CS 251 Part 2:
What's in a Type

Standard **ML**
and Static Types

Topics

- Standard ML basics
- Static type system: types and type-checking rules

ML: Meta-Language for Theorem-Proving

Dana Scott, 1969

Logic of **Computable Functions (LCF):** for stating theorems about programs

Robin Milner, 1972

Logic for **Computable Functions (LCF):** automated theorem proving for LCF

Theorem proving is a hard search problem.

ML: Meta-Language for writing programs (tactics) to find proofs of theorems (about other programs)

Proof Tactic: *Partial* function from formula to proof.

Guides proof search, resulting in one of:

- find and return proof
- never terminate
- report an error

Language Support for Tactics

Static type system
- guarantee correctness of generated proof

Exception handling
- deal with tactics that fail (Turing Award)
- make failure explicit, force programmer to deal with it

First-class/higher-order functions
- compose other tactics

Defining ML

- Focus on static types.
- New syntax.
- Highly familiar semantics
  - Formal definitions only for the new/different.
  - Some of our simplifications in defining Racket match SML perfectly.
- Move faster since we share some formal experience now.

An ML program is a sequence of bindings.

```
(* My first ML program *)

val x = 34

val y = 17

val z = (x + y) + (y + 2)

val q = z + 1

val abs_of_z = if z < 0 then 0 – z else z

val abs_of_z_simpler = abs z

(* comment: ML has (* nested comments! *) *)
```


Bindings, types, and environments

A program is a sequence of *bindings.*

Bindings build ***two*** environments:
- ***static*** *environment* maps variable to type ***before evaluation***
- ***dynamic*** *environment* maps variable to value ***during evaluation***

Type-check each binding in order:
- using ***static*** *environment* produced by previous bindings
- and extending it with a binding from variable to type

Evaluate each binding in order:
- using ***dynamic*** *environment* produced by previous bindings
- and extending it with a binding from variable to value

SML syntax starter

Bindings

```
b ::= val x = e
    | fun x (x : t) = e
```

Types

```
t ::= bool | int | real | string
    | (t) | t * t | t -> t | …
```

Expressions: `e ::= …`
Identifiers: `x`

Meta-syntax
Type environments

```
T ::= . | x : t, T
```


Type-checking judgments

Bindings:

`T ⊢ b % T'`

Under static environment `T`, binding `b` type-checks and produces extended static environment `T'`.

Expressions:

`T ⊢ e : t`

Under static environment `T`, expression `e` type-checks with type `t`.

Variable bindings

Optional semicolon can improve messages for syntax errors.

Syntax: `val x = e`   `val x = e;`

variable name   expression

Type checking: `T ⊢ b % T'`

If the expression, `e`, type-checks with type `t` under the current static environment, `T`, then the binding is well-typed and extends the static environment with typing `x : t`.

```
        T ⊢ e : t
--------------------------- [t-val]
T ⊢ val x = e % x : t, T
```

Evaluation (only if it type-checks):

`E ⊢ b ⇓ E'`

```
        E ⊢ e ↓ v
--------------------------- [e-val]
E ⊢ val x = e ⇓ x ↦ v, E
```


Expression type-checking rules

`T ⊢ e : t`

Value examples:

```
T ⊢ 34 : int            T ⊢ ~1 : int

        T ⊢ 3.14159 : real

T ⊢ true : bool         T ⊢ false : bool
```

Variables:

Under static environment `T`, variable reference `x` type-checks with type `t` if the static environment maps `x` to `t`.

```
  T(x) = t
------------ [t-var]
 T ⊢ x : t
```

Binary expression type-checking rules

Syntax: `e1 + e2` `e1 < e2` `e1 = e2` `e1 <> e2`

Type checking: `T ⊢ e : t`

```
T ⊢ e1 : int
T ⊢ e2 : int
----------------------  [t-add]
T ⊢ e1 + e2 : int

T ⊢ e1 : int
T ⊢ e2 : int
----------------------  [t-less]
T ⊢ e1 < e2 : bool

T ⊢ e1 : t   <- same type
T ⊢ e2 : t   <-
----------------------  [t-eq]
T ⊢ e1 = e2 : bool

T ⊢ e1 : t
T ⊢ e2 : t
----------------------  [t-ne]
T ⊢ e1 <> e2 : bool
```

(One more restriction later)

`if` expressions

Syntax: `if e1 then e2 else e3`

Type checking: `T ⊢ e : t`

```
T ⊢ e1 : bool
T ⊢ e2 : t        (same type)
T ⊢ e3 : t
---------------------------------  [t-if]
T ⊢ if e1 then e2 else e3 : t
```

Evaluation: `E ⊢ e ↓ v`

```
E ⊢ e1 ↓ true
E ⊢ e2 ↓ v2
---------------------------------  [e-if-true]
E ⊢ if e1 then e2 else e3 ↓ v2

E ⊢ e1 ↓ false
E ⊢ e3 ↓ v3
---------------------------------  [e-if-false]
E ⊢ if e1 then e2 else e3 ↓ v3
```


ML static types and evaluation

Soundness

A program that type-checks never encounters a dynamic type error when evaluated.

