Lecture 3

Locals Bindings, Options, Benefits of No Mutation, Data types

Local Bindings aka *Let Expressions*

Expressions can have their own local (private) variable and function bindings

- For style and convenience: lets us name intermediate results
- For efficiency: avoid unnecessary recomputation
  - *Not* “just a little faster”
- For safety and maintenance: hide implementation details from clients

Let Expressions : Local Variable Binding

Syntax: `let val x = e1 in e2 end`

- Where `e1` and `e2` are expressions

Let Expressions : Local Variable Binding

Syntax: `let val x = e1 in e2 end`

Type Checking

- If `e1` has type `t1` in current static environment AND
- If `e2` has type `t2` in static environment extended with `x ↦ t1`
- Then `let val x = e1 in e2 end` has type `t2`
- Else, report error and fail

$$\frac{\text{env} \vdash \text{e1:t1} \quad \text{env, x} \mapsto \text{t1} \vdash \text{e2:t2}}{\text{env} \vdash \text{let val x=e1 in e2 end: t2}}$$

Let Expressions : Local Variable Binding

Syntax: `let val x = e1 in e2 end`

Evaluation

- If `e1` evaluates to `v1` in current dynamic environment AND
- If `e2` evaluates to `v2` in dynamic environment extended with `x ↦ v1`
- Then `let val x = e1 in e2 end` evaluates to value `v2`

```
env ⊢ e1 ↦ v1  env,x ↦ v1 ⊢ e2 ↦ v2
------------------------------------
env ⊢ let val x=e1 in e2 end ↦ v2
```

Local Bindings: What's New Here?

- Scope
  - More control over which parts of a program can access a binding
  - For **`let`** expressions: *just the body of the let* !

Let Expressions : Local Function Binding

Syntax: `let fun f ((x1:t1), …, (xN:tN)) = e in e2 end`

Type Checking: Same as top-level function binding, using static environment where the let expression occurs

Evaluation rules: Same as top-level function binding, using dynamic environment where the let expression occurs

A local binding: `f` used only in `e` and `e2`

Let Expression Example

```
nats_upto 5;
- : int list = [0, 1, 2, 3, 4]
```

Not great style:

```
fun nats_upto (x : int) =
  let fun range ((lo : int),(hi : int)) =
    if lo < hi then
      lo :: (range (lo + 1,hi))
    else
      []
  in
  range (0,x)
 end
```

range is hidden…

No one else can use it!

Also, does **range**

really need the **hi** parameter?

Let Expression Example

Better style:

```
fun nats_upto_new (x : int)=
  let fun loop (lo : int) =
    if lo < x then
      lo :: loop (lo + 1)
    else
      []
  in
  loop 0
 end
```

```
nats_upto_new 5;
- : int list = [0, 1, 2, 3, 4]
```

- Functions can use bindings from the environment where they are defined. Including:
  - “outer” environments
  - parameters to outer function
  - etc.
- Unnecessary parameters are *bad style*

DANGER: Avoid Repeated Recursion ⚠️

- How much work does this function do?

```
fun bad_max (xs : int list) =
 if xs = [] then
     0 (* bad style, will fix later *)
 else if tl xs = [] then
    hd xs
 else if hd xs > bad_max (tl xs) then
   hd xs
 else
   bad_max (tl xs)
```

DANGER: Avoid Repeated Recursion ⚠️

- How much work does this function do?

```
bad_max (List.rev (nats_upto 50));;
- : int = 49 (* returns in microseconds *)

bad_max (nats_upto 50);;
- : int = ... (* still running *)
```

