8~~,9,~~10~~,11,~~12~~,13,~~14~~,15,~~16~~,17,~~18~~,...

3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17,...

Repeat the process with the remaining numbers.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, ...

3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, ...

5, 7, 11, 13, 17, ...

Keep repeating this process.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, ...

3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, ...

5, 7, 11, 13, 17, ...

The diagonal of leftmost elements constitutes all primes.

Another example: prime numbers

To implement this algorithm, we augment our signature with the following function:

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
```

Moreover, we define locally, the following helper function:

```
val notDivides p q = (q mod p <> 0)
```

returns false if q is a multiple of p

otherwise true

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

```
fun sieve s = S.delay (fn () => sieve' (S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
      S.Cons(p, sieve (S.filter (notDivides p) s))
```

```
val primes = sieve (S.delay (nat' 2))
```

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

delays actual sieving

Now, the algorithm:

```
fun sieve s = S.delay (fn () => sieve' (S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
      S.Cons(p, sieve (S.filter (notDivides p) s))

val primes = sieve (S.delay (nat' 2))
```

Recall:

```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

not really needed because primes are infinite

```
fun sieve s = S.delay (fn () => si[illegible]ose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
      S.Cons(p, sieve (S.filter (notDivides p) s))

val primes = sieve (S.delay (nat' 2))
```

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

```
fun sieve s = S.delay (fn () => sieve' (S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
      S.Cons(p, sieve (S.filter (notDivides p) s))

val primes = sieve (S.delay (nat'
```

filters multiples of current element p

recursively constructs stream of larger primes, with p at front

Recall: `(* delay ... -> ... *)`
`fun delay`

Imperative programming

Functional programming

So far we have used the term "functional programming" as a synonym for pure programming.

- But what does **pure** really mean?
- Well, the prototypical answer is, **without** any **side-effects**.
- But what does that really mean? 🤔

Functional programming

So far we have used the term "functional programming" as a synonym for pure programming.

- But what does **pure** really mean?
- Well, the prototypical answer is, **without** any **side-effects**.

Let's reconsider the correctness proofs that we carried out.

- Can you think of an implicit assumption that we made when proving a function correct, ensuring that our reasoning is valid?
- We assumed that it suffices to *only* consider the function specification and implementation, *nothing else*.
- We carried out per-function (aka **local**) **reasoning**.

Functional programming

Let's reconsider the correctness proofs that we carried out.

- Can you think of an implicit assumption that we made when proving a function correct, ensuring that our reasoning is valid?
- We assumed that it suffices to *only* consider the function specification and implementation, *nothing else*.
- We carried out per-function (aka **local**) **reasoning**.

Functional programming validates local reasoning and guarantees that:

- Repeated evaluation of an expression yields the same result.
- Sequential and parallel evaluation of independent sub-expressions produces the same result.

Effects (impure or imperative programming)

In the presence of effects, local reasoning* breaks down.

→ Effect, aka anything else that we can observe when evaluating an expression other than the returned value.

Examples of effects:

- When two functions share state, mutations by one affect the other.
- A non-terminating function will cause its caller to diverge too.

In the presence of effects, the **order of evaluation** matters.

→ Repeated evaluation of an expression may not yield the same result.

→ Sequential and parallel evaluation of independent sub-expressions may not produce the same result.

*(Local reasoning can be re-established by using program logics such as separation logic.)

SML supports imperative programming

To reap all the benefits of functional programming, we have stayed entirely* in the pure fragment of SML until now.

However, SML supports imperative features, such as reference cells, arrays, and commands for I/O.

- We may use effects locally to increase efficiency, for example.
  - referred to as "benign effects"
- Expressions that engender effects typically are of `unit` type.

*(Except for non-termination and exceptions.)

Today's menu

- Shared state through mutable reference cells
  - reference type
  - typing and evaluation rules
- Aliasing
- Race conditions
- Persistent versus ephemeral data
- Examples of benign effects

Mutable reference cells

Reference type: `t ref`

arbitrary SML type*

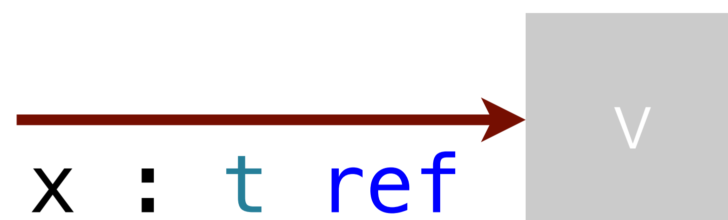
including t ref

*(Restriction: at top level, t must be monomorphic.)

