CS320 Handout 31
A Fragment of SML

Assaf Kfoury

8 November 2005

We define a (very small) fragment of SML — call it **mini-ML**. We first present the syntax of the (untyped) underlying programming language, then the syntax of types, and then the rules to combine the underlying language with the types.

1 Syntax of mini-ML

i, j, k	$\in$ int	$::=$	$\cdots \mid \tilde{}3 \mid \tilde{}2 \mid \tilde{}1 \mid 0 \mid 1 \mid 2 \mid \cdots$	
b	$\in$ bool	$::=$	`true` $\mid$ `false`	
x, y, z	$\in$ var	$::=$	$\cdots \mid$ `a` $\mid$ `b` $\mid \cdots \mid$ `temp` $\mid$ `new` $\mid \cdots$	
$\diamond$	$\in$ op	$::=$	`(op +)` $\mid$ `(op * )` $\mid$ `(op =)` $\mid$ `hd` $\mid$ `tl` $\mid$ `(op ::)` $\mid \cdots$	
M, N, P	$\in$ exp	$::=$	$i \mid b \mid x \mid \diamond$	atom
		$\mid$	MN	application
		$\mid$	`fn` x `=>` M	abstraction
		$\mid$	`if` M `then` N `else` P	conditional
		$\mid$	$[M_1, \ldots, M_n]$	list, $n \geqslant 0$
		$\mid$	(M_1, M_2)	pair
		$\mid$	`#1` M $\mid$ `#2` M	projection
		$\mid$	`let val` x_1 `=` M_1 `and` $\cdots$ `and` x_n `=` M_n `in` N `end`	let-val, $n \geqslant 1$
		$\mid$	`let fun` $x_1(y_1)$ `=` M_1 `and` $\cdots$ `and` $x_n(y_n)$ `=` M_n `in` N `end`	let-fun, $n \geqslant 1$

Remark 1 : **mini-ML** is obtained from the syntax of SML by imposing several restrictions. In particular:

1. **mini-ML** includes parallel, but not sequential, versions of `let-val` and `let-fun` — note the keyword "`and`".

2. **mini-ML** excludes tuples of the form $(M_1, \ldots, M_n)$ with arbitrarily many entries, $n \geqslant 0$. It only allows pairs (M_1, M_2), together with the first and second projections, `#1` M and `#2` M.

3. Every variable binding in **mini-ML** consists of exactly one variable, not a tuple of variables, e.g., none of the following expressions is legal in **mini-ML**:

`fn` $(x_1, \ldots, x_n)$ `=>` M	where $n \geqslant 2$ or $n = 0$
`let fun` $x(y_1, \ldots, y_n)$ `=` M `in` N `end`	where $n \geqslant 2$ or $n = 0$

   where M and N are valid **mini-ML** expressions. ∎

Exercise 2 : In the two parts below, assume that $M_1, \ldots, M_n$ and N are valid **mini-ML** expressions.

1. Show that an SML expression using a sequential `let-val` of the form:

   ```
   let val x_1 = M_1
       val x_2 = M_2
       ...
       val x_n = M_n
   in  N   end
   ```

   can be de-sugared into an equivalent expression of **mini-ML**.

2. Show that an SML expression using a sequential `let-fun` of the form:

   ```
   let fun x_1(y_1) = M_1
       fun x_2(y_2) = M_2
       ...
       fun x_n(y_n) = M_n
   in  N   end
   ```

   can be de-sugared into an equivalent expression of **mini-ML**. ∎

Exercise 3 : Let $M_1, \ldots, M_4$ and N be valid **mini-ML** expressions. Show that an SML expression of the form:

```
let fun x_1(y_1) = M_1
    fun x_2(y_2) = M_2
    val x_3      = M_3
    val x_4      = M_4
in  N   end
```

