15-150

Principles of Functional Programming

Slides for Lecture 21

Imperative Programming

April 10, 2025

Michael Erdmann

Lessons:

- Mutation
  - Mutable cells
  - Typing rules
  - Evaluation rules
- Aliasing
- Race Conditions
- Ephemeral Data vs Persistent Data
- Benign Effects

A New Type

The type is written

`t ref`

with `t` any ML type.

A New Type

The type is written

`t ref`

with **`t`** any ML type.

Restriction: at top-level, **`t`** must be monomorphic.

(This is a consequence of SML's "value restriction", designed to avoid bizarre side-effects. We won't discuss details.)

Values

We think of a value of type **`t ref`** as being a *cell* that contains a value **`v`** of type **`t`**:

$$\boxed{\mathtt{v}}$$

E.g, $\boxed{\mathtt{7}}$ is a value of type **`int ref`** containing the value **`7`** of type **`int`**.

(Create such a cell by writing **`ref 7`**.)

Typing and Evaluation

- Expressions involving reference cells have precise type-checking and evaluation rules.
- As always in SML, type-checking happens before evaluation.
- We will discuss evaluation first, since that is a natural way to introduce new constructs involving reference cells. (We assume all expressions are well-typed during evaluation.)

`ref e`

Evaluation rules:

- Evaluate expression **`e`**.
- If **`e`** reduces to a value **`v`**, then create and return a new cell containing **`v`**.

Pictorially: If **`e`** $\hookrightarrow$ **`v`** , then **`ref e`** $\hookrightarrow$ $\boxed{\mathtt{v}}$.

Example:

```
val c = ref 7
```

That creates a binding $\boxed{\mathtt{7}}$**`/c`** .

`!e`

Evaluation rules:

- Evaluate expression **e**.
- If **e** reduces to a cell containing value **v**, then return **v**.

Pictorially: If **e** $\hookrightarrow \boxed{\mathbf{v}}$, then `!e` $\hookrightarrow$ **v**.

Example:

```
val c = ref 7
val v = !c
```

That creates bindings $\boxed{\mathbf{7}}$`/c` and `7/v`.

$\mathbf{e_1 \ := \ e_2}$

Evaluation rules:

- Evaluate expression $\mathbf{e_1}$.
- If $\mathbf{e_1}$ reduces to cell **c**, then evaluate $\mathbf{e_2}$.
- If $\mathbf{e_2}$ reduces to value **v**, then change the contents of **c** to be **v** and return **()**.

observe

$\mathtt{e_1 \;:=\; e_2}$

Evaluation rules:

- Evaluate expression $\mathtt{e_1}$.
- If $\mathtt{e_1}$ reduces to cell **`c`**, then evaluate $\mathtt{e_2}$.
- If $\mathtt{e_2}$ reduces to value **`v`**, then change the contents of **`c`** to be **`v`** and return **`()`**.

Pictorially: If $\mathtt{e_1} \hookrightarrow \boxed{\mathtt{w}}$ (some **`w`**) and if $\mathtt{e_2} \hookrightarrow \mathtt{v}$, then replace **`w`** with **`v`** in the cell above.

Example: **`val c = ref 7`** $\boxed{\mathtt{7}}\,\mathtt{/c}$

$\mathtt{e_1 := e_2}$

Evaluation rules:

- Evaluate expression $\mathtt{e_1}$.
- If $\mathtt{e_1}$ reduces to cell **`c`**, then evaluate $\mathtt{e_2}$.
- If $\mathtt{e_2}$ reduces to value **`v`**, then change the contents of **`c`** to be **`v`** and return **`()`**.

Pictorially: If $\mathtt{e_1} \hookrightarrow \boxed{\mathtt{w}}$ (some **`w`**) and if $\mathtt{e_2} \hookrightarrow \mathtt{v}$, then replace **`w`** with **`v`** in the cell above.

Example:

```
val c = ref 7
val () = c := 4
val v = !c
```

$\boxed{4}/\mathtt{c}$

$4/\mathtt{v}$

Typing Rules

- **`ref e : t ref`** if **`e : t`**.
- **`!e : t`** if **`e : t ref`**.
- **e_1 `:=` e_2 `: unit`**
  if **e_1 `: t ref`**
  and **e_2 `: t`**.