Evaluation Rules

Same as **our** Racket evaluation rules (for ML syntax) **except there is no dynamic type checking.**

Function examples

```
                 (* Anonymous function expression *)
val double = fn (x : int) => x + x
val four = double (2)

(* Function binding *)
fun pow (x : int, y : int) =
  if y = 0
  then 1
  else x * pow (x,y-1)

fun cube (x : int) =
  pow (x,3)

val sixtyfour = cube (four)
val fortytwo =
  pow (2,2+2) + pow (4,2) + cube (2) + 2
```

Function type syntax

```
(t1 * … * tn) -> t
```

argument types — result type

A function that takes *n* arguments of types `t1` … `tn` and returns a result of type `t`.

Anonymous function expressions

Syntax: `fn (x1 : t1, … , xn : tn) => e`

argument variable names (`xi`) and types (`ti`) — body expression

Type checking: `T ⊢ e : t`

If the function body, `e`, type-checks with type `t`, under the current static environment, `T`, extended with the argument types, then the function type-checks with type `(t1 * … * tn) -> t` under the current static environment, `T`.

```
x1:t1, …, xn:tn, T ⊢ e : t
------------------------------------------------------ [t-fn]
T ⊢ fn (x1:t1,…,xn:tn) => e : (t1 * … * tn) -> t
```

function type

Function bindings

Syntax: `fun x0 (x1 : t1, … , xn : tn) = e`

function variable name — argument variable names (`xi`) and types (`ti`) — body expression

Type checking: `T ⊢ b % T'`

Otherwise equivalent to `val x0 = fn (x1 : t1, …, xn : tn) => e`

argument typings — function typing (for recursion)

```
x1:t1, …, xn:tn, x0 : (t1*…*tn) -> t, T ⊢ e : t
------------------------------------------------------ [t-fun]
  T ⊢ fun x0 (x1 : t1, …, xn : tn) = e
                % T, x0 : (t1 * … * tn) -> t
```

Evaluation: same as Racket.

Generalize later.

Function application

Syntax: `e0 (e1, …, en)`

expressions

Type checking: `T ⊢ e : t`

```
T ⊢ e0 : (t1 * … * tn) -> t
T ⊢ e1 : t1
…
T ⊢ en : tn
------------------------------------------------------ [t-apply]
T ⊢ e0 (e1, …, en) : t
```

```
(* Example *)
fun pow (x : int, y : int) =
  if y = 0
  then 1
  else x * pow (x,y-1)
```

Function application

Syntax: **`e0 (e1, …, en)`**

Evaluation:

1. Under the current dynamic environment, `E`, evaluate `e0` to a function closure value ⟨`E'`, `fn (x1, …, xn) => e`⟩.
   - **No dynamic type-checking:** Static type-checking guarantees `e0`'s result value will be a function closure taking parameters `x1,…,xn` of types matching those of `e1,…,en`.
2. Under the current dynamic environment, `E`, evaluate argument expressions `e1, …, en` to values `v1, …, vn`
3. The result is the result of evaluating the closure body, `e`, under the closure environment, `E'`, extended with argument bindings: `x1 ⟼ v1, …, xn ⟼ vn`.

Function application

Syntax: **`e0 (e1, …, en)`**

Evaluation: `E ⊢ e ↓ v`

```
E ⊢ e0 ↓ ⟨E', fn (x1, …, xn) => e⟩
E ⊢ e1 ↓ v1
    …
E ⊢ en ↓ vn
x1 ⟼ v1, …, xn ⟼ vn, E' ⊢ e ↓ v
-------------------------------------- [e-apply]
E ⊢ e0 (e1, …, en) ↓ v
```


Watch out

Odd error messages for function-argument syntax errors

`*` in type syntax is not arithmetic

- Example: **`int * int -> int`**
- In expressions, `*` is multiplication: **`x * pow(x,y-1)`**

Cannot refer to later function bindings

- Helper functions must come before their uses
- Special construct for mutual recursion (later)

`let` expressions ... *but*

Syntax: **`let b in e end`**

- `b` is any *binding* and `e` is any *expression*

Type checking: `T ⊢ e : t`

```
T  ⊢ b % T'
T' ⊢ e : t
------------------------- [t-let]
T ⊢ let b in e end : t
```

Evaluation: `E ⊢ e ↓ v`

```
E  ⊢ b ⇓ E'
E' ⊢ e ↓ v
------------------------- [e-let]
E ⊢ let b in e end ↓ v
```

`let` is sugar

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

desugars to:

```
((fn (x) => e2) e1)
```

(Rules [t-let] and [e-let] are not needed.)

Multi-binding let:

```
let b1 b2 … bn in e end
```

Like Racket's let*

desugars to:

```
let b1 in let b2 in … let bn in e end … end end
```