can be de-sugared into an equivalent expression of **mini-ML**. ∎

Exercise 4 : **mini-ML** does not allow arbitrary tuples of the form $(M_1, \ldots, M_n)$ where $n \neq 2$. In the case $n \geqslant 3$, however, we can consider such a tuple to be syntactic sugar for the following valid **mini-ML** expression:

$$(M_1, (M_2, (M_3, \cdots (M_{n-1}, M_n) \cdots)))$$

assuming that $M_1, \ldots, M_n$ are also valid **mini-ML** expressions. Show that the corresponding projections `#1` M, `#2` M, `#3` M, ..., `#`n M can be de-sugared into equivalent **mini-ML** expressions. ∎

Exercise 5 : Show that an SML expression of the form:

```
let fun x(y_1, ..., y_n) = M in N end      where n ⩾ 2 or n = 0
```

can be de-sugared into an equivalent **mini-ML** expression. Assume M is a valid **mini-ML** expression that may mention $y_1, \ldots, y_n$ but not x; and assume N is a valid **mini-ML** expression except that if x occurs in N, then it occurs as $x(M_1, \ldots, M_n)$ for some valid **mini-ML** expressions $M_1, \ldots, M_n$. *Hint*: Consider separately the two cases, $n = 0$ and $n \geqslant 2$, and use the result of the preceding exercise. ∎

2 Syntax of Types

tcons	::=	`int` | `bool`	type constant
$\alpha \in \mathsf{tvar}$	::=	`'a` | `'b` | `'c` | $\cdots$	type variable
$\tau \in \mathsf{mono}$	::=	`int` | `bool` | α	atom
	|	τ_1 `->` τ_2	function type
	|	τ `list`	list type
	|	τ_1 `*` τ_2	pair type
$\sigma \in \mathsf{poly}$	::=	$\{\alpha_1, \ldots, \alpha_n\}.\ \tau$	polymorphic type, $n \geqslant 0$

In a poly type $\{\alpha_1, \ldots, \alpha_n\}.\ \tau$, we say that the type variables $\alpha_1, \ldots, \alpha_n$ are *bound* (or *quantified*). A special case is $n = 0$, for which we write τ instead of $\{\}.\ \tau$.

Note that τ ranges over mono types, σ ranges over poly types, and every mono type τ is viewed as as special case of a poly type — with the convention that $\tau = \{\}.\ \tau$.

Given $\tau \in \mathsf{mono}$, let $\mathsf{FTV}(\tau)$ denote the set of type variables occurring in τ. For example, if $\tau = \alpha_1$ `->` `int` `->` α_2 then $\mathsf{FTV}(\tau) = \{\alpha_1, \alpha_2\}$. We use "FTV" for "free type variables", meaning that every type variable occurring in a mono type is free.

Given $\sigma = \{\alpha_1, \ldots, \alpha_n\}.\ \tau \in \mathsf{poly}$, we define $\mathsf{FTV}(\sigma)$ as $\mathsf{FTV}(\tau) - \{\alpha_1, \ldots, \alpha_n\}$. For example, if $\sigma = \{\alpha_1, \alpha_3\}.\ \tau$ where $\tau = \alpha_1$ `->` `int` `->` α_2, then $\mathsf{FTV}(\sigma) = \{\alpha_2\}$. A type σ is said to be *closed* if $\mathsf{FTV}(\sigma) = \varnothing$.

Definition 6 (Type Instantiation) : Let $\sigma = \{\alpha_1, \ldots, \alpha_n\}.\ \tau$ be an arbitrary poly type, and τ' an arbitrary mono type. We say that τ' is an *instance* of σ, written as $\sigma \preccurlyeq \tau'$, iff there exist mono types $\tau_1, \ldots, \tau_n$ such that:

$$\tau' \ = \ \tau[\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]$$

i.e., τ' is obtained from τ by substituting $\tau_1, \ldots, \tau_n$ for $\alpha_1, \ldots, \alpha_n$, respectively. ∎