(and so we also have)

- **`ref`** is similar to a constructor.
  It has type **`'a -> 'a ref`**.
- **`! : 'a ref -> 'a`**.
- **`(op :=) : 'a ref * 'a -> unit`**.

Side Comment

There is no explicit “deallocation” of cells.

In practice, a garbage collector reclaims cells once they become inaccessible via any code (e.g., permanently shadowed).

We do not worry about that in this course.

pattern matching

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 = containsZero (ref 0)
```

Aliasing

```
val c = ref 10
val w = !c
val d = c

val () = d := 42
val v = !c
```

To what values are **`w`** and **`v`** bound?

Aliasing

```
val c = ref 10
val w = !c
val d = c

val () = d := 42
val v = !c
```

We say that `c` and `d` are *aliases* for the same cell.

To what values are `w` and `v` bound?

Answer: `10/w` `42/v`

Another Example

```
fun twice (x : 'a) : 'a ref * 'a ref =
    let
        val c = ref x
    in
        (c, c)
    end
val (p, q) = twice 7
val () = p := 2
val (y, z) = (!p, !q)
```

To what values are **`y`** and **`z`** bound?

Another Example

```
fun twice (x : 'a) : 'a ref * 'a ref =
    let
        val c = ref x
    in
        (c, c)
    end
val (p, q) = twice 7
val () = p := 2
val (y, z) = (!p, !q)
```

`p` and `q` are *aliases* for the same cell, so `y` and `z` are both bound to value `2`.

To what values are `y` and `z` bound?

Cells Can Contain Cells

```
val cc = ref (ref 9) : int ref ref
```

$\boxed{\boxed{9}}$ **/cc**

```
val d = ref 5
val (x, y) = (ref d, ref d)
```

$\boxed{5}$ **/d** $\boxed{\boxed{5}}$ **/x** $\boxed{\boxed{5}}$ **/y**

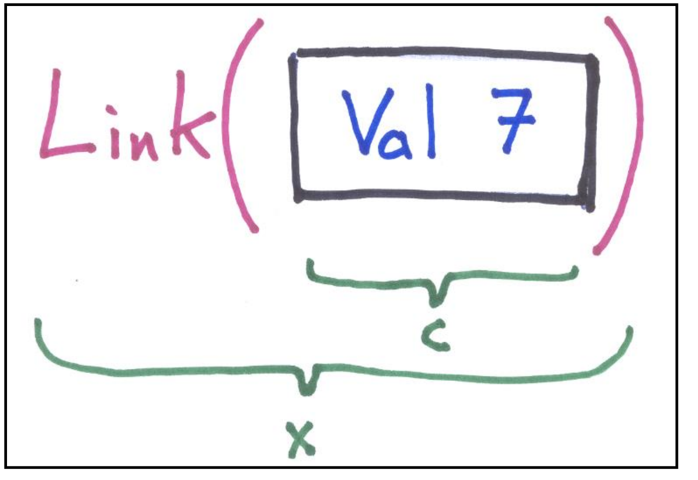
Caution: **`x`** and **`y`** are *different cells* but *contain the same cell* **`d`**, i.e., $\boxed{5}$.

Potentially Circular Structures

