CS 251 Fall 2019
Principles of Programming Languages
Ben Wood

CS 251 Part 2: What's in a Type

https://cs.wellesley.edu/~cs251/f19/

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: **M**eta-**L**anguage 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 → *x*; expression → *e*

Type checking: $\boxed{T \vdash b \,\%\, T'}$

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`.

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

Evaluation (only if it type-checks):

$\boxed{E \vdash b \Downarrow E'}$

$$\frac{E \vdash e \downarrow v}{E \vdash \mathtt{val}\ x = e \Downarrow x \mapsto v,\ E}\ \text{[e-val]}$$

Expression type-checking rules

$\boxed{\texttt{T} \vdash \texttt{e : t}}$

Value examples:

$\texttt{T} \vdash \texttt{34 : int}$ $\qquad$ $\texttt{T} \vdash \texttt{\~{}1 : int}$

$\texttt{T} \vdash \texttt{3.14159 : real}$

$\texttt{T} \vdash \texttt{true : bool}$ $\qquad$ $\texttt{T} \vdash \texttt{false : bool}$

Variables:

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

$$\frac{\texttt{T(x) = t}}{\texttt{T} \vdash \texttt{x : t}}\ \text{[t-var]}$$

Binary expression type-checking rules

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

Type checking: $\boxed{\mathtt{T} \vdash \mathtt{e} : \mathtt{t}}$

$$\frac{\begin{array}{l} \mathtt{T} \vdash \mathtt{e1} : \mathtt{int} \\ \mathtt{T} \vdash \mathtt{e2} : \mathtt{int} \end{array}}{\mathtt{T} \vdash \mathtt{e1} + \mathtt{e2} : \mathtt{int}}\ \text{[t-add]} \qquad \frac{\begin{array}{l} \mathtt{T} \vdash \mathtt{e1} : \mathtt{int} \\ \mathtt{T} \vdash \mathtt{e2} : \mathtt{int} \end{array}}{\mathtt{T} \vdash \mathtt{e1} < \mathtt{e2} : \mathtt{bool}}\ \text{[t-less]}$$

$$\frac{\begin{array}{l} \mathtt{T} \vdash \mathtt{e1} : \mathbf{t} \\ \mathtt{T} \vdash \mathtt{e2} : \mathbf{t} \end{array}}{\mathtt{T} \vdash \mathtt{e1} = \mathtt{e2} : \mathtt{bool}}\ \text{[t-eq]} \qquad \frac{\begin{array}{l} \mathtt{T} \vdash \mathtt{e1} : \mathtt{t} \\ \mathtt{T} \vdash \mathtt{e2} : \mathtt{t} \end{array}}{\mathtt{T} \vdash \mathtt{e1} <> \mathtt{e2} : \mathtt{bool}}\ \text{[t-ne]}$$

same type

(One more restriction later)

`if` expressions

Syntax: `if` ***e1*** `then` ***e2*** `else` ***e3***

Type checking:

$$\boxed{T \vdash e : t}$$

same type

$$\frac{T \vdash e1 : \texttt{bool} \qquad T \vdash \mathbf{e2} : \mathbf{t} \qquad T \vdash \mathbf{e3} : \mathbf{t}}{T \vdash \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 : \mathbf{t}} \;\text{[t-if]}$$

Evaluation:

$$\boxed{E \vdash e \downarrow v}$$

$$\frac{E \vdash e1 \downarrow \texttt{true} \qquad E \vdash \mathbf{e2} \downarrow \mathbf{v2}}{E \vdash \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 \downarrow \mathbf{v2}} \;\text{[e-if-true]}$$

$$\frac{E \vdash e1 \downarrow \texttt{false} \qquad E \vdash \mathbf{e3} \downarrow \mathbf{v3}}{E \vdash \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 \downarrow \mathbf{v3}} \;\text{[e-if-false]}$$

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

$$(\underbrace{\texttt{t1 * … * tn}}_{\text{argument types}})\ \texttt{->}\ \underbrace{\texttt{t}}_{\text{result type}}$$

