Lesson 2

EQUIVALENCE, BINDING, AND SCOPE

May 18, 2023

In the last lecture, we learned about **expressions**, **values**, and **types**.

We learned that **only well-typed expressions are evaluated**, and that expressions can exhibit one of three behaviors:

- Evaluate to a value
- Raise an exception
- Loop forever

We also saw some examples of **typing rules** which SML uses to determine whether an expression is well-typed or not.

1 - More Types

There are a few basic types to know in SML.

The **base types** comprise of a few simple primitives:

- `int` - `1, 150, 412`
- `real` - `1.0, 1.50`
- `char` - `#"a", #"1"`
- `bool` - `true, false`
- `string` - `"functions", "are", "values"`

Types get a lot more interesting, though!

SML has **tuples**, which are just collections of values of other types.

So for instance, valid tuples include:

- `(1, 2) : int * int`
- `(1, "hi") : int * string`
- `("a", true, 1.0) : string * bool * real`

Def We call a type like `int * int` or `string * bool * real` a **tuple type**, or **product type**.

Note Tuples evaluate from **left to right**. So `(1 + 1, 2 + 2)` $\Longrightarrow$ `(2, 2 + 2)` $\Longrightarrow$ `(2, 4)`.

Def The typing rule for tuples is that if `e1 : t1`, `e2 : t2`, ..., `en : tn`, then `(e1, e2, ..., en) : t1 * t2 * ... * tn`

Note Parentheses matter! The type `int * string * bool` is **different** than `int * (string * bool)`

So for instance, `(1, 2, 3) : int * int * int` is a valid tuple, but so is `(1, (2, 3)) : int * (int * int)`. These are different values!

The first value is a tuple of three things, but the second is a tuple of two things, where the second part is itself a tuple.

2 - Functions

Similarly to how we can use `*` to make tuple types out of other types, we can use `->` to make function types out of other types.

Remark This is because `*` and `->` are known as *type constructors*.

Def We call a type like `int -> int` a **function type**, which takes in a value of type `int` and evaluates to an expression of type `int`.

For instance, we have `not : bool -> bool`, such that `not true` $\Longrightarrow$ `false`.

Similarly to how we can create expressions with tuple type, we can create expressions with function type.

We write `fn (x : int) => x + 1` for the function which takes in an int, and adds one to it. We call this a **lambda expression**.

Note Lambda expressions are expressions! They do not declare a function that can be referenced, they are **anonymous**, and do not have names .

So `(fn (x : int) => x + 1) 2` $\Longrightarrow 3$.

Lambda expressions are themselves values, meaning that they do not reduce to anything else.

Note Functions are values.

As we will say many times over the course of the semester, **functions are values**.

This means that they can be bound to variables like any other value! So while we earlier wrote

```
fun double (n : int) : int = n + n
```

we could equivalently have written

```
val double = fn (n : int) => n + n
```

We will have more to say on this idea later.

Suppose I wanted to declare the `fact` function using a lambda expression.

I might write something like:

```
fn (n : int) =>
  if n = 0 then 1
  else (* ??? *)
```

But what do I call in the recursive case? How do I call a lambda recursively?

Note You can't. Lambdas are non recursive. They don't have names, so they can't reference themselves.

Note An expression `e1 e2` **first evaluates** `e1`, and then `e2`.

SML is an **eagerly evaluated** language, meaning that arguments to functions are **always** reduced to values, before the function can be called.

Consider the expression `(fn (x : int * int) => 150) (1 + 1, 3 * 4)`

```
(fn (x : int * int) => 150) (1 + 1, 3 * 4)
⟹ (fn (x : int * int) => 150) (2, 3 * 4)        (def of +)
⟹ (fn (x : int * int) => 150) (2, 12)           (def of *)
⟹ 150                                           (function application)
```

Note that, when stepping expressions like this, usually we will want to cite a **justification** for each step to the right of the newly-obtained expression. You should do this on your homework too.

We glossed over this in the previous lecture, but when we apply a function, it's with the goal of stepping into the **function body**.

Def The **body** of a function is the expression that the function should evaluate to, given arguments.

So for instance, in `fn (n : int) => n + n`, the body is the expression `n + n`.

When the arguments to a function are values, we can then **substitute** those values for the function's arguments, within its body. This produces **bindings**.

3 - Binding and Scope

What is a variable declaration really doing?

The syntax:

```
val x : int = 2
```

binds the value `2` to the variable `x`.

Def **Binding** is the act of producing a new association of a value to a variable name.

