Examples of Typing Derivations for **mini-ML** Expressions

Assaf Kfoury

November 8, 2005

This document is best read on a monitor screen, using a previewer (ghostscript for ps, acroread for pdf). If you print it out, which you can of course, it will not give you the effect of the overlayer pages.

Example 1: `fn f => (op +) (f 5, 10)`

This expression is well-formed in **mini-ML** and, therefore, in SML too. Running the SML interpreter confirms that the expression is well-formed and returns the following type for it: `(int -> int) -> int`.

Goal: Verify the expression type-checks according to the typing rules of **mini-ML**.

Example 1: `fn f => (op +) (f 5, 10)`

This expression is well-formed in **mini-ML** and, therefore, in SML too. Running the SML interpreter confirms that the expression is well-formed and returns the following type for it: `(int -> int) -> int`.

Goal: Verify the expression type-checks according to the typing rules of **mini-ML**.

1.	`f:int -> int` $\vdash$ `(op +) : int * int -> int`	OP
2.	`f:int -> int` $\vdash$ `f : int -> int`	VAR
3.	`f:int -> int` $\vdash$ `5 : int`	INT
4.	`f:int -> int` $\vdash$ `f 5 : int`	APP from 2, 3
5.	`f:int -> int` $\vdash$ `10 : int`	INT
6.	`f:int -> int` $\vdash$ `(f 5,10) : int * int`	PAIR from 4, 5
7.	`f:int -> int` $\vdash$ `(op +)(f 5,10) : int`	APP from 1, 6
8.	$\vdash$ `fn f => (op +)(f 5,10) : (int -> int) -> int`	ABS from 7

Example 2: `let val f = fn x => x in (f f) 5 end`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression, with `int` as the final type assigned to it.

Goal: Verify the expression type-checks according to the typing rules of **mini-ML**.
First, we show the skeleton of the typing derivation we want, i.e., the derivation without any types.

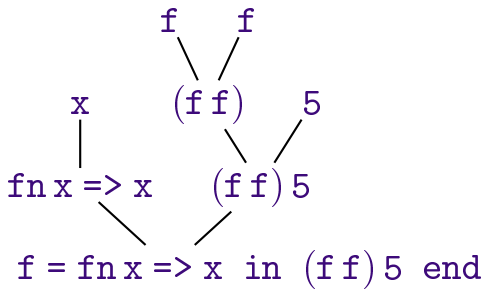
1.	$\vdash$ `f`	VAR
2.	$\vdash$ `f`	VAR
3.	$\vdash$ `f f`	APP from 1, 2
4.	$\vdash$ `5`	INT
5.	$\vdash$ `(f f) 5`	APP from 3, 4
6.	$\vdash$ `x`	VAR
7.	$\vdash$ `fn x => x`	ABS from 6
8.	$\vdash$ `let val f = fn x => x in (f f) 5 end`	LET-VAL from 5, 7

Example 2: `let val f = fn x => x in (f f) 5 end`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression, with `int` as the final type assigned to it.

Goal: Verify the expression type-checks according to the typing rules of **mini-ML**.
First, we show the skeleton of the typing derivation we want, i.e., the derivation without any types. The skeleton is produced using the parse tree of the expression, shown below.

1.	$\vdash$ `f`	VAR
2.	$\vdash$ `f`	VAR
3.	$\vdash$ `f f`	APP from 1, 2
4.	$\vdash$ `5`	INT
5.	$\vdash$ `(f f) 5`	APP from 3, 4
6.	$\vdash$ `x`	VAR
7.	$\vdash$ `fn x => x`	ABS from 6
8.	$\vdash$ `let val f = fn x => x in (f f) 5 end`	LET-VAL from 5, 7

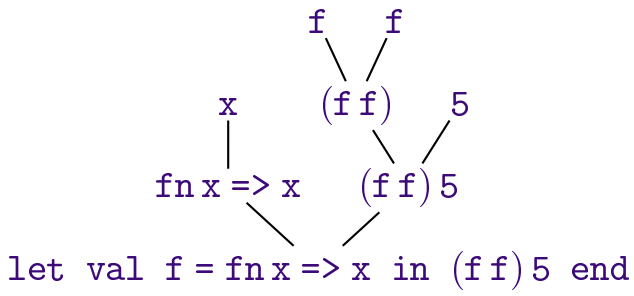


Example 2: `let val f = fn x => x in (f f) 5 end`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter on the expression, with `int` as the final type assigned to it.