Fast vs. Unusable

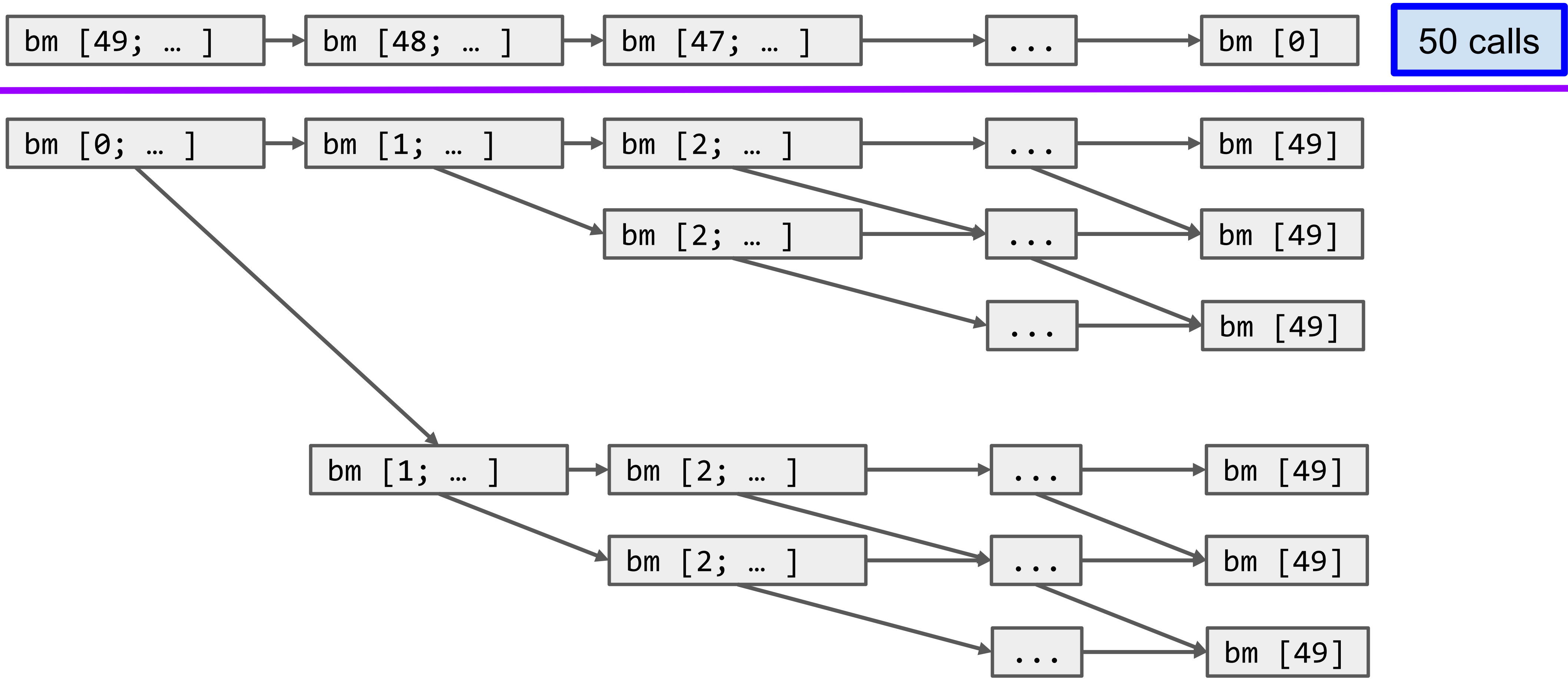
```
if hd xs > bad_max (tl xs)
then hd xs
else bad_max (tl xs)
```

bm [49; …] → bm [48; …] → bm [47; …] → ... → bm [0]