Example 7 : Let $\sigma = \{\alpha_1, \alpha_3\}.\ \alpha_1$ `->` `int` `->` α_2. Then:

1. $\sigma \preccurlyeq \alpha_1$ `->` `int` `->` α_2.
2. $\sigma \preccurlyeq$ `int` `->` `int` `->` α_2.
3. $\sigma \preccurlyeq$ (`int` `->` `int`) `->` `int` `->` α_2.
4. $\sigma \preccurlyeq \alpha_2$ `->` `int` `->` α_2.
5. $\sigma \preccurlyeq (\alpha_2$ `->` $\alpha_3)$ `->` `int` `->` α_2. ∎

Exercise 8 : Give an example of a poly type σ for each of the following cases:

1. There exists exactly one $\tau \in \mathsf{mono}$ such that $\sigma \preccurlyeq \tau$.
2. For all $\tau \in \mathsf{mono}$ it holds that $\sigma \preccurlyeq \tau$.
3. For infinitely many $\tau \in \mathsf{mono}$ it holds that $\sigma \preccurlyeq \tau$,
   and for infinitely many $\tau \in \mathsf{mono}$ it does not hold that $\sigma \preccurlyeq \tau$. ∎

3 Typing Rules

The typing rules are used to combine the syntax of the (untyped) programming language and the syntax of types. Each typing rule consists of:

1. Finitely many (possibly zero) *premisses*.
2. Exactly one *conclusion*.

The conclusion of a typing rule, as well as some (if not all) of its premisses, are called *(typing) judgments*. A judgment is of the form:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \ \vdash \ M : \tau$$

for some $\sigma \in \mathsf{poly}$, some $\tau \in \mathsf{mono}$, and some expression M of **mini-ML**. The set $\{x_1, \ldots, x_n\}$ includes at least all the program variables occurring free in M. Each pair $x : \sigma$ on the left of "$\vdash$" is called a *type assumption*, and the order in which type assumptions are listed is not important, i.e., we think of the list on the left of "$\vdash$" as the set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. We use the letter A to denote such a set of type assumptions, which is called a *type environment*, and the only requirement we impose on A is that it does not mention a program variable more than once. (Put differently, we can view A as a partial function from tvar to poly.)

Every atom which is not a variable in the (untyped) programming language is assigned a closed type. In the case of **mini-ML**, the empty list [ ] is also an atom, and we have:

[ ]	: $\{\alpha\}.\ \alpha$ `list`	polymorphic empty list
i	: `int`	integers
b	: `bool`	booleans
`(op +)`	: `int * int -> int`	integer addition
`(op *)`	: `int * int -> int`	integer multiplication
`(op =)`	: $\{\alpha\}.\ \alpha$ `* `α `-> bool`	polymorphic equality
`hd`	: $\{\alpha\}.\ \alpha$ `list -> `α	polymorphic head
`tl`	: $\{\alpha\}.\ \alpha$ `list -> `α `list`	polymorphic tail
`(op ::)`	: $\{\alpha\}.\ \alpha$ `* `α `list -> `α `list`	polymorphic cons
⋮	⋮	

Remark 9 : In our presentation of the typing rules below, we assume that in every **mini-ML** expression, a variable has at most one binding occurrence. Thus, we do not allow an expression of the form — to take a silly example: `fn a => (fn a => a) a`. But this is a mild restriction, because we can always rename bound variable occurrences without changing the meaning of an expression. For the previous (silly) example, an equivalent expression is: `fn a => (fn b => b) a`. ∎

If $\diamond$ is a primitive operator, let $\mathsf{type}(\diamond)$ be its assigned type as shown above. We start with the typing rules for atoms that are not variables:

$$\text{NIL}\ \frac{\tau \in \mathsf{mono}}{A \vdash [\,] : \tau\ \texttt{list}} \qquad \text{INT}\ \frac{}{A \vdash i : \texttt{int}} \qquad \text{BOOL}\ \frac{}{A \vdash b : \texttt{bool}} \qquad \text{OP}\ \frac{\tau \in \mathsf{mono} \quad \mathsf{type}(\diamond) \preccurlyeq \tau}{A \vdash \diamond : \tau}$$