Mutable reference cells

Reference type: `t ref`

Reference type values:



(where `v : t`)

➡ The type `t ref` represents mutable reference cells that store a value of type `t`.

Functions:

```
ref : 'a -> 'a ref            allocation
!   : 'a ref -> 'a            read
:=  : 'a ref * 'a -> unit     write
```

*(Restriction: at top level, t must be monomorphic.)

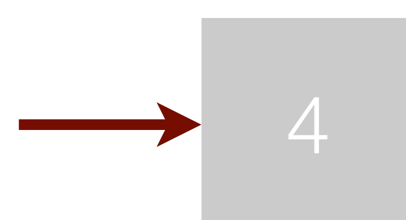
Allocation: `ref: 'a -> 'a ref`

Evaluation rules: `ref e`

1. Evaluate expression **e**.
2. If **e** reduces to a value **v**, create a new cell containing **v** and return the reference to it.

Example: `val r = ref (1 + 3)`

evaluates to: `r`



Here, `r : int ref` is bound to a reference to the reference cell containing the value `4 : int`.

Allocation: `ref: 'a -> 'a ref`

Evaluation rules: `ref e`

1. Evaluate expression **e**.
2. If **e** reduces to a value **v**, create a new cell containing **v** and return the reference to it.

Typing rules: `ref e`

➡ If `e : t`, then `ref e : t ref`.

Read: `!`: 'a ref -> 'a

Evaluation rules: `!e`

1. Evaluate expression `e`.
2. If `e` reduces to reference to a cell containing `v`, then return `v`.

Example:

```
val r = ref (1 + 3)
val x = !r
```

evaluates to: `r` → `4` and `[4/x]`

Here, `r : int ref` is bound to a reference to the cell containing the value `4 : int` and `x : int` is bound to `4`.

Read: `!:` 'a ref -> 'a

Evaluation rules: `!e`

1. Evaluate expression **`e`**.
2. If **`e`** reduces to reference to a cell containing **`v`**, then return **`v`**.

Typing rules: `!e`

→ If `e : t ref`, then `!e : t`.

Write: `:= : 'a ref * 'a -> unit`

Evaluation rules: e_1 `:=` e_2

1. Evaluate expression e_1.
2. If e_1 reduces to a reference `r`, then evaluate expression e_2.
3. If e_2 reduces to a value `v`, update contents of `r` to `v`, return `()`.

Example:

```
val r = ref (1 + 3)
r := (!r * 2)
```

evaluates to: `r` → 8 and `[()/it]`

Here, `r : int ref` is bound to a reference to the cell containing the value `8 : int` and `()` is returned.

Write: `:= : 'a ref * 'a -> unit`

Evaluation rules: e_1 `:=` e_2

1. Evaluate expression e_1.
2. If e_1 reduces to a reference `r`, then evaluate expression e_2.
3. If e_2 reduces to a value `v`, update contents of `r` to `v`, return `()`.

Typing rules: e_1 `:=` e_2

→ If e_1 `: t ref` and e_2 `: t`, then e_1 `:=` e_2 `: unit`.

Reference cells support pattern matching

We can pattern match on `ref`:

```
(* containsZero : int ref -> bool *)

fun containsZero (ref 0) = true
  | containsZero _ = false
```

pattern

Reference cells support pattern matching

We can pattern match on `ref`:

```
(* containsZero : int ref -> bool *)

fun containsZero (ref 0) = true
  | containsZero _ = false
```

Reference cells support pattern matching

We can pattern match on **`ref`**:

```
(* containsZero : int ref -> bool *)

fun containsZero (ref 0) = true
  | containsZero _ = false

val d = ref 42

val false = containsZero d
```

Reference cells support pattern matching

We can pattern match on `ref`:

```
(* containsZero : int ref -> bool *)

fun containsZero (ref 0) = true
  | containsZero _ = false

val d = ref 42

val false = containsZero d

val false = containsZero (ref 7)
```

Reference cells support pattern matching

We can pattern match on `ref`:

```
(* containsZero : int ref -> bool *)

fun containsZero (ref 0) = true
  | containsZero _ = false

val d = ref 42

val false = containsZero d

val false = containsZero (ref 7)

val true = containsZeros (ref 0)
```

Sequential composition

In the presence of effects, the **order of evaluation** matters.

