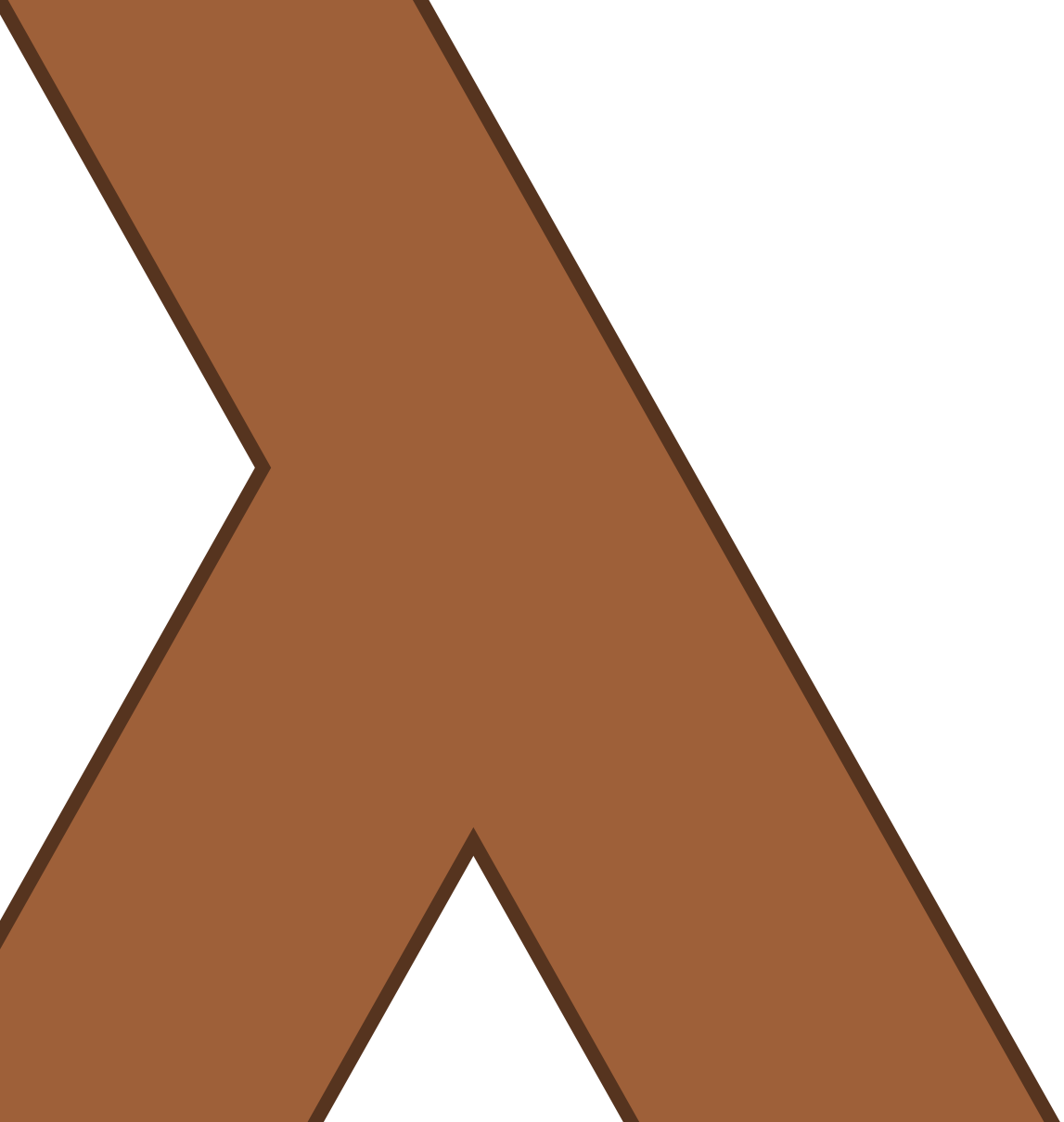


Functional Code

15-150 M21

Lecture 0524
24 May 2021

Friday's slogan:

Computation is evaluation

Today's slogan:

Computation is typed evaluation

0 Compile Time

Person 1: Hey, do you know what 2^{17} is?
Person 2: Yeah, it's banana.
Here, Person 2 is just saying nonsense. Clearly, 'banana' is not the correct answer to what 2^{17} is.