Note **Binding is not assignment**.

The easy way to see how binding differs from assignment is to consider the following code:

```
val x : int = 2
fun foo (y : int) : int = x + y
val x : int = 4
```

After this code, what is the value of `foo 1`?

The imperative answer is `5`.

The SML answer is `3`.

```
val x : int = 2
fun foo (y : int) : int = x + y
val x : int = 4
```

In an imperative language, you **change the world** by **reassigning** the value of the variable `x`.

This changes the value referenced by `fun foo (y : int) = x + y`.

In a functional language, you bind a **new, unrelated** variable called `x`, whose value is `4`, but is **not** the same as the one referenced in `foo`.

Suppose you are named Brandon, and you have a 9-5 job.

Your manager walks in and says "Brandon, your performance has been suffering lately", and walks out.

Then, another engineer named Brandon walks in and sits down.

You are still in trouble.

The point: Just because something (or someone) named the same walked in, doesn't change who your manager was talking about!

Def The **environment** at a particular point in a program is the collection of all currently active bindings.

`val` bindings and `fun` declarations introduce **new bindings** into the environment, and displace old ones. This is called **shadowing**.

When a binding is shadowed, we are no longer in its scope.

We use the mathematical notation $[5/\texttt{x}, \texttt{true}/\texttt{y}]$, for instance, for the environment where `5` is bound to `x` and `true` is bound to `y`.

Let's look at an example of a trace with environments attached.

Suppose our program is:

```
val x : int = 1
val y : int = 1 + x
val x : int = 3
val z : int = x + y
```

`val x : int = 1`	(results in)	$[1/x]$
`val y : int = 1 + x`	(results in)	$[1/x, 2/y]$
`val x : int = 3`	(results in)	$[3/x, 2/y]$
`val z : int = x + y`	(results in)	$[3/x, 2/y, 5/z]$

It might seem like this is no different than assignment.

Key Environments and bindings are shown as different when functions get involved.

Def A function binding is composed of two things, a lambda expression and the environment **at the time of binding**. This is known as a **closure**.

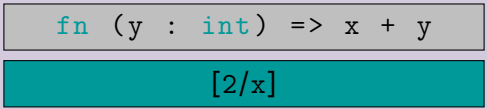
This means that a function always only knows about what was in the environment when it was first bound. It doesn't see any bindings that happen later.

Note Functions are elephants. (they remember everything)

```
val x : int = 2
fun foo (y : int) : int = x + y
val x : int = 4
```

After the first binding of `x`, we have the environment $[2/x]$.

Then, we could represent the closure as:



which is bound to the identifier `foo`. **Future bindings will not change this.**

4 - Pattern Matching

We can write declarations that bind tuples to variables, for instance:

```
val x : int * int = (1, 2)
```

But now, how do we access the different parts of the tuple? What if we want to get `1` and `2` back out of the tuple?

We can unpack it using a **pattern**.

```
val (x, y) : int * int = (1, 2)
```

This produces the environment $[1/\mathtt{x}, 2/\mathtt{y}]$

Def A **pattern** is a way to describe the **form** of a value. A value can either match to a pattern, or not.

The goal is that patterns should be used to *describe* the values that you are interested in.

For the example we gave above, we used the pattern `(x, y)` to **deconstruct** the value `(1, 2)`, because we knew that the tuple `(1, 2)` has two entries.

The right-hand side of the expression doesn't even need to be a value! It turns out, all you need to know to figure out the right pattern to use is the expression's **type**.

Types and patterns have a nice relationship, where they both correspond to each other.

For instance, if we wanted to have a declaration like:

```
val (x, y) = (* something goes here *)
```

then it would be ill-typed to put something like `1`. Why?

Answer: `1` is not a tuple! It doesn't have components to unpack.

The fact that we have the pattern `(x, y)` *implies* that the right-hand side should be of type `t1 * t2`, for some types `t1`, `t2`. In other words, it needs to be a tuple of two components.

Patterns can be used for a lot more than tuples, however. Here's some other examples of patterns:

- variables, such as `x`, `y`, `z`. These produce a binding on a successful match.
- the **wildcard** pattern, `_`. This matches **all** values, and produces no binding.
- constants, such as `1`, `"hi"`, and `true`
- tuples, where each entry is itself a pattern. For instance, `(_, x, 2)` is a valid pattern.