50 calls

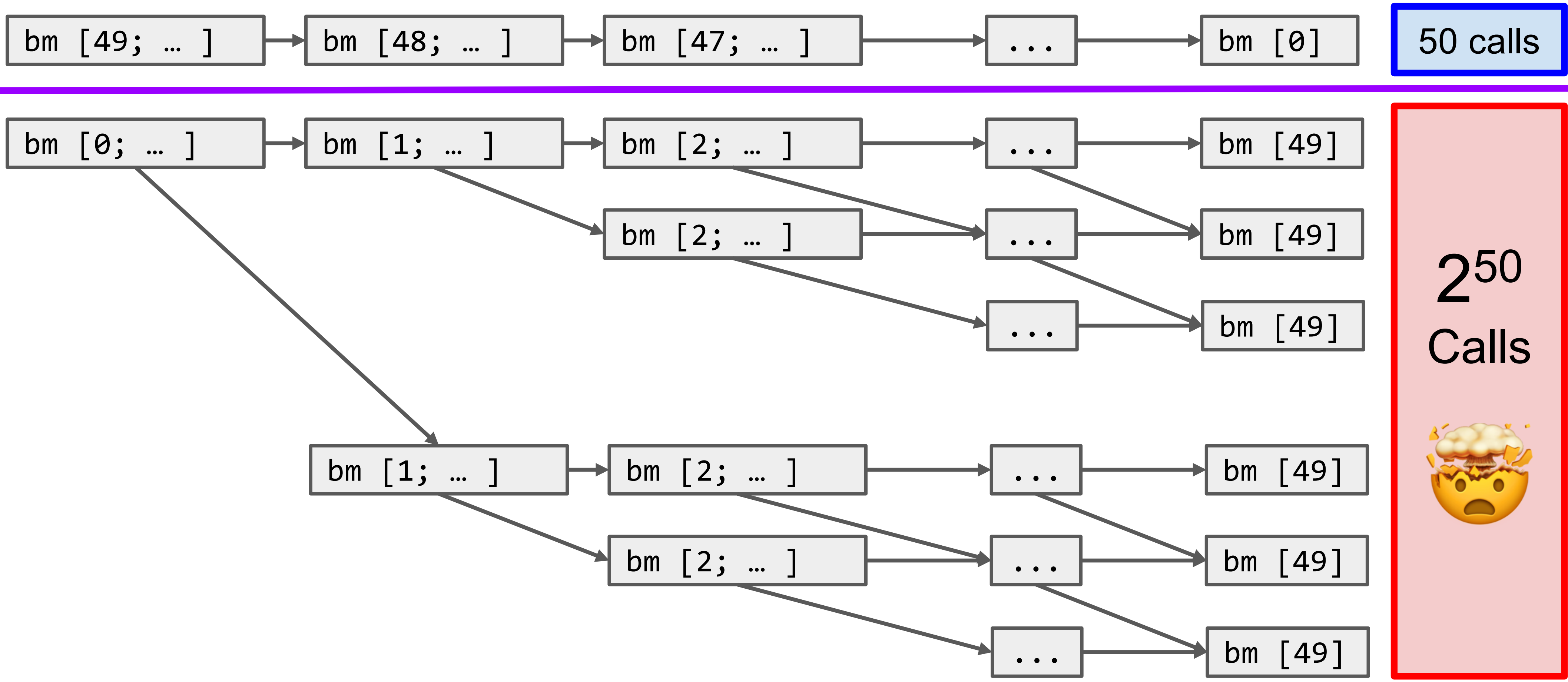
Fast vs. Unusable

```
if hd xs > bad_max (tl xs)
then hd xs
else bad_max (tl xs)
```



Fast vs. Unusable

```
if hd xs > bad_max (tl xs)
then hd xs
else bad_max (tl xs)
```



Math and explosion

- Suppose that one `bad_max` call *not counting recursion* takes 1 microsecond
  - It doesn't matter if this guess is off by 1000x
- Then `bad_max [49; … ; 0]` takes 50 microseconds
- And `bad_max [0; … ; 49]` takes 2^{50} microseconds
  - That's 35.7 years
  - And `bad_max [0; … ; 51]` takes 4x longer

Using `let` to Avoid Repeated Recursion

```
fun better_max (xs : int list) =
if xs = [] then
   0 (* bad style *)
 else if tl xs = [] then
   hd xs
 else
   let val tl_max = better_max (tl xs) in
   if hd xs > tl_max then
     hd xs
   else
     tl_max
   end
```

- Recurse over the list tail *once*
- Store *result* in environment, bound to **`tl_max`**

What to do about partial functions?