Moral: Queries like "what is 2^{17}? also come with implicit constraints on what *kind of thing* the answer is allowed to be. Note that 'banana' might be a value, it's just not the proper *type* of value.

`smlnj` REPL

Key Point

SML is **strongly-typed**: every expression we want to evaluate *must* have a **type**.

Demonstration: Typechecking

- `int` is a type
- Each integer literal is a **value** of type `int`. `0:int`, `17:int`, `~23:int`, etc.
- If `e:int`, then `~e:int`
- If `e1:int` and `e2:int`, then `(e1+e2):int`
- If `e1:int` and `e2:int`, then `(e1 * e2):int`
- If `e1:int` and `e2:int`, then `(e1 - e2):int`
- If `e1:int` and `e2:int`, then `(e1 div e2):int`
- If `e1:int` and `e2:int`, then `(e1 mod e2):int`

- `string` is a type
- Each string literal is a **value** of type `string`. So `"hello":string`, `"":string`, `"mwef8892 cjqq" :string`, etc.
- If `e1:string` and `e2:string`, then `(e1 ^ e2):string`
- If `e:int`, then `Int.toString(e):string`

- `bool` is a type
- There are exactly two values of type `bool`, namely `true : bool` and `false : bool`.
- If `e : bool`, then `(not e) : bool`.
- If `e1 : bool` and `e2 : bool`, then `(e1 orelse e2) : bool` and `(e1 andalso e2) : bool`.
- If `t` is any type and `e1 : t` and `e2 : t` and `b : bool`, then

```
(if b then e1 else e2) : t
```

- If `e1` and `e2` are expressions of type `int` (or `string`, `bool`, some other types), then

```
(e1 = e2) : bool
```

Check Your Understanding

- Is this expression well-typed? If so, what's its type? What happens when you evaluate it?

```
(if true then 5+5 else 7) = 1
```

- Is this expression well-typed? If so, what's its type? What happens when you evaluate it?

```
if (3+3)=6 then "red" else 42
```

- Is this expression well-typed? If so, what's its type? What happens when you evaluate it?

```
1 div 0
```

More often than not, inconsistent typing is a bug

A good tool enables you to do what you want; a *smart* tool prevents you from doing what you *shouldn't*

- Recursive: The type of an expression is determined by the types of its sub-expressions
- Static: We do not evaluate the code when typechecking

Disallows:

```
def foo(x):
    if (x==2): return "a"
    else: return 3
```

The type of `foo(v)` depends on the *value* of `v`

Key Point: Compile time vs. runtime

The execution of SML code happens in two steps:

- Compile Time: Syntax- and Type-checking
- Runtime: Evaluation

1 Runtime

Product Types

If `T1` and `T2` are types, then `T1*T2` is a type – the *product type* of `T1` and `T2`

Pair Rule If `e1 : T1` and `e2 : T2`, then `(e1 , e2) : T1 * T2`

To evaluate `(e1 , e2)`:

1. Evaluate `e1` down to some value `v1` (if possible)
2. Evaluate `e2` down to some value `v2` (if possible)
3. The value of `(e1 , e2)` is `(v1 , v2)`

- `int` is a type
- Each integer literal is a **value** of type `int`. `0:int`, `17:int`, `~23:int`, etc.
- If `e:int`, then `~e:int`
- If `e1:int` and `e2:int`, then `(e1+e2):int`
- If `e1:int` and `e2:int`, then `(e1 * e2):int`
- If `e1:int` and `e2:int`, then `(e1 - e2):int`
- If `e1:int` and `e2:int`, then `(e1 div e2):int`
- If `e1:int` and `e2:int`, then `(e1 mod e2):int`
- If `P : int*int`, then `(Int.max P):int`
- If `P : int*int`, then `(Int.min P):int`