Now, we are ready to talk about the form of a `val` declaration. A `val` declaration looks like:

```
val <pattern> : <type> = <expr>
```

You can put any pattern on the left-hand side of a binding! Some patterns might not type-check, however.

```
val 2 = (1, 2) (* doesn't typecheck! never runs *)
```

If a value matches to a pattern, then it will produce some number of bindings, and then proceed with the program. If the value doesn't match, then an exception will be raised, for instance on the binding

```
val 2 = 1 (* doesn't match! exception raised *)
```

We have seen some simple examples of functions, such as

```
fun double (n : int) : int = n + n
```

But what if we want functions with more complicated control flow? We might introduce **conditionals**.

```
fun isEven (n : int) : bool =
  if n mod 2 = 0 then true
  else false
```

Warning Do not do this.

Hold for a brief interjection.

In SML, we have the **if expression**. Note that it is an *expression*. It looks like:

```
if <expr1> then <expr2> else <expr3>
```

Def The typing rule for if expressions is that for `if e1 then e2 else e3` : t iff `e1 : bool`, and `e2 : t` and `e3 : t`.

Why? Because otherwise we could get type unsafety! Consider the following conditional expression:

```
if sasquatchIsReal then 2 else "foo"
```

Depending on if `sasquatchIsReal`, we might return an `int`, or we might return a `string`! This is really bad, because this breaks our whole future-sight of type-checking.

Back to the warning.

The way that `if e1 then e2 else e3` works is that if `e1` evaluates to `true`, then the whole expression evaluates to `e2`. Otherwise, if `e1` evaluates to `false`, then the whole expression evaluates to `e3`.

So let's write the computation trace of `isEven 2`.

`isEven 2`	$\Longrightarrow$ `if 2 mod 2 = 0 then true else false`	(def of `isEven`)
	$\Longrightarrow$ `if 0 = 0 then true else false`	(def of `mod`)
	$\Longrightarrow$ `if true then true else false`	(def of `=`)
	$\Longrightarrow$ `true`	(if expression)

This is really redundant.

Lesson Never write `if e then true else false`!

Now that we have if expressions, we can write more complicated functions.

```
fun fact (n : int) : int =
  if n = 0 then 1
  else n * fact (n - 1)
```

Function syntax in SML actually offers a better way of writing certain things. For instance, we could write:

```
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact (n - 1)
```

We call these **function clauses**.

We write function clauses in the general form as:

```
fun f (<pat1> : <ty1>) : <ty2> = <expr1>
  | f (<pat2> : <ty1>) : <ty2> = <expr2>
  ...
```

When a function written in this way takes in an argument, that argument has already been evaluated to a value.
That value is then matched against each pattern linearly, until it finds a pattern that matches.

So `fact 1` matches against the pattern `0`, fails, and then matches against `n`. This succeeds, so then we enter the case of `n * fact (n - 1)`, with the environment $[1/\texttt{n}]$.

Function clauses behave similarly to if expressions, in that they must always return the same type. Otherwise, we might get type unsafety.

Writing functions via clauses is generally more powerful than when using if expressions. Pattern matching is a more fundamental notion than conditionals, and we will see in the next lecture what else we can use it for.

5 - Equivalence

We have now described the difference between variable binding in SML and assignment in other languages.

A question remains - **what's the point?**

In the first lecture, we described some of the ideals of a good programming language, and one of them was **modularity**. We should be able to change a part of a program without affecting another.

Remember that functions in SML are like mathematical functions, they always give the same outputs for the same inputs (purity).

Variable binding allows a stronger property, which is that **no binding after a function declaration can change that function's behavior**.

In other words, **binding preserves function equivalence**.

Functional programming lends itself to reasoning mathematically about code.

To supplement that, we will have a notion of when code is equivalent.

Def Two expressions **of the same type** are said to be **extensionally equivalent** if they:

- evaluate to the same value,
- both loop forever,
- or both raise the same kind of exception

We write `e1` $\cong$ `e2` when `e1` and `e2` are extensionally equivalent.

So for instance, `2 + 2` $\cong$ `4` $\cong$ `1 + 3`.

You might think that we already saw something like this.

In the last lecture, we explored the idea of **reduction**, which is when an expression is simplified to another. For instance, we learned that $1 + 1 \Longrightarrow 2$.

Based on the definition given before, we also know that $1 + 1 \cong 2$. What gives? What's the difference?

The reason is that **reduction is stronger than equivalence**.

To say that a condition, or conclusion, is *stronger* than another, is to say that it implies the thing that it is stronger than.

For instance, if some fact A being true implies that B must be true, then A is a stronger condition, because knowing A also gives you information about B. But, knowing B doesn't necessarily tell you anything about A.

