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
\[ b ::= \text{val } x = e \]
\[ \quad | \quad \text{fun } x \ (x : t) = e \]

Types
\[ t ::= \text{bool} \mid \text{int} \mid \text{real} \mid \text{string} \]
\[ \quad | \quad (t) \mid \ t \ast t \mid \ t \rightarrow t \mid \ldots \]

Expressions:
\[ e ::= \ldots \]

Identifiers:
\[ x \]

Meta-syntax
Type environments
\[ T ::= . \mid x : t, T \]
Type-checking judgments

Bindings:

\[ T \vdash b : T' \]

Under static environment \( T \), binding \( b \) type-checks and produces extended static environment \( T' \).

Expressions:

\[ T \vdash e : t \]

Under static environment \( T \), expression \( e \) type-checks with type \( t \).
Variable bindings

Syntax:
\[ \text{val } x = e \]
\[ \text{val } x = e; \]

Variable name \( x \)
Expression \( e \)

Type checking:
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 \( x : t \).

\[
\frac{T \vdash e : t}{T \vdash \text{val } x = e \% x : t, T} \quad \text{[t-val]}
\]

Evaluation (only if it type-checks):

\[
\frac{E \vdash b \downarrow E'}{E \vdash \text{val } x = e \% x \mapsto v, E} \quad \text{[e-val]}
\]

Optional semicolon can improve messages for syntax errors.
Expression type-checking rules

\[
T \vdash e : t
\]

Value examples:
\[
T \vdash 34 : \text{int} \quad T \vdash \neg 1 : \text{int}
\]
\[
T \vdash 3.14159 : \text{real}
\]
\[
T \vdash \text{true} : \text{bool} \quad T \vdash \text{false} : \text{bool}
\]

Variables:
Under static environment \( T \), variable reference \( x \) type-checks with type \( t \) if the static environment maps \( x \) to \( t \).

\[
T(x) = t \quad \frac{[t \text{-var}]}{T \vdash x : t}
\]
Binary expression type-checking rules

Syntax: $e_1 + e_2$, $e_1 < e_2$, $e_1 = e_2$, $e_1 <> e_2$

Type checking:

- **[t-add]**
  
  $\begin{align*}
  T \vdash e_1 : \text{int} \\
  T \vdash e_2 : \text{int} \\
  \hline
  T \vdash e_1 + e_2 : \text{int}
  \end{align*}$

- **[t-less]**
  
  $\begin{align*}
  T \vdash e_1 : \text{int} \\
  T \vdash e_2 : \text{int} \\
  \hline
  T \vdash e_1 < e_2 : \text{bool}
  \end{align*}$

- **[t-eq]**
  
  $\begin{align*}
  T \vdash e_1 : t \\
  T \vdash e_2 : t \\
  \hline
  T \vdash e_1 = e_2 : \text{bool}
  \end{align*}$

- **[t-ne]**
  
  $\begin{align*}
  T \vdash e_1 : t \\
  T \vdash e_2 : t \\
  \hline
  T \vdash e_1 <> e_2 : \text{bool}
  \end{align*}$

(One more restriction later)
**if expressions**

**Syntax:**

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

**Type checking:**

\[
T \vdash e_1 : \text{bool} \\
T \vdash e_2 : t \\
T \vdash e_3 : t \\
T \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t
\]

**Evaluation:**

\[
E \vdash e_1 \downarrow \text{true} \\
E \vdash e_2 \downarrow v_2 \\
E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow v_2
\]

\[
E \vdash e_1 \downarrow \text{false} \\
E \vdash e_3 \downarrow v_3 \\
E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow v_3
\]
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

\[(t_1 \ast \ldots \ast t_n) \rightarrow t\]

A function that takes \(n\) arguments of types \(t_1 \ldots t_n\) and returns a result of type \(t\).
Anonymous function expressions

Syntax: $\text{fn } (x_1 : t_1, \ldots, x_n : t_n) \Rightarrow e$

Type checking: $T \vdash e : t$

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 $(t_1 \ast \ldots \ast t_n) \Rightarrow t$ under the current static environment, $T$.

$x_1 : t_1, \ldots, x_n : t_n, T \vdash e : t$

$T \vdash \text{fn } (x_1 : t_1, \ldots, x_n : t_n) \Rightarrow e : (t_1 \ast \ldots \ast t_n) \Rightarrow t$

function type
Function bindings

Syntax:

\[
\text{fun } x_0 \ (x_1 : t_1, \ldots, x_n : t_n) = e
\]

Type checking:

\[
T \vdash b \% T'
\]

Otherwise equivalent to

\[
\text{val } x_0 = \text{fn} \ (x_1 : t_1, \ldots, x_n : t_n) \Rightarrow e
\]

Evaluation: same as Racket.
Function application

Syntax: \[ e_0 \ (e_1, \ldots, \ e_n) \]

Type checking:

\[
\begin{align*}
T \vdash e_0 &: (t_1 \times \cdots \times t_n) \rightarrow t \\
T \vdash e_1 &: t_1 \\
& \vdots \\
T \vdash e_n &: t_n \\
T \vdash e_0 \ (e_1, \ldots, \ e_n) &: t
\end{align*}
\]

(* Example *)

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

Generalize later.
Function application

Syntax: \[ e_0 \ (e_1, \ldots, \ e_n) \]

Evaluation:

1. Under the current dynamic environment, \( E \), evaluate \( e_0 \) to a function closure value \( \langle E', \ \text{fn} \ (x_1, \ldots, x_n) \Rightarrow e \rangle \).
   - No dynamic type-checking: Static type-checking guarantees \( e_0 \)'s result value will be a function closure taking parameters \( x_1, \ldots, x_n \) of types matching those of \( e_1, \ldots, e_n \).

2. Under the current dynamic environment, \( E \), evaluate argument expressions \( e_1, \ldots, e_n \) to values \( v_1, \ldots, v_n \)

3. The result is the result of evaluating the closure body, \( e \), under the closure environment, \( E' \), extended with argument bindings: \( x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \).
Function application

Syntax: \( e0 \ (e1, \ldots, \ en) \)

Evaluation: \( E \vdash e \downarrow v \)

\[
E \vdash e0 \downarrow \langle E', \ fn \ (x1, \ldots, \ xn) => e \rangle \\
E \vdash e1 \downarrow v1 \\
\ldots \\
E \vdash en \downarrow vn \\
x1 \mapsto v1, \ldots, xn \mapsto vn, E' \vdash e \downarrow v ~ [\text{e-apply}] \\
E \vdash e0 \ (e1, \ldots, \ en) \downarrow v
\]
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

Syntax: \texttt{let } b \texttt{ in } e \texttt{ end}

– \texttt{b} is any \textit{binding} and \texttt{e} is any \textit{expression}

Type checking:

\[
\frac{T \vdash b : T'}{T' \vdash e : t} \quad \frac{T' \vdash e : t}{T \vdash \text{let } b \text{ in } e \text{ end} : t} \quad \text{[t-let]}
\]

Evaluation:

\[
\frac{E \vdash b \Downarrow E'}{E' \vdash e \Downarrow v} \quad \frac{E' \vdash e \Downarrow v}{E \vdash \text{let } b \text{ in } e \text{ end} \Downarrow v} \quad \text{[e-let]}
\]
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
```
desugars to:
```
let b1 in let b2 in ... let bn in e end ... end end
```