For convenience, SML supports the semicolon expression:

$$(e_1;\ e_2)$$

Which is syntactic sugar for:

$$\texttt{let val \_ = } e_1 \texttt{ in } e_2 \texttt{ end}$$

1. Evaluate e_1, executing effects but ignoring any returned value.
2. Then, evaluate e_2, executing effects and return the value of e_2.

Generalizes to:

$$(e_1;\ e_2;\ \ldots;\ e_n) : t_n$$

Sequential composition

Example:

```
let
  val c = ref 10
in
  (print(Int.toString(!c));
   c)
end
```

What is the type of this `let` expression? `int ref`

What is its value? `ref 10`

What its effect? prints 10

Sequential composition

Alternative implementation of previous example:

```
let
  val c = ref 10
  val _ = print(Int.toString(!c))
in
  c
end
```

Aliasing

Consider this code:

```
val c = ref 10
val w = !c
val d = c
val () = d := 42
val v = !c
```

d is now referring to the same cell as c

assignment to d affects what can be read from c

What values are w and v bound to?

w is bound to 10, v is bound to 42.

➡ To account for aliasing, we must extend dynamics with a store.

Aliasing

To account for aliasing, we must extend dynamics with a store.

For pure expressions:

```
e ==> e'
```

For impure expressions:

```
{s | e} ==> {s' | e'}
```

store,
i.e., all allocated
reference cells

evaluation may alter
the store!

Aliasing

- To account for aliasing, we must extend dynamics with a store.

For pure expressions:

```
e ==> e'
```

For impure expressions:

```
{s | e} ==> {s' | e'}
```

- We won't go into any further details in 15-150.
  - More on this in 15-312!
- Note: aliasing complicates reasoning about programs 😢

Extensional equivalence

For pure programs:

- extensional equivalence as defined until now
- allow equals to be replaced by equals ("referential transparency")

For imperative programs:

- extensional equivalence must additionally account for the store
  - requires advanced program logics (even beyond 15-312)
- For pure expressions e and e', to show e $\cong$ e', we must show that e and e' are independent of any store.

Extensional equivalence

For imperative programs:

- extensional equivalence must additionally account for the store
  - requires advanced program logics (even beyond 15-312)
- For pure expressions e and e', to show e $\cong$ e', we must show that e and e' are independent of any store.

Note:

- `ref` types are so called equality types

For `r : 'a ref` and `s : 'a ref`, `r = s` evaluates to `true`, if `r` and `s` are aliases, i.e., point to the same cell.

Race conditions

In the presence of mutation, reasoning about parallel program becomes complicated.

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n
val chk = ref 100
val _ = (deposit chk 50; withdraw chk 80)
```

What is the value of !`chk`?

```
70
```

Now, if we parallelize, what is the value of !`chk`?

Race conditions

In the presence of mutation, reasoning about parallel program becomes complicated.

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n
val chk = ref 100
val _ = (deposit chk 50, withdraw chk 80)
```

What is the value of !chk?

```
70
```

Now, if we parallelize, what is the value of !chk?

Race conditions

In the presence of mutation, reasoning about parallel program becomes complicated.

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n
val chk = ref 100
val _ = (deposit chk 50, withdraw chk 80)
```

What is the value of !`chk`?

`70`

Now, if we parallelize, what is the value of !`chk`?

We could end up with `20`, `70`, or `150`.

Race conditions

In the presence of mutation, reasoning about parallel program becomes complicated.

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n
val chk = ref 100
val _ = (deposit chk 50, withdraw chk 80)
```

Mutation and parallelism leads to non-deterministic outcomes 😢

Persistent versus ephemeral data

Pure programs:

- yield persistent data structures
- facilitate reasoning and support deterministic parallelism

Imperative programs:

- yield ephemeral data structures
- complicate reasoning and demand concurrent scheduling

However, not all effects are evil.