The rule for variables is:

$$\text{VAR}\ \frac{\sigma \in \mathsf{poly} \quad \tau \in \mathsf{mono} \quad \sigma \preccurlyeq \tau}{A, x : \sigma \ \vdash x : \tau} \qquad (A \text{ does not mention } x.)$$

The typing rules for non-atom expressions in **mini-ML** are:

$$\text{APP}\ \frac{A \vdash M : \tau_1 \texttt{ -> } \tau_2 \qquad A \vdash N : \tau_1}{A \vdash MN : \tau_2} \qquad \text{ABS}\ \frac{A, x : \tau_1 \vdash M : \tau_2}{A \vdash \texttt{fn}\, x \texttt{ => } M : \tau_1 \texttt{ -> } \tau_2}$$

$$\text{IF}\ \frac{A \vdash M : \texttt{bool} \quad A \vdash N : \tau \quad A \vdash P : \tau}{A \vdash \texttt{if}\, M \,\texttt{then}\, N \,\texttt{else}\, P : \tau} \qquad \text{LIST}\ \frac{A \vdash M_p : \tau \quad 1 \leqslant p \leqslant n}{A \vdash [M_1, \ldots, M_n] : \tau\ \texttt{list}}$$

$$\text{PAIR}\ \frac{A \vdash M_1 : \tau_1 \quad A \vdash M_2 : \tau_2}{A \vdash (M_1, M_2) : \tau_1 \texttt{ * } \tau_2} \qquad \text{PROJ\_1}\ \frac{A \vdash M : \tau_1 \texttt{ * } \tau_2}{A \vdash \texttt{\#1}\, M : \tau_1} \qquad \text{PROJ\_2}\ \frac{A \vdash M : \tau_1 \texttt{ * } \tau_2}{A \vdash \texttt{\#2}\, M : \tau_2}$$

The most complicated typing rules are for `let-val` and `let-fun` expressions:

$$\text{LET-VAL}\ \frac{\begin{array}{l} A,\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \vdash N : \tau \\ A\ \vdash M_p : \tau_p \qquad \sigma_p = (\mathsf{FTV}(\tau_p) - \mathsf{FTV}(A)).\ \tau_p \qquad 1 \leqslant p \leqslant n \end{array}}{A\ \vdash \texttt{let val}\ x_1 \texttt{ = } M_1\ \texttt{and}\ \cdots\ \texttt{and}\ x_n \texttt{ = } M_n\ \texttt{in}\ N\ \texttt{end} : \tau}$$

$$\text{LET-FUN}\ \frac{\begin{array}{l} A,\ x_1 : \tau_1 \texttt{ -> } \tau_1', \ldots, x_n : \tau_n \texttt{ -> } \tau_n'\ \vdash N : \tau \\ A,\ x_1 : \tau_1 \texttt{ -> } \tau_1', \ldots, x_n : \tau_n \texttt{ -> } \tau_n',\ y_1 : \tau_1, \ldots, y_n : \tau_n\ \vdash M_p : \tau_p' \qquad 1 \leqslant p \leqslant n \end{array}}{A \vdash \texttt{let fun}\ x_1(y_1) \texttt{ = } M_1\ \texttt{and}\ \cdots\ \texttt{and}\ x_n(y_n) \texttt{ = } M_n\ \texttt{in}\ N\ \texttt{end} : \tau}$$

Example 10 : Consider the following expression M of **mini-ML**: `fn f => f 5`. This is also a SML expression. The SML interpreter returns the following type for M:

```
(int -> 'a) -> 'a
```