- Many computations are not defined or “misbehave” for some inputs
  - `hd []`, `tl []`, maximum element of `[]`, 1 / 0, etc.
- SML does provide *exceptions* for such cases…
  - We will see and use exceptions more later
  - But exceptions let you forget-about-the-exceptional-thing
  - And that can lead to surprising and undesirable control flow
- SML also provides an `option` type for handling partial functions
  - Also for when your data structure “may or may not have some data”

Options

`option` is another type constructor

- Basically: "lists" of length zero or one, but a *different* type constructor
- A value of type `t option` can either be:
  - `NONE`
    - Valid value of type `t option` for *any* type `t`
      - Like `[]` for `t list`
  - `SOME v`
    - Where value `v` has type `t`
      - Like `[v]` for `t list`

Just like our other friends

`t1 * t2`

`t1 -> t2 t list`

Options : Build -- None

Syntax: `NONE`

Type Checking

- `None` has type `'a option`

None has type

`t option` for any `t`

Evaluation

- `None` is a value

Options : Build -- Some

Syntax: `SOME e`

- Where `e` is an expression

Type Checking

- If `e` has type `t`
- Then `SOME e` has type `t option`
- Else, report error and fail

Evaluation

- If `e` evaluates to value `v`
- Then `SOME e` evaluates to value `SOME v`

A Better Better Better Max

```
Fun good_max (xs : int list) =
  if xs = [] then
    NONE
  else
    let val tl_ans = good_max1 (tl xs) in
    if isSome tl_ans andalso valOf tl_ans > hd xs then
      tl_ans
    else
      SOME (hd xs)
```

returns an `int option`, *NOT* an `int`, so can finally give a reasonable answer for even “bad” inputs (i.e., `[]`)

clients are forced to consider data and no-data cases

Aliasing and Mutation

Can you spot the difference? Can a client?

```
fun sort_pair (pr : int*int)=
   if #1 pr < #2 pr then
     pr
   else
     (#2 pr, #1 pr)
```

```
fun sort_pair (pr : int*int)=
   if #1 pr < #2 pr then
     (#1 pr, #2 pr)
   else
     (#2 pr, #1 pr)
```

Can you spot the difference? Can a client?

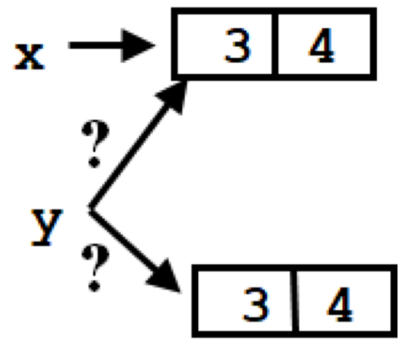
```
fun sort_pair (pr : int*int)=
   if #1 pr < #2 pr then
     pr
   else
     (#2 pr, #1 pr)
```

```
fun sort_pair (pr : int*int)=
   if #1 pr < #2 pr then
     (#1 pr, #2 pr)
   else
     (#2 pr, #1 pr)
```

- If pair already sorted, return (an alias to) the same pair, or a new pair with the same contents?
- When data is immutable, aliasing does not matter