- When employed locally, effects can be **benign**.

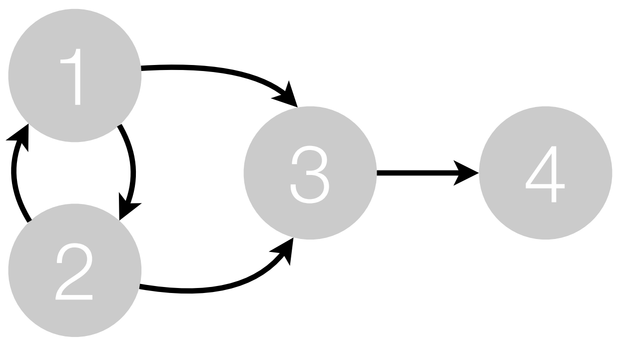
Benign effects

A **benign effect** is an effect (such as mutation) that is **localized** within some sufficiently small chunk of code (e.g., function or structure) so that external users can use the code as **if it were purely functional**.

- Benign effects can be useful, for instance, in improving efficiency.
- Because effect is local, local reasoning remains intact.
- Let's look at some examples!

Example: graph reachability

Consider this directed graph:



We can represent this graph as a function, giving for a node the nodes immediately reachable from it:

```
type  graph = int -> int list

val G : graph = fn 1 => [2,3]
                 | 2 => [1,3]
                 | 3 => [4]
                 | _ => [ ]
```

Example: graph reachability

Now, let's define a function, **`reach g (x,y)`**, determining whether **`y`** is transitively reachable from **`x`** in graph **`g`**.

```
fun reach (g:graph) (x:int, y:int) : bool =
  let
    fun dfs n = (n=y) orelse (List.exists dfs (g n))
  in
    dfs x
  end
```

did we reach **`y`**?

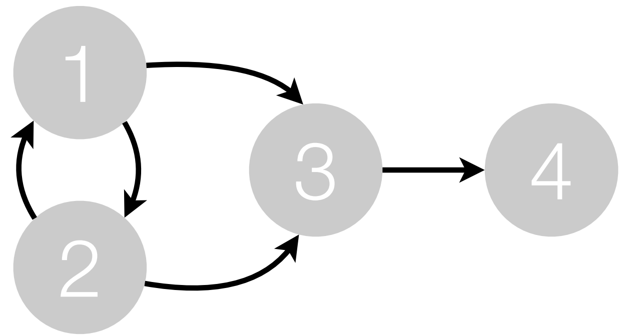
neighbors of **`n`**

Example: graph reachability

Now, let's define a function, **`reach g (x,y)`**, determining whether **`y`** is transitively reachable from **`x`** in graph **`g`**.

```
fun reach (g:graph) (x:int, y:int) : bool =
  let
    fun dfs n = (n=y) orelse (List.exists dfs (g n))
  in
    dfs x
  end
```

➡ Problem: reach can loop in our example graph, which is cyclic!



Example: graph reachability

We can fix this by recording who we have already visited.

```
fun mem (n:int) = List.exists (fn x => n=x)
```

mem n L checks whether n is in list L

```
fun reachable (g:graph) (x:int, y:int)
  let
    val visited = ref []
    fun dfs n = (n=y) orelse
                (not (mem n (!visited)) andalso
                (visited := n::(!visited);
                 List.exists dfs (g n)))
   in
      dfs x
   end
```

Example: graph reachability

We can fix this by recording who we have already visited.

```
fun mem (n:int) = List.exists (fn x => n=x)

fun reachable (g:graph) (x:int, y:int) : bool =
  let
    val visited = ref []
    fun dfs n = (n=y) orelse
                (not (mem n (!visited)) andalso
                (visited := n::(!visited);
                 List.exists dfs (g n)))
   in
      dfs x
   end
```

reference that
records visited nodes

Example: graph reachability

We can fix this by recording who we have already visited.

```
fun mem (n:int) = List.exists (fn x => n=x)

fun reachable (g:graph) (x:int, y:int) : bool =
  let
    val visited = ref []
    fun dfs n = (n=y) orelse
                (not (mem n (!visited))) andalso
                (visited := n::(!visited);
                 List.exists dfs (g n)))
   in
     dfs x
   end
```

