



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

$b ::= \text{val } x = e$
 $\quad \mid \text{fun } x (x : t) = e$

Types

$t ::= \text{bool} \mid \text{int} \mid \text{real} \mid \text{string}$
 $\quad \mid (t) \mid t * t \mid t \rightarrow t \mid \dots$

Expressions: $e ::= \dots$

Identifiers: x

Meta-syntax

Type environments

$T ::= \cdot \mid x : t, T$

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 expression

Optional semicolon can improve messages for syntax errors.

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 $x : t$. ★

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

Evaluation (only if it type-checks):

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

Expression type-checking rules

$T \vdash e : t$

Value examples:

$T \vdash 34 : \text{int}$

$T \vdash \sim 1 : \text{int}$

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

$T \vdash \text{true} : \text{bool}$

$T \vdash \text{false} : \text{bool}$

Variables:

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

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

Binary expression type-checking rules

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

Type checking: $T \vdash e : t$

$$\frac{T \vdash e1 : \text{int} \quad T \vdash e2 : \text{int}}{T \vdash e1 + e2 : \text{int}} \text{ [t-add]} \quad \frac{T \vdash e1 : \text{int} \quad T \vdash e2 : \text{int}}{T \vdash e1 < e2 : \text{bool}} \text{ [t-less]}$$

$$\frac{T \vdash e1 : t \quad T \vdash e2 : t}{T \vdash e1 = e2 : \text{bool}} \text{ [t-eq]} \quad \frac{T \vdash e1 : t \quad T \vdash e2 : t}{T \vdash e1 <> e2 : \text{bool}} \text{ [t-ne]}$$

(One more restriction later)

SML and Static Types 15

if expressions

Syntax: $\text{if } e1 \text{ then } e2 \text{ else } e3$

Type checking:

 $T \vdash e : t$

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

same type

Evaluation:

 $E \vdash e \downarrow v$

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

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

SML and Static Types 16

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.

SML and Static Types 17

Function examples

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

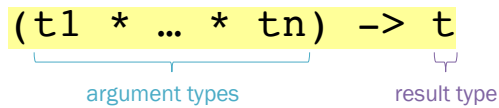
```
(* Function binding *)
fun pow (x : int, y : int) =
  if y = 0
  then 1
  else x * pow (x,y-1)
```

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

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

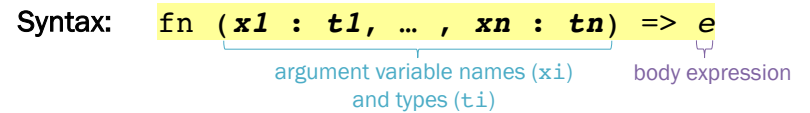
SML and Static Types 18

Function type syntax



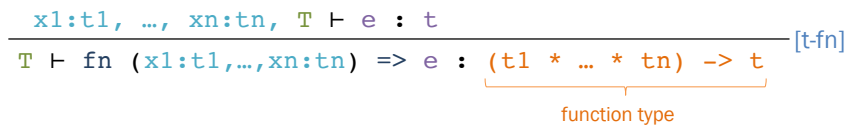
A function that takes n arguments of types $t_1 \dots t_n$ and returns a result of type t .

Anonymous function expressions

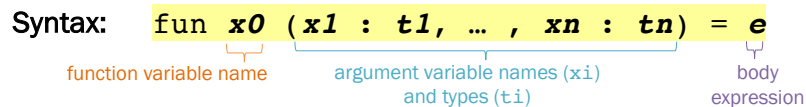


Type checking: $\mathbb{T} \vdash e : t$

If the function body, e , type-checks with type t , under the current static environment, \mathbb{T} , extended with the argument types, then the function type-checks with type $(t_1 * \dots * t_n) \rightarrow t$ under the current static environment, \mathbb{T} .

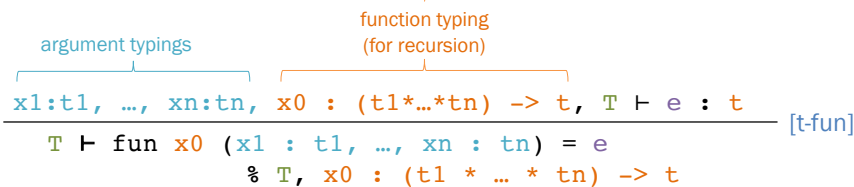


Function bindings



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

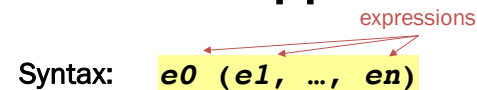
Otherwise equivalent to $\text{val } x_0 = \text{fn } (x_1 : t_1, \dots, x_n : t_n) \Rightarrow e$



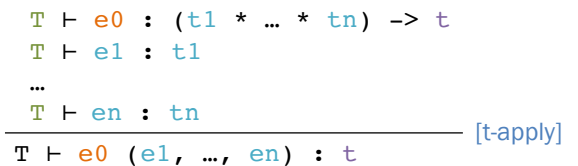
Evaluation: same as Racket.

Generalize later.

Function application



Type checking: $\mathbb{T} \vdash e : t$



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

Function application

Syntax: $e0 (e1, \dots, en)$

Evaluation:

1. Under the current dynamic environment, E , evaluate $e0$ to a function closure value $\langle E', \text{fn } (x1, \dots, xn) \Rightarrow e \rangle$.
 - **No dynamic type-checking:** Static type-checking guarantees $e0$'s result value will be a function closure taking parameters $x1, \dots, xn$ of types matching those of $e1, \dots, en$.
2. Under the current dynamic environment, E , evaluate argument expressions $e1, \dots, en$ to values $v1, \dots, vn$
3. The result is the result of evaluating the closure body, e , under the closure environment, E' , extended with argument bindings: $x1 \mapsto v1, \dots, xn \mapsto vn$.

Function application

Syntax: $e0 (e1, \dots, en)$

Evaluation: $E \vdash e \downarrow v$

$$\begin{array}{l} E \vdash e0 \downarrow \langle E', \text{fn } (x1, \dots, xn) \Rightarrow e \rangle \\ E \vdash e1 \downarrow v1 \\ \dots \\ E \vdash en \downarrow vn \\ \hline x1 \mapsto v1, \dots, xn \mapsto vn, E' \vdash e \downarrow v \quad \text{[e-apply]} \\ \hline E \vdash e0 (e1, \dots, en) \downarrow v \end{array}$$

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:

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

Evaluation:

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

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
```

Like Racket's let*

desugars to:

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