Why lack-of-mutation matters

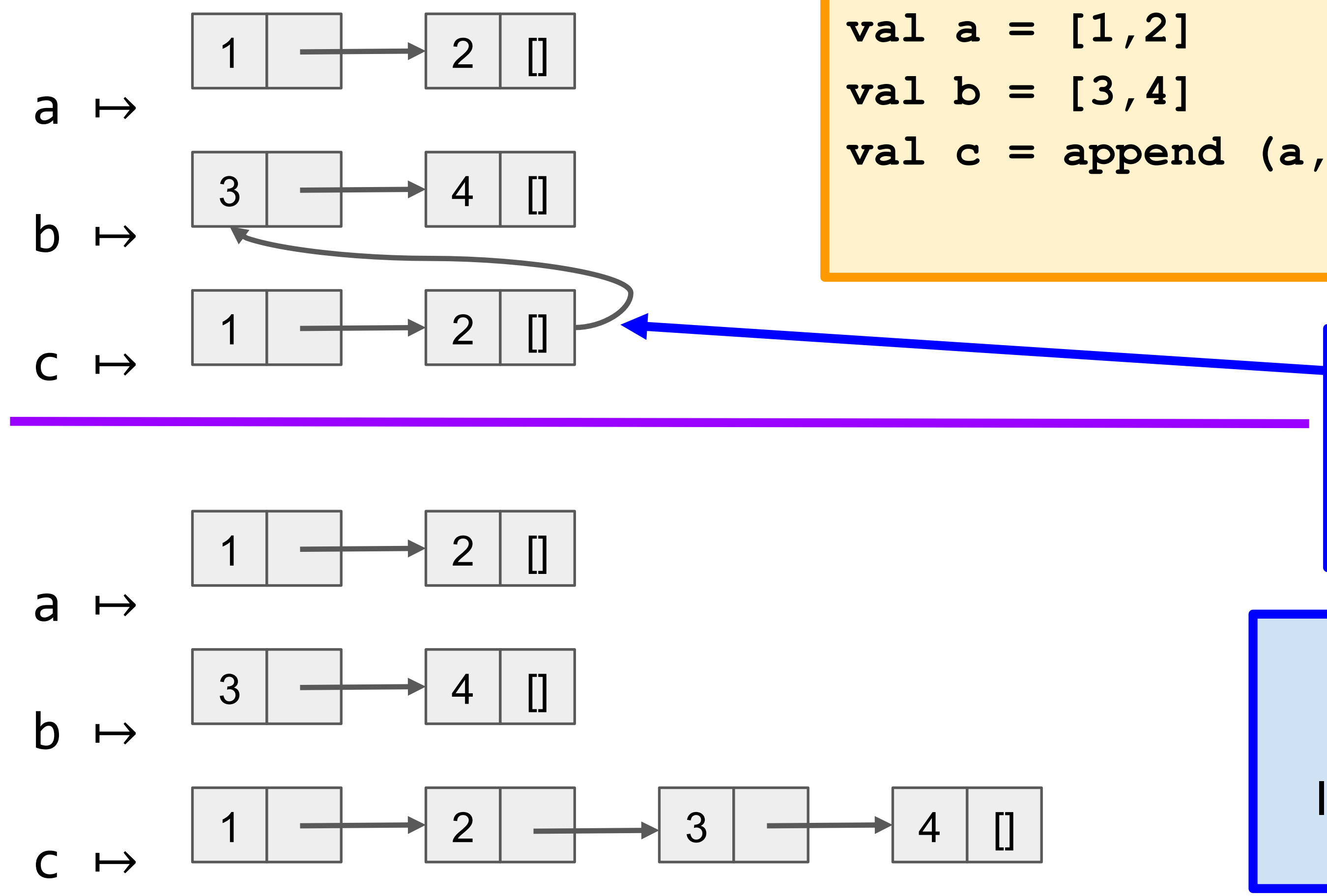
```
val x = (3,4)
val y = sort_pair x
somehow mutate #1 x to hold 5
val z = #1 y
```



- What is **`z`**?
  - Would depend on how we implemented **`sort_pair`**
- Would have to decide carefully and document **`sort_pair`**
  - Without mutation, we can implement “either way”
    - No code can ever distinguish aliasing vs. identical copies
    - No need to think about aliasing: focus on other things
    - Can use aliasing, which saves space, without danger

Aliasing with append

```
fun  append ((xs: ''a list),(ys : ''a list))=
if xs = []then ys
 else hd xs :: append (tl xs, ys)
val a = [1,2]
val b = [3,4]
val c = append (a,b);
```



Clients can never tell!