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

Anonymous function expressions

Syntax: $\texttt{fn}\ \underbrace{(\boldsymbol{x1} : \boldsymbol{t1}, \ldots , \boldsymbol{xn} : \boldsymbol{tn})}_{\text{argument variable names (}\texttt{xi}\text{) and types (}\texttt{ti}\text{)}}\ \texttt{=>}\ \underbrace{e}_{\text{body expression}}$

Type checking: $\boxed{\texttt{T} \vdash \texttt{e : t}}$

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.

$$\frac{\texttt{x1:t1, …, xn:tn, T} \vdash \texttt{e : t}}{\texttt{T} \vdash \texttt{fn (x1:t1,…,xn:tn) => e :}\ \underbrace{\texttt{(t1 * … * tn) -> t}}_{\text{function type}}}\ [\text{t-fn}]$$

Function bindings

Syntax:

$$\texttt{fun}\ \underbrace{\textit{x0}}_{\text{function variable name}}\ (\underbrace{\textit{x1} : \textit{t1}, \ldots, \textit{xn} : \textit{tn}}_{\text{argument variable names (xi) and types (ti)}}) = \underbrace{\textit{e}}_{\text{body expression}}$$

Type checking: $\boxed{\texttt{T} \vdash \texttt{b} \ \% \ \texttt{T'}}$

Otherwise equivalent to

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

$$\frac{\overbrace{\texttt{x1:t1, …, xn:tn,}}^{\text{argument typings}}\ \overbrace{\texttt{x0 : (t1*…*tn) -> t,}}^{\text{function typing (for recursion)}}\ \texttt{T} \vdash \texttt{e : t}}{\texttt{T} \vdash \texttt{fun x0 (x1 : t1, …, xn : tn) = e} \ \% \ \texttt{T, x0 : (t1 * … * tn) -> t}}\ \text{[t-fun]}$$

Evaluation: same as Racket.

Generalize later.

Function application

expressions

Syntax: ***`e0 (e1, …, en)`***

Type checking: $\boxed{\mathtt{T \vdash e : t}}$

$$\frac{\begin{array}{l}\mathtt{T \vdash e0 : (t1 * \ldots * tn) \to t} \\ \mathtt{T \vdash e1 : t1} \\ \ldots \\ \mathtt{T \vdash en : tn}\end{array}}{\mathtt{T \vdash e0\ (e1, \ldots, en) : t}}\ \text{[t-apply]}$$

```
(* 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: $\boxed{\mathtt{E} \vdash \mathtt{e} \downarrow \mathtt{v}}$

$$\frac{\begin{array}{l}\mathtt{E} \vdash \mathtt{e0} \downarrow \langle \mathtt{E'},\ \mathtt{fn}\ (\mathtt{x1},\ \dots,\ \mathtt{xn})\ \texttt{=>}\ \mathtt{e}\rangle \\ \mathtt{E} \vdash \mathtt{e1} \downarrow \mathtt{v1} \\ \qquad \dots \\ \mathtt{E} \vdash \mathtt{en} \downarrow \mathtt{vn} \\ \mathtt{x1} \longmapsto \mathtt{v1},\ \dots,\ \mathtt{xn} \longmapsto \mathtt{vn},\ \mathtt{E'} \vdash \mathtt{e} \downarrow \mathtt{v}\end{array}}{\mathtt{E} \vdash \mathtt{e0}\ (\mathtt{e1},\ \dots,\ \mathtt{en}) \downarrow \mathtt{v}}\ \text{[e-apply]}$$

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:

$$\boxed{T \vdash e : t}$$

$$\frac{\begin{array}{l} T \;\;\vdash b \;\%\; T' \\ T' \vdash e : t \end{array}}{T \vdash \texttt{let } b \texttt{ in } e \texttt{ end} : t}\;[\text{t-let}]$$

Evaluation:

$$\boxed{E \vdash e \downarrow v}$$

$$\frac{\begin{array}{l} E \;\;\vdash b \Downarrow E' \\ E' \vdash e \downarrow v \end{array}}{E \vdash \texttt{let } b \texttt{ in } e \texttt{ end} \downarrow v}\;[\text{e-let}]$$

`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`

Like Racket's let*

desugars to:

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