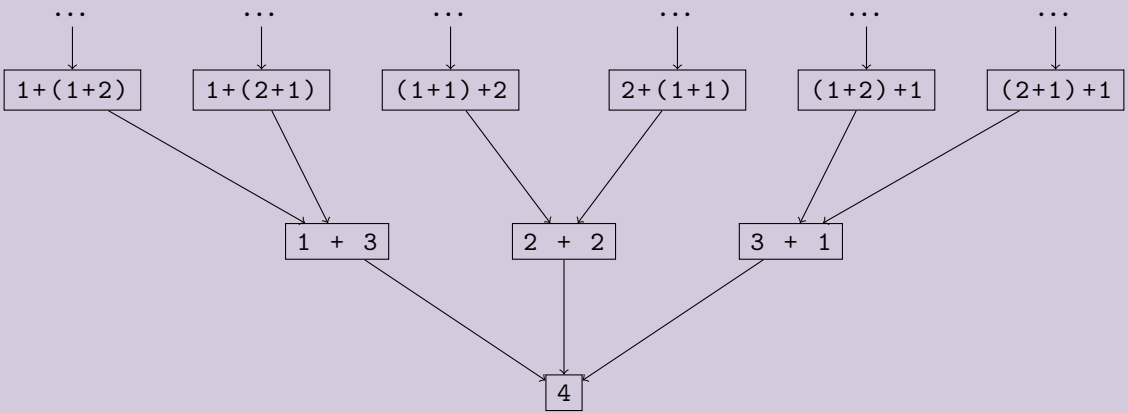
So for instance, "I am a human" would be a stronger condition than "I breathe air", because if I know I am a human, then I also already know I breathe air.

What I'm getting at here is that *reduction implies equivalence*.

Any time that you know that `e1` $\Longrightarrow$ `e2`, you also know that `e1` $\cong$ `e2`. So reduction is a stronger condition.

What this means is that, whenever you are deciding which to use, know that you can't freely go from one to another! If you know equivalence, you don't necessarily know reduction. You need to take care to make sure you use the right notation.[1]

[1] There is at least one homework problem in this class where the distinction is important.



Every node[2] in this graph is extensionally equivalent, but reduction only flows one way!

[2] If you want to sound really fancy, say "lattice".

It turns out that this idea gives us some very nice properties.

Def **Referential Transparency** - If `e1` $\cong$ `e2`, then replacing `e1` with `e2` anywhere in a program will produce an extensionally equivalent program.

So for instance, you could replace `2 + 2` with `1 + 3` anywhere, and always be assured that program behavior **will not change**.

In other words, you can swap "equals for equals".

Note This is the foundation of **equational reasoning**, which lets us reason about code like we would reason about math.

Another way to think of referential transparency is as the **refactoring lemma**. Refactoring is easy, safe, and mathematically guaranteed.

For instance, suppose that you are the engineering manager at a large tech firm named after a red fruit that grows on apple trees. You hire an intern for the summer, and you review their first PR, and you see:

```
if flagIsSet then (
  flagisSet andalso (
    if permissionsGranted then
      true
    else
      permissionsGranted
  )
) else false
```

What the hell is this.

The intern didn't remember their equivalences!

For instance, if we are within the `then` branch of an if expression, we know that the expression we cased on must be equivalent to `true`. The same is true for the `else` branch and `false`.

Let's do some refactoring based on equivalences:

```
if flagIsSet then (
  true andalso (
    if permissionsGranted then
      true
    else
      false
  )
) else false
```

Well, now we see `true andalso e`, which we know should always evaluate to `e`. So let's simplify again:

```
if flagIsSet then (
  if permissionsGranted then
    true
  else
    false
) else false
```

We also see that we have `if e then true else false`, which we learned a few slides ago is a cardinal sin. So let's get rid of that:

```
if flagIsSet then permissionsGranted
else false
```

We can make a final observation that this condition is only `true` when both `flagIsSet` and `permissionsGranted` are true. Therefore, we can simply write it as:

```
flagIsSet andalso permissionsGranted
```

Note Referential transparency saves lives.

Up until this point, we've said that a value is a "final answer". You should be able to tell if values are the same just by looking at them, like with `2` and `2`, or with `(3, 4)` and `(3, 4)`.

But what about functions?

Are `fn (x : int) => x + x` and `fn (x : int) => 2 * x` extensionally equivalent?

It's now hard to tell, because these lambda expressions are values, but it's not obvious if they are extensionally equivalent or not.