(but it's actually the upper one)

Can *safely* reuse and share data.

Immutability can be *more efficient*!

ML vs. Imperative Languages

- In ML, we create aliases all the time without thinking about it because it is *impossible* to tell where there is aliasing
  - Example: `tl` is constant time; doesn't copy anything
  - No need to worry!
- In languages with mutable data (e.g., Java), programmers are *obsessed* with aliasing and object identity
  - They have to be (!) so that subsequent field assignments affect the right parts of the program
  - Often crucial to make copies in just the right places
    - Consider a Java example

Java security nightmare (bad code)

```
class ProtectedResource {
   private Resource theResource = ...;
   private String[] allowedUsers = ...;
   public String[] getAllowedUsers() {
     return allowedUsers;
   }
   public String currentUser() { ... }
   public void useTheResource() {
     for(int i=0; i < allowedUsers.length; i++) {
            if(currentUser().equals(allowedUsers[i])) {
            ... // access allowed: use it
            return;
            }
     }
     throw new IllegalAccessException();
   }
}
```

Have to make copies

The problem:*

```
p.getAllowedUsers()[0] = p.currentUser();
p.useTheResource();
```

The fix:

```
public String[] getAllowedUsers() {
    … return a copy of allowedUsers …
 }
```

Copying immutable data is unnecessary!

Building Types

- We have studied several types already
  - Base types : `bool` `int` `float` `string` `unit`
  - Compound types : `t1 * t2` `t list` `t option`
- **H**ow to design and define our own types
  - Type definitions in SML provide mechanisms for building values of those type : constructors
  - Will need a general mechanism for use (accessing their pieces) : pattern matching

Data Types

Syntax: datatype `t = C1 of t1 | … | CN of tN`

- Where `t1, …, tN` are types (*self reference*: `t` can appear in these type arguments!)
- Where `C1, …, CN` are *constructors* (name must start with capital letter)

Adds new type `t` and constructors `C1, …, CN`

- Types are in their own name space
- Constructors are in their own name space

Data Types

Syntax: `datatype t = C1 of t1 | … | CN of tN`

- Where `t1, …, tN` are types (nts!)
- Where `C1, …, CN` are *constru*

Constructor parameters are optional.

Constructors that don't take any arguments are just values of their type.

Construction

Syntax: `C e`

- Where `C` is a constructor and `e` is an expression

Type Checking

- If `t` is defined as `datatype t = C1 of t1 | … | C of tC | … | CN of tN`
- If has `e` has type `tC`
- Then `C e` has type `t`
- Else, report error and fail

Evaluation

- If `e` evaluates to value `v`
- Then `C e` evaluates to `C v`

Construction

Syntax: `C e`

- Where `C` is a constructor and `e` is an expression

Type Checking

- If `t` is defined as `type t = C1 of t1 | … | C of tC | … | CN of tN`
- If has `e` has type `tC`
- Then `C e` has type `t`
- Else, report error and fail

Values “tagged with a constructor” are values!

Evaluation

- If `e` evaluates to value `v`
- Then `C e` evaluates to `C v`

Construction

Syntax: `C e`

- Where `C` is a cons

Type Checking

- If `t` is defined as `type t = C1 of t1 | … | C of tC | … | CN of tN`
- If has `e` has type `tC`
- Then `C e` has type `t`
- Else, report error and fail

Evaluation

- If `e` evaluates to value `v`
- Then `C e` evaluates to `C v`

- Constructors are **NOT** functions!
  - They don't have a function body or other code
  - They don't evaluate, just "tag some data"
- Instead, each constructor of a datatype **`t`** is more like a "tag" that can make some kind of data also belong to **`t`**.

Values "tagged with a constructor" are values!