only continue if **n** has not yet been visited

Example: graph reachability

We can fix this by recording who we have already visited.

```
fun mem (n:int) = List.exists (fn x => n=x)

fun reachable (g:graph) (x:int, y:int) : bool =
  let
    val visited = ref []
    fun dfs n = (n=y) orelse
                (not (mem n (!visited)) andalso
                (visited := n::(!visited);
                 List.exists dfs (g n)))
     in
       dfs x
     end
```

update visited list

Example: random number generator

```
signature RANDOM =
sig
  type gen (*abstract *)
  val init: int -> gen (* REQUIRES: seed > 0 *)
  val random: gen -> int -> int
end
```

bound

pseudo-random
nonnegative integer
less than bound

(Reference: "ML for the Working Programmer" by Paulson. 1996.)

Example: random number generator

```
signature RANDOM =
sig
  type gen (*abstract *)
  val init: int -> gen (* REQUIRES: seed > 0 *)
  val random: gen -> int -> int
end

val G = R.init(12345)
val L = List.tabulate(42,fn _ => R.random G 1000)
```

(Reference: "ML for the Working Programmer" by Paulson. 1996.)

Example: random number generator

```
signature RANDOM =
sig
  type gen (*abstract *)
  val init: int -> gen (* REQUIRES: seed > 0 *)
  val random: gen -> int -> int
end

struct R :> RANDOM
  type gen = real ref
  val a = 16807.0
  val m = 2147483647.0
  fun next r = a * r - m*real(floor(a*r/m))
  val init = ref o real
  fun random g b = (g := next(!g);
                    floor( (!g/m)* (real b)))
end
```

reference cell

(Reference: "ML for the Working Programmer" by Paulson. 1996.)

Example: stream memoization

Previously, we had the following code inside our `Stream` structure:

```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)

(* expose : 'a stream -> 'a front *)
fun expose (Stream(d)) = d ()
```

- Let's add a hidden reference cell that remembers the result of computing `d()`.
- We will will leave `expose` as is, but change `delay`.

Example: stream memoization

Updated function **`delay`**:

```
fun delay (d) =
  let









  in

  end
```

Example: stream memoization

Updated function **`delay`**:

```
fun delay (d) =
  let
    val cell = ref d
```

let's put **d** in a reference cell

Recall:

```
(* expose : 'a stream -> 'a front *)
fun expose (Stream(d)) = d ()
```

```
  in

  end
```

Example: stream memoization

Updated function **delay**:

```
fun delay (d) =
  let
    val cell = ref d
```

let's put **d** in a reference cell

Recall:

```
(* expose : 'a stream -> 'a front *)
fun expose (Stream(d)) = d ()
```

we now need a suspension, when forced, accesses the reference cell and forces the function in the reference cell

```
  in
    Stream (fn () => !cell())
  end
```

Example: stream memoization

Updated function **`delay`**:

```
fun delay (d) =
  let
    val cell = ref d
```

```
  in
    Stream (fn () => !cell())
  end
```

Example: stream memoization

Updated function **delay**:

```
fun delay (d) =
  let
    val cell = ref d
    fun memoFn () =
      let
        val r = d()
      in
        (cell := (fn () => r); r)
      end

  in
    Stream (fn () => !cell())
  end
```

memoFn is a function that computes **d()**, remembers the result **r** in a suspension, puts that suspension in **cell**, and returns **r**.

Example: stream memoization

Updated function **`delay`**:

```
fun delay (d) =
  let
    val cell = ref d
    fun memoFn () =
      let
        val r = d()
      in
        (cell := (fn () => r); r)
      end

  in
    Stream (fn () => !cell())
  end
```

Example: stream memoization

Updated function **delay**:

```
fun delay (d) =
  let
    val cell = ref d
    fun memoFn () =
      let
        val r = d()
      in
        (cell := (fn () => r); r)
      end
    val _ = cell := memoFn
  in
    Stream (fn () => !cell())
  end
```

> we put **memoFn** into **cell**, where it will sit until someone exposes the stream, at which point **memoFn** replaces itself with **fn () => r**.

That's all for today. Have a good weekend!