```
datatype 'a chain = Val of 'a
                  | Link of 'a chain ref

val (x as Link c) = Link (ref (Val 7))
```

```
x : int chain
c : int chain ref
```

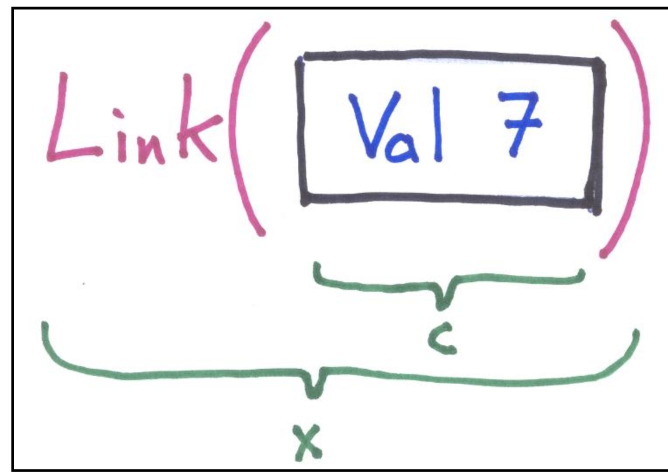


Potentially Circular Structures

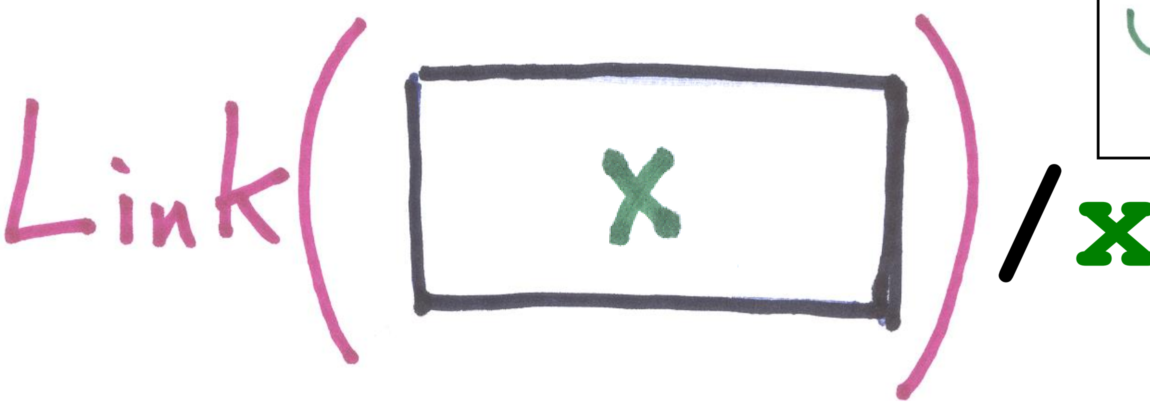
```
datatype 'a chain = Val of 'a
                  | Link of 'a chain ref
```

```
val (x as Link c) = Link (ref (Val 7))
```

```
x : int chain
c : int chain ref
```




```
val () = c := x
```

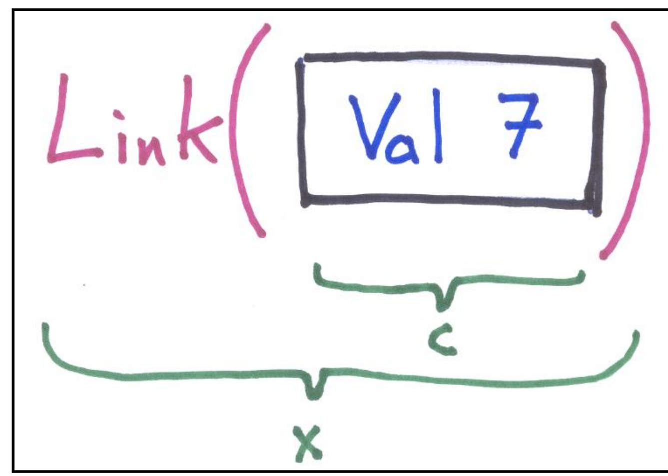


Potentially Circular Structures

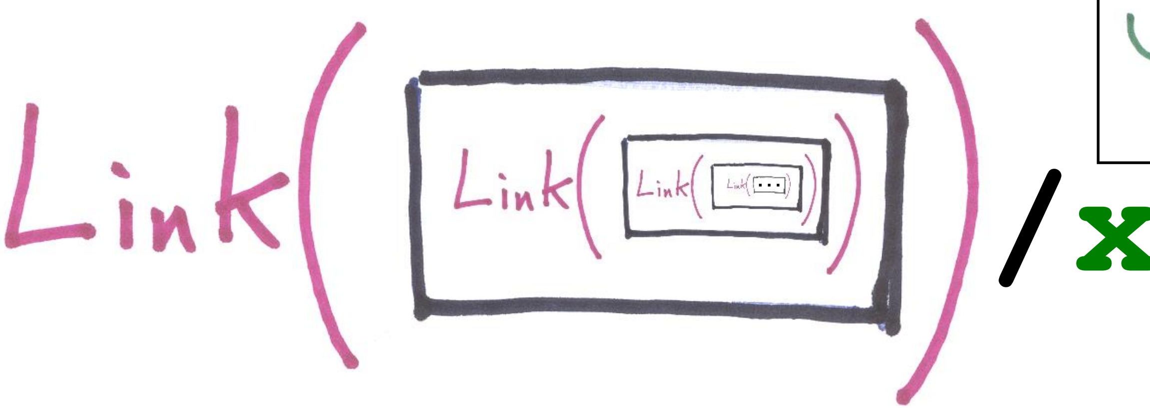
```
datatype 'a chain = Val of 'a
                  | Link of 'a chain ref