Demonstration: Evaluation

`smlnj` REPL: `val` declarations

Often it's helpful to give names to particular values, to be able to refer to them later.

```
val x : t = e
```

What SML does with this syntax:

1. (Compile Time) Checks that `e` is a well-typed expression of type `t`
2. (Runtime) Evaluates `e`
3. (Runtime) *If evaluating `e` results in a value* (call it `v`), SML **binds** the value `v` to the variable name `x`.

We denote such a binding with the notation `[v/x]`. Note this is not valid SML syntax, but mathematical notation *about* SML.

Each valid `val` declaration adds a **binding** to the **environment**

To evaluate an expression containing variable names, we substitute in for each variable the **most recent binding** to that variable in the environment

Moral: Don’t shadow!

```
val x : int = let
                  val y = 2
                  val z = y * y
              in
                  3 + z
              end
```

`[7/x]` is added to the environment, but `[2/y]` and `[4/z]` are not.

Check Your Understanding

What value gets bound to z as a result of this code?

```
val z = let
          val y = let
                    val z = 2
                  in
                    z * z
                  end
          val z = y + y
          val y = 5
        in
          y - z
        end
```

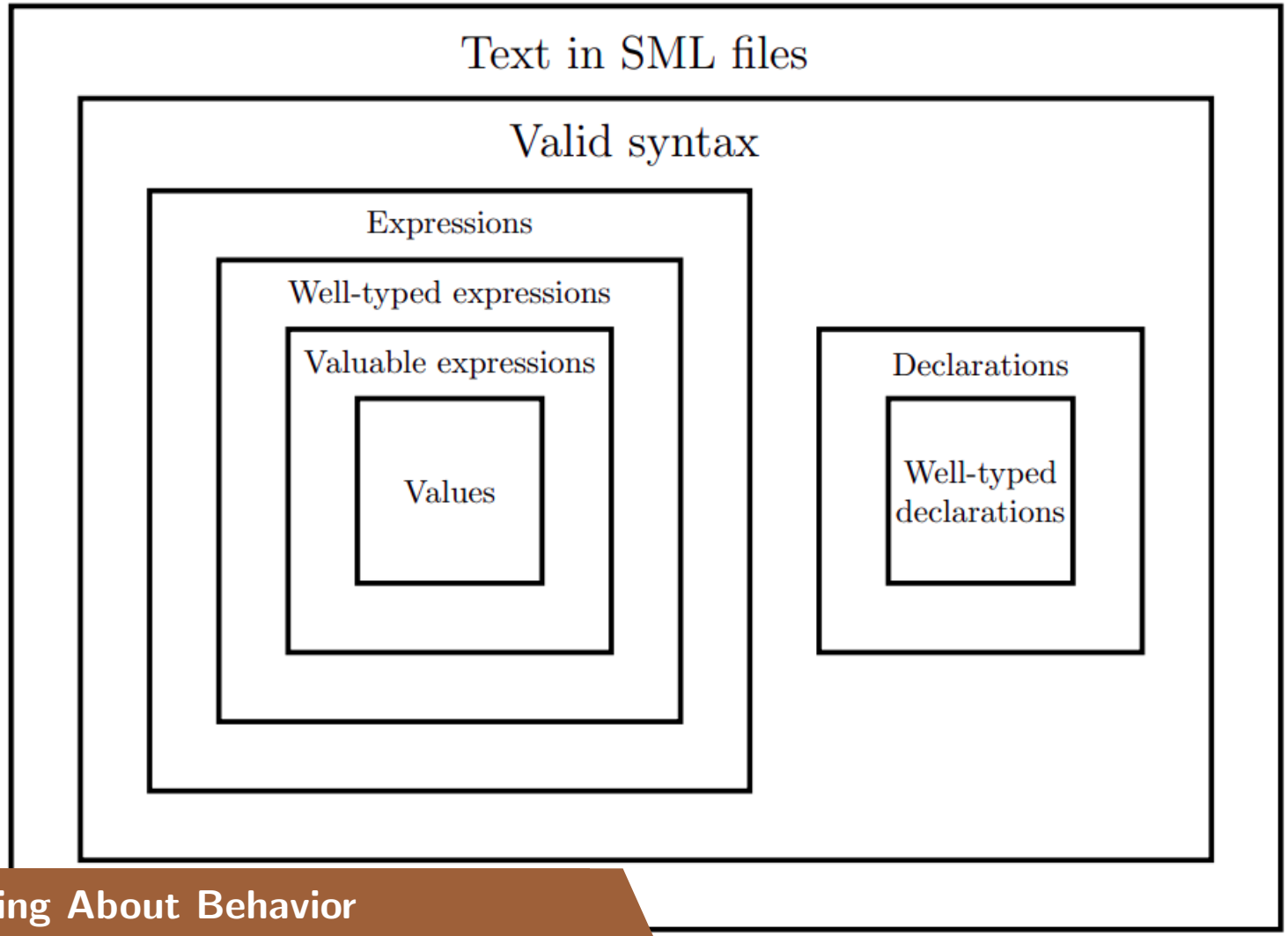
5-minute break

2 Reasoning About Behavior

Claim For every well-typed expression e, exactly one of the following holds:

- $e \Longrightarrow v$ for some value v
- the evaluation of e raises some exception
- the evaluation of e loops forever

The Syntactic Hierarchy



Two expressions e and e' are *equivalent* if they have the same runtime behavior.

Defn. Two well-typed expressions e and e' are said to be **extensionally equivalent** (written $e \cong e'$)if they have the same type and either:

- there is some value v such that $e \hookrightarrow v$ and $e' \hookrightarrow v$
- the evaluation of e and e' both raise the *same* exception
