Examples of Typing Derivations

in a “Mini-Haskell” where the only available types are `Int` and `Bool`

Assaf Kfoury

1 November 2016

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: \ `f -> (+)` (`f 5`) 10

Running a Haskell interpreter confirms that the expression is well-formed and that the following type is valid for it (there are other valid types for it in full Haskell): (`Int -> Int`) `-> Int`.

Example 1: \ `f -> (f 5) + 10`

Running a Haskell interpreter confirms that the expression is well-formed and that the following type is valid for it (there are other valid types for it in full Haskell): `(Int -> Int) -> Int`.

1.	`f : Int -> Int`	⊢ `f : Int -> Int`	VAR