Functions necessitate their own rule for extensional equivalence.

Def Two functions `f : t1 -> t2` and `g : t1 -> t2` are **extensionally equivalent** if for all values `x : t1`, `f x` $\cong$ `g x`.

In other words, two functions are equivalent if **for all inputs, they give equivalent outputs**.

Note We specified that `f x` $\cong$ `g x`, not that `f x` and `g x` reduce to the same value. Why?

The reason why is simple - some functions do not have defined outputs!

We described functions in SML as mathematical functions, which is true, but not exactly in the same way as some mathematical functions, such as $+$, or sin.

SML functions can be partial, that is, undefined on some input. There may be `f` and `x` such that there is no `v` where `f x` $\hookrightarrow$ `v`.

Def We say that `f : t1 -> t2` is **total** if for all values `v : t1`, there is a value `v' : t2` such that `f v` $\hookrightarrow$ `v'`.

For example, the SML functions `+`, `not`, and `^` are all total.

Concepts like purity, extensional equivalence, and totality are all just means to an end, which is being able to **specify** the behavior of code.

In particular, for functions, we are interested in writing **descriptive** code, that accurately reflects the function's behavior.

To that end, it is helpful to write alongside a function the conditions which must be true **prior** to calling the function, and must be true **after** calling the function.

```
(* REQUIRES: x is not 0 *)
(* ENSURES: divide x is total *)
fun divide (x : int) : int = 2 div x
```

In this class, our way of writing specifications will follow the five-step methodology.

There are five components to this methodology, shockingly:

1. the function's type
2. the `REQUIRES` clause (preconditions)
3. the `ENSURES` clause (postconditions)
4. the function's definition
5. test cases!

We will frequently annotate our functions this way this semester.

We use comments in the form of `REQUIRES` and `ENSURES` to describe what must be true of the inputs the function receives, and what then is guaranteed to hold of the function's behavior.

It is often unrealistic to have a function which has meaning on *every* possible input (like `div`, or square root, or logarithm). The `REQUIRES` helps to describe the range of "relevant inputs", and the `ENSURES` helps to describe what the function does.

These "contracts" pop up in real code all the time:

- this function must be called with only safe values,
- this library can only be invoked in single-threaded programs,
- this API is not guaranteed to work with non-ASCII characters

It is important that code is documented so users and maintainers know pertinent information about it!

The last piece of the formula is writing tests.[3]

In this class, we will use a 150-specific testing framework, where we write something like:

```
val () = Test.int ("test1", 2 + 2, 1 + 3)
```

which will raise an exception if the second and third parts of the tuple do not evaluate to the same value.

Note I realize writing tests sucks. So does life, sometimes.

[3] There's a lot I could say here about the importance of writing tests. Test-driven development is a real thing, and especially in imperative languages, it's very important to have a solid backbone of tests to make sure you don't accidentally regress behavior when introducing a small change. However, this is a class on functional programming, and not software engineering, so I'll decline to comment for now.[4]

[4] This is a really long footnote.

Now that we have all these tools for mathematically analyzing code, let's look at a specific example. Let's take the factorial function that we wrote earlier.

```
fun fact (0 : int) : int = 1
  | fact n = n * fact (n - 1)
```

Question Is `fact` total?

Answer: It is not.

The `fact` function loops forever on a negative input.

But for our intents and purposes, we don't really care about what `fact` does on negative inputs, anyways. So let's restrict our domain of interest to strictly non-negative numbers.

```
(* fact : int -> int *)
(* REQUIRES: n >= 0 *)
(* ENSURES: fact n evaluates to the nth factorial *)
fun fact (0 : int) : int = 1
  | fact n = n * fact (n - 1)
```

Lesson Oftentimes, we are interested in only a subset of the domain of a function, and we only get to make interesting claims about its behavior when we restrict our attention to it.

Instead of putting specifications in comments above the code, when writing code within these slides, they will often be conveyed via specification blocks.

For instance, for the `fact` function, I would instead write:

```
fact : int -> int
```
REQUIRES: `n >= 0`
ENSURES: `fact n` evaluates to $n!$

In this lecture, we learned a lot about some of the core tools that we have at our disposal when writing SML programs!

We also learned about some of the more mathematical underpinnings of the language, which comes up in **binding and scope**, as well as our ability to reason about code via using concepts like **referential transparency**, **totality**, and **specifications**.

We also saw a little bit of the interplay between **patterns** and **types**. We will develop this more in the future.

Thank you!