- the evaluation of both e and e' loop forever

- $\cong$ is an equivalence relation
- If $\texttt{e1} \Longrightarrow \texttt{e2}$, then $\texttt{e1} \cong \texttt{e2}$.

  Check Your Understanding: Does $\texttt{e1} \cong \texttt{e2}$ imply $\texttt{e1} \Longrightarrow \texttt{e2}$?

  No (e.g. $5 \cong 4+1$ but $5 \not\Longrightarrow 4+1$)

Extensionally-equivalent expressions are interchangeable: if e $\cong$ e', then any instance of e in a piece of SML code can be replaced with e', without changing the behavior of the overall code.

This is useful because it allows us to swap out parts of code with better implementations without affecting the surrounding code.

3 Function Application

Recall this expression from the previous lecture:

```
exp 17
```

Of course, `17` is a value of type `int`. But we need to say what type of expression `exp` is, and how that determines the type of `(exp 17)`.

If `T1` and `T2` are types, then `T1 -> T2` is a type: the type of **functions** *from* `T1` *to* `T2`.

Application Rule If `f : T1 -> T2` and `e : T1`, then

`(f e) : T2`

- `~ : int -> int`
- `exp : int -> int`
- `Int.toString : int -> string`

Some functions of type `(T1 * T2) -> T3` are written in *infix* position (between its arguments) instead of *prefix* position. In SML, the `op` keyword turns infix functions into prefix ones.

```
(op +)    : int * int -> int
(op * )   : int * int -> int
(op -)    : int * int -> int
(op div)  : int * int -> int
(op >)    : int * int -> bool
(op =)    : int * int -> bool
(op =)    : string * string -> bool
```

- Expressions must be well-typed in order to be evaluated
- Syntax & Type checking happen at compile time, which is before runtime, when evaluation occurs
- We can reason mathematically about runtime using bindings, $\Longrightarrow$, and $\cong$
- Functions are expressions that can be applied to other (appropriately-typed) expressions

Next Time

- Functions and closures
- Pattern Matching
- Recursion

Thank you!