Goal: Verify the expression type-checks according to the typing rules of **mini-ML**.
Second, we insert appropriate types into the skeleton, thus producing a completed typing derivation.

1.	`f:{'a}.'a -> 'a`	⊢ `f :(int -> int) -> (int -> int)`	VAR
2.	`f:{'a}.'a -> 'a`	⊢ `f :int -> int`	VAR
3.	`f:{'a}.'a -> 'a`	⊢ `f f :int -> int`	APP from 1, 2
4.	`f:{'a}.'a -> 'a`	⊢ `5 :int`	INT
5.	`f:{'a}.'a -> 'a`	⊢ `(f f) 5 :int`	APP from 3, 4
6.	`x:'b`	⊢ `x :'b`	VAR
7.		⊢ `fn x => x :'b ->'b`	ABS from 6
8.		⊢ `let val f = fn x => x in (f f) 5 end :int`	LET-VAL from 5, 7



Example 3: `let fun f(x) = x in (f f) 5 end`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter, with `int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules of **mini-ML**?
If it does, we should be able to insert an appropriate type environment to the left of "$\vdash$" and an appropriate type to the right of the expression --- on each line of the skeleton (typing derivation without types) shown below.

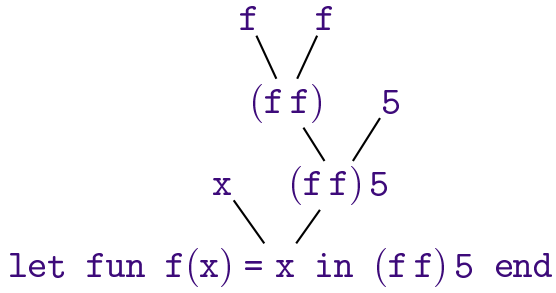
1.	$\vdash$ `f`	VAR
2.	$\vdash$ `f`	VAR
3.	$\vdash$ `f f`	APP from 1, 2
4.	$\vdash$ `5`	INT
5.	$\vdash$ `(f f) 5`	APP from 3, 4
6.	$\vdash$ `x`	VAR
7.	$\vdash$ `let fun f(x) = x in (f f) 5 end`	LET-FUN from 5, 6

Example 3: `let fun f(x) = x in (f f) 5 end`

This expression is well-formed in **mini-ML** and, therefore, in SML. This is confirmed by running the SML interpreter, with `int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules of **mini-ML**?
If it does, we should be able to insert an appropriate type environment to the left of "$\vdash$" and an appropriate type to the right of the expression --- on each line of the skeleton (typing derivation without types) shown below.

1.	$\vdash$ `f`	VAR
2.	$\vdash$ `f`	VAR
3.	$\vdash$ `f f`	APP from 1, 2
4.	$\vdash$ `5`	INT
5.	$\vdash$ `(f f) 5`	APP from 3, 4
6.	$\vdash$ `x`	VAR
7.	$\vdash$ `let fun f(x) = x in (f f) 5 end`	LET-FUN from 5, 6



Answer: No, we cannot type-check the expression with the typing rules of **mini-ML** -- not shown here -- although we can with the rules of SML (not presented in CS 320).

Example 4: The following expression is well-formed in the **mini-ML** syntax and, thus, in the SML syntax too:

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

Running the SML interpreter on the expression returns `int * bool` as a final type.