Below is a typing derivation, which confirms M type-checks, by assigning the type `(int -> 'a) -> 'a` to M:

```
1. f: int -> 'a |- f : int -> 'a                    VAR
2. f: int -> 'a |- 5 : int                          INT
3. f: int -> 'a |- f 5: 'a                          APP, from 1 and 2
4. |- fn f => f 5: (int -> 'a) -> 'a                ABS, from 3
```

(We have written the typing derivation in ASCII to illustrate what we expect to be turned in for some of the exercises below.) Note, for each line of the derivation, we mention on the right-hand side:

- The name of the typing rule according to which the judgment, on the same line, is derived.
- The premises (if any) on preceding lines which are used by the typing rule. ■

Exercise 11 : Consider the following expression M of **mini-ML**: `fn f => fn x => f (f x)`. Write a typing derivation that confirms M type-checks and assigns the same type to M as the type obtained by running the SML interpreter. ■

Exercise 12 : Consider the following expression M of **mini-ML**: `fn f => fn x => f (f ((op +) (x, 3)))`. Write a typing derivation that confirms M type-checks and assigns the same type to M as the SML interpreter. ■

Exercise 13 : Consider the following expression M of **mini-ML**: `let val f = fn x => x in (f f) 3 end`. Write a typing derivation that confirms M type-checks and assigns the same type to M as the SML interpreter. ■

Exercise 14 : The typing rules presented above are for an *implicitly typed* version of **mini-ML**. It is implictly typed in the sense that, if an expression M of **mini-ML** type-checks according to the above rules, then no type annotations are inserted in M and only a final type is derived for the whole of M.

An expression of **mini-ML** is (fully) *explicitly typed* if an appropriate type annotation is inserted next to every binding occurrence of a variable. This is just as in SML. We want to adjust the typing rules so that they produce explicitly typed expressions. The only rules that need to be adjusted are ABS, LET-VAL and LET-FUN. The adjusted rule ABS is called ABS′ and reads as follows:

$$\text{ABS}' \quad \frac{A, x : \tau_1 \vdash M : \tau_2}{A \vdash \texttt{fn}\, x : \tau_1 \texttt{ => } M \texttt{ : } \tau_1 \texttt{ -> } \tau_2}$$

Similar adjustments need to be made in rules LET-VAL and LET-FUN, now called LET-VAL′ and LET-FUN′.

1. Write rule LET-VAL′.
2. Write rule LET-FUN′. ∎

Exercise 15 : The following is an expression of **mini-ML** — call it M:

```
let   val inc = fn x => (op +) (x,1)
      and double = fn f => fn y => f (f y)
in    (double inc 3, double not true)
end
```

The following is also an expression of **mini-ML** — call it N:

```
let   fun inc x = (op +) (x,1)
      and double f = fn y => f (f y)
in    (double inc 3, double not true)
end
```

M type-checks according to the typing rules of **mini-ML** presented earlier in this handout, but N does not according to the same rules.

1. Give a precise reason for the failure of N to type-check in **mini-ML**: State the typing rule that is violated and determine the part in that typing rule which does not allow a typing derivation to be completed for N.
2. Propose a relaxation of the typing rule, stated in part 1, that will allow N to be type-checked and to return the same value as M.
3. Write a typing derivation that confirms N type-checks according to the adjusted typing rules, as proposed in part 2. ∎

Exercise 16 : In Remark **??** we imposed a restriction on **mini-ML** expressions, by requiring that every variable has at most one binding occurrence in an expression.

1. What goes wrong with the typing rules, if this restriction is lifted? *Hint*: Consider a "typing derivation" for the expression `fn a => (fn a => a) a`, and compare it with a typing derivation for `fn a => (fn b => b) a`.
2. Suggest a change in the definition of the set of type assumptions in a judgment, i.e., the set appearing on the left of "⊢", so that this restriction can be lifted safely. *Hint*: Test your proposed change on a "typing derivation" for `fn a => (fn a => a) a`, ∎