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**

\[
\begin{align*}
  b ::= & \text{val } x = e \\
        & \text{fun } x (x : t) = e
\end{align*}
\]

**Types**

\[
\begin{align*}
  t ::= & \text{bool | int | real | string} \\
        & (t) | t * t | t \rightarrow t | \ldots
\end{align*}
\]

**Expressions:**

\[
\begin{align*}
  e ::= & \ldots
\end{align*}
\]

**Identifiers:**

\[x\]

**Meta-syntax**

Type environments

\[
T ::= . | x : t, T
\]

---

**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:**

\[
\text{val } x = e \quad \text{val } x = e;
\]

**Type checking:**

\[
T \vdash b \; \& \; T'
\]

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 \(x : t\).

\[
\begin{align*}
T \vdash e : t \\
T \vdash \text{val } x = e \; \& \; x : t, T
\end{align*}
\]

Evaluation (only if it type-checks):

\[
\begin{align*}
E \vdash b \; \& \; E' \\
E \vdash e \; \downarrow \; v \\
E \vdash \text{val } x = e \; \downarrow \; x \mapsto v, E
\end{align*}
\]

---

**Expression type-checking rules**

**Value examples:**

\[
\begin{align*}
T \vdash 34 : \text{int} & \quad T \vdash \sim 1 : \text{int} \\
& \quad T \vdash 3.14159 : \text{real} \\
T \vdash \text{true} : \text{bool} & \quad T \vdash \text{false} : \text{bool}
\end{align*}
\]

**Variables:**

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

\[
T(x) = t
\]

\[
T \vdash x : t
\]
Binary expression type-checking rules

Syntax: $e_1 + e_2 \quad e_1 < e_2 \quad e_1 = e_2 \quad e_1 <> e_2$

Type checking:

- $T \vdash e_1 : t$
- $T \vdash e_2 : t$

- $T \vdash e_1 + e_2 : int$  \[t\text{-add}\]
- $T \vdash e_1 < e_2 : bool$  \[t\text{-less}\]
- $T \vdash e_1 = e_2 : bool$  \[t\text{-eq}\]
- $T \vdash e_1 <> e_2 : bool$  \[t\text{-ne}\]

(One more restriction later)

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.

if expressions

Syntax: if $e_1$ then $e_2$ else $e_3$

Type checking:

- $T \vdash e_1 : bool$
- $T \vdash e_2 : t$
- $T \vdash e_3 : t$

Evaluation:

- $E \vdash e_1 \downarrow \text{true}$
- $E \vdash e_2 \downarrow v_2$
- $E \vdash e_3 \downarrow v_3$

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

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

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

Anonymous function expressions

Syntax:

\[fn \; (x_1 : t_1, \ldots, x_n : t_n) \Rightarrow e\]

Type checking:

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 \((t_1 \times \ldots \times t_n) \rightarrow t\) under the current static environment, \(T\).

Function bindings

Syntax:

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

Type checking:

Otherwise equivalent to

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

Evaluation: same as Racket.

Function application

Syntax:

\[e_0 \; (e_1, \ldots, e_n)\]

Type checking:

Generalize later.

(* Example *)

\[
\text{fun pow } (x : \text{int}, y : \text{int}) = \\
\text{if } y = 0 \\
\text{then } 1 \\
\text{else } x \times (\text{pow } (x, y-1))
\]
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 \( \langle E', \ fn \ (x_1, \ldots, x_n) \Rightarrow e \rangle \).
   - **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: \( x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \).

Watch out

Odd error messages for function-argument syntax errors

* in type syntax is not arithmetic
  - Example: \( \text{int} * \text{int} \rightarrow \text{int} \)
  - In expressions, * is multiplication: \( x * \text{pow}(x, y-1) \)

Cannot refer to later function bindings
  - Helper functions must come before their uses
  - Special construct for mutual recursion (later)

let expressions

Syntax: \( \text{let } b \text{ in } e \text{ end} \)

... but

Type checking:

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

Evaluation:

\[
E \vdash e \downarrow v \\
E' \vdash e \downarrow v \\
E \vdash \text{let } b \text{ in } e \text{ end} \downarrow v
\]  
\[\text{[e-let]}\]
**let is sugar**

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

desugars to:

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

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

Multi-binding let:

```ml
let b1 b2 ... bn in e end
```

desugars to:

```ml
let b1 in let b2 in ... let bn in e end ... end end
```

Like Racket's let*