Goal: Verify the expression type-checks according to the typing rules of **mini-ML**.

We first show the skeleton of an appropriate typing derivation, then we show the same skeleton with types inserted into it.

1.	⊢ `x`	VAR
2.	⊢ `1`	INT
3.	⊢ `(x,1)`	PAIR from 1, 2
4.	⊢ `(op +)`	OP
5.	⊢ `(op +)(x,1)`	APP from 3, 4
6.	⊢ `fn x => (op +)(x,1)`	ABS from 5
7.	⊢ `y`	VAR
8.	⊢ `f`	VAR
9.	⊢ `f y`	APP from 7,8
10.	⊢ `f`	VAR
11.	⊢ `f (f y)`	APP from 9,10
12.	⊢ `fn y => f (f y)`	ABS from 11
13.	⊢ `fn f => fn y => f (f y)`	ABS from 12
14.	⊢ `dbl`	VAR
15.	⊢ `inc`	VAR
16.	⊢ `dbl inc`	APP from 14, 15
17.	⊢ `3`	INT
18.	⊢ `dbl inc 3`	APP from 16,17
19.	⊢ `dbl`	VAR
20.	⊢ `not`	OP
21.	⊢ `dbl not`	APP from 19, 20
22.	⊢ `true`	BOOL
23.	⊢ `dbl not true`	APP from 21,22
24.	⊢ `(dbl inc 3, dbl not true)`	PAIR from 18,23

25.	⊢ `let val inc = fn x => (op +) (x,1)`
	`and dbl = fn f => fn y => f (f y)`
	`in (dbl inc 3, dbl not true) end`
		LET-VAL
		from 6, 13, ,24

1.	`x : int`	$\vdash$ `x : int`	VAR
2.	`x : int`	$\vdash$ `1 : int`	INT
3.	`x : int`	$\vdash$ `(x,1) : int * int`	PAIR from 1, 2
4.	`x : int`	$\vdash$ `(op +) : int * int -> int`	OP
5.	`x : int`	$\vdash$ `(op +)(x,1) : int`	APP from 3, 4
6.		$\vdash$ `fn x => (op +)(x,1) : int -> int`	ABS from 5
7.	`f : 'a -> 'a, y : 'a`	$\vdash$ `y : 'a`	VAR
8.	`f : 'a -> 'a, y : 'a`	$\vdash$ `f : 'a -> 'a`	VAR
9.	`f : 'a -> 'a, y : 'a`	$\vdash$ `f y : 'a`	APP from 7,8
10.	`f : 'a -> 'a, y : 'a`	$\vdash$ `f : 'a -> 'a`	VAR
11.	`f : 'a -> 'a, y : 'a`	$\vdash$ `f (f y) : 'a`	APP from 9,10
12.	`f : 'a -> 'a`	$\vdash$ `fn y => f (f y) : 'a -> 'a`	ABS from 11
13.		$\vdash$ `fn f => fn y => f (f y) : ('a -> 'a) -> ('a -> 'a)`	ABS from 12

14.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ dbl : (int -> int) -> (int -> int)	VAR
15.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ inc : int -> int	VAR
16.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ dbl inc : int -> int	APP from 14, 15
17.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ 3 : int	INT
18.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ dbl inc 3 : int	APP from 16,17
19.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ dbl : (bool -> bool) -> (bool -> bool)	VAR
20.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ not : bool -> bool	OP
21.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ dbl not : bool -> bool	APP from 19, 20
22.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ true : bool	BOOL
23.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ dbl not true : bool	APP from 21,22
24.	dbl : {'b}. ('b -> 'b) -> ('b -> 'b), inc : int -> int	⊢ (dbl inc 3, dbl not true) : int * bool	PAIR from 18,23
25.		⊢ let val inc = fn x => (op +) (x,1) and dbl = fn f => fn y => f (f y) in (dbl inc 3, dbl not true) end : int * bool	LET-VAL from 6, 13, ,24