val (x as Link c) = Link (ref (Val 7))
```

```
x : int chain
c : int chain ref
```




```
val () = c := x
```



Equality

As usual, avoid equality comparisons in code, except for some base types like **`int`**.

Be aware:

There is a difference between
comparing cells and
comparing cell contents.

Equality

Variables of type `t ref` are equal iff they are bound to the same cell. Consider:

```
val c = ref 10
val d = c
val e = ref 10
```

All three variables are bound to cells containing the integer **10**, so the expressions **!c=!d** and **!c=!e** both evaluate to **true**.

Variables **c** and **d** are aliases, so **c=d** $\hookrightarrow$ **true**.
Variable **e** is bound to a different cell, so **c=e** $\hookrightarrow$ **false**.

Sequential Expressions

SML allows this form of an expression:

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

observe the semi-colons (and the parentheses)

Sequential Expressions

SML allows this form of an expression:

$$(\mathbf{e}_1;\ \mathbf{e}_2;\ \ldots;\ \mathbf{e}_n)$$

The overall expression is well-typed
iff each expression $\mathbf{e}_i$ is well-typed.
In that case, the overall type is the type of $\mathbf{e}_n$:

$$(\mathbf{e}_1;\ \mathbf{e}_2;\ \ldots;\ \mathbf{e}_n)\ :\ \mathbf{t}_n$$

if there exist types $\mathbf{t}_i$ such that
$\mathbf{e}_i : \mathbf{t}_i$, $\mathbf{i}=1,\ldots,\mathbf{n}$.

Sequential Expressions

SML allows this form of an expression:

$$(\mathtt{e_1;\ e_2;\ \ldots;\ e_n})$$

The overall expression has a value
iff each expression $\mathtt{e_i}$ has a value.
In that case, the overall expression has the value of $\mathtt{e_n}$:

$$(\mathtt{e_1;\ e_2;\ \ldots;\ e_n}) \hookrightarrow \mathtt{v_n}$$

if there exist values $\mathtt{v_i}$ such that
$\mathtt{e_i} \hookrightarrow \mathtt{v_i}$, $\mathtt{i=1,\ldots,n}$.

Sequential Expressions

SML allows this form of an expression:

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

Evaluation is left-to-right.

If any e_i raises an exception or loops forever, then the overall expression raises an exception or loops forever, as determined by the leftmost e_i that fails to reduce to a value.

Sequential Expressions

Example:

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

This code creates a reference cell `c`,
prints the contents `10`,
then returns the cell.

What is the type of this `let`?

What is the value?

Sequential Expressions

Example:

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

This code creates a reference cell `c`, prints the contents `10`, then returns the cell.

What is the type of this `let`?

```
int ref
```

What is the value?

```
ref 10
```

Alternate implementation

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

Extensional Equivalence

- Reasoning about equivalence must take into account changes in reference cells.
- We define the *store* to be the set of accessible reference cells along with their contents.
- When evaluating code, we now should write

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

  with `e` and `e'` expressions and `s` and `s'` stores.
- To say `e` ≅ `e'` independent of store means that `{e;s}==>{v;s'}` and `{e';s}==>{w;s'}`, with `v` and `w` equivalent values (or both reductions raise equivalent exceptions with identical stores or both reductions loop forever with identical changes in store), for all initial stores `s`.

Race Conditions

Consider:

```
fun deposit a n = a := !a + n
```

`deposit` increments the contents of cell **`a`** by **`n`**.

`deposit : int ref -> int -> unit`

When we see a return type of **`unit`** in a function, we understand that the function is being called for effect.

Race Conditions

Consider:

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n

val chk = ref 100  (* bank account *)
```

Race Conditions

Consider:

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n
val chk = ref 100   (* bank account *)

val _ = (deposit chk 50; withdraw chk 80)
```

Assume sequential evaluation.

What is the value of **!chk** ?

Race Conditions

Consider:

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n

val chk = ref 100   (* bank account *)

val _ = (deposit chk 50; withdraw chk 80)
```

Assume sequential evaluation.

What is the value of `!chk` ?   **70**

Race Conditions

Now assume parallel evaluation of the pair.

```
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 now is the value of **`!chk`** ?

Race Conditions

Now assume parallel evaluation of the pair.

```
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 now is the value of **`!chk`** ?

There is no definitive answer.

If **`deposit`** and **`withdraw`** happen atomically, then **`70`** as before.
Otherwise, timing of read and write could mean **`20`**, **`70`**, or **`150`**.
If simultaneous writes to the underlying bits, then maybe garbage.

Deterministic Parallelism

The previous example has multiple outcomes, determined *nondeterministically* (that means: beyond our knowledge or control).

We want deterministic outcomes.

Concerns: Sequential vs Parallel Evaluation

Persistent vs Ephemeral Data

no mutation mutable

	Persistent	Ephemeral
Sequential	**Functional programming is a good tool**	**Reasoning is more complicated, but FP is fine.**
Parallel		**need to think about concurrency**

Can include diverging code by left-to-right evaluation semantics.

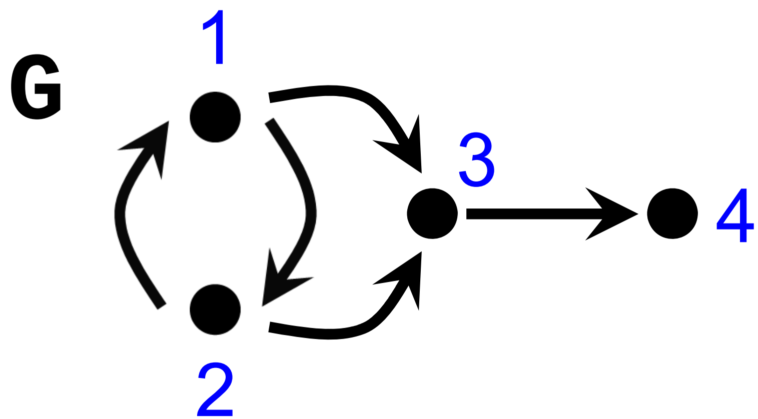
Can also include some mutation as *benign effects* (see subsequent slides).

Benign Effects

A *benign effect* is some effect (such as mutation) that is localized within some sufficiently small chunk of code (such as a 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 while still keeping code simple enough to analyze and prove correct.

Example: Graph Reachability



Can get to vertex 4 from any other vertex, but cannot get to any other vertex from 4.

Let us model a graph as a function that encodes neighbors reachable by a single edge:

```
type graph = int -> int list
```

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

First (naïve) attempt to check reachability:

```
(* reach : graph -> int*int -> bool *)
```

`reach g (x,y)` is supposed to return **`true`** if **`y`** is reachable from **`x`** in **`g`**, and return **`false`** otherwise.

REQUIRE: **`g`** is total.

First (naïve) attempt to check reachability:

```
(* reach : graph -> int*int -> bool *)

fun reach (g : graph) (x,y) =
    let
        fun dfs n = (n=y) orelse

    in
        dfs x
    end
```

Perform a depth-first search. Current vertex is **n**. First, check whether **n** is the desired destination **y**.

Start the search from **x** initially.

First (naïve) attempt to check reachability:

```
(* reach : graph -> int*int -> bool *)

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

Check whether **y** is reachable from any of **n**'s neighbors.

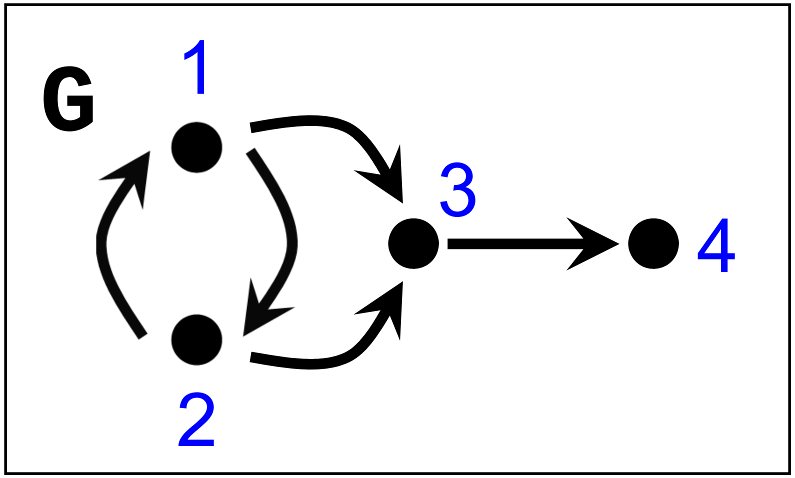
Recall

List.exists : ('a -> bool) -> 'a list -> bool

checks whether some element in the list satisfies the predicate.

First (naïve) attempt to check reachability:

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



Issue: The depth-first search can loop forever on **G**.

We can fix this by updating a visited list:

```
(* mem: int -> int list -> bool *)
fun mem (n:int) = List.exists (fn x => n=x)
```

mem n L checks whether **n** is in list **L**.

We can fix this by updating a visited list:

```
(* mem: int -> int list -> bool *)
fun mem (n:int) = List.exists (fn x => n=x)

(* reachable : graph -> int*int -> bool *)
fun reachable (g:graph) (x,y) =
    let
       val visited = ref []
```

Create a reference cell that will hold a list of vertices (integers) visited during depth first search of the graph.
Initially the list is empty.

```
    in

    end
```

We can fix this by updating a visited list:

```
(* mem: int -> int list -> bool *)
fun mem (n:int) = List.exists (fn x => n=x)

(* reachable : graph -> int*int -> bool *)
fun reachable (g:graph) (x,y) =
    let
       val visited = ref []
       fun dfs n = (n=y) orelse
```

As before, the first thing **`dfs`** does is to check whether it has arrived at the destination **`y`**.

```
    in
       dfs x
    end
```

We can fix this by updating a visited list:

```
(* mem: int -> int list -> bool *)
fun mem (n:int) = List.exists (fn x => n=x)

(* reachable : graph -> int*int -> bool *)
fun reachable (g:graph) (x,y) =
    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 the depth first search if the current vertex **n** has *not* already been visited.
In that case, also update the visited list with **n**.

Alternative approaches

- Pass and return **`visited`** explicitly as an argument.
- Use continuations with **`visited`** as an argument.

Other Roles for Mutation

- Maintain local state in a random number generator.
- Remember stream values that have been exposed previously, so that re-exposing them does not require repeating potentially expensive computations.
  (This is called *memoization*.)

A Random Number Generator

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

bound

pseudo
random nonnegative
integer less than bound

Reference: Paulson, *ML for the Working Programmer*, 1996, p. 108, who points to Park & Miller, *CACM*, 1988, **31**, pp.1192-1201.

A Random Number Generator

```
structure R :> RANDOM =
struct
    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

val G = R.init(12345)
val L = List.tabulate(100, fn _ => R.random G 1000)
```

Reference: Paulson, *ML for the Working Programmer*, 1996, p. 108, who points to Park & Miller, *CACM*, 1988, **31**, pp.1192-1201.

`L` is a list of `100` pseudo random integers in the range `[0,999]`.

Stream Memoization

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

```
fun delay d = Stream d
fun expose (Stream d) = d ()
```

Data persistence means that any and every time someone exposes a given stream, the computation **`d()`** will occur.

Let us add a hidden reference cell that remembers the result of computing **`d()`**. We will leave **`expose`** as is, and change **`delay`**.

Stream Memoization

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

Our first observation is that we can put **d** in a reference cell.

Recall the code for **expose**:

```
fun expose (Stream d) = d()
```

That means we now need a suspension, which when forced will access the reference cell and force the function we put there:

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

Stream Memoization

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

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

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

Stream Memoization

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

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

```
        val _ = cell := memoFn
    in
        Stream (fn () => !cell())
    end
```

Stream Memoization

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

One can even memoize raised exceptions this way.

Stream Memoization

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

That is all.

Have a good weekend.

See you Tuesday, when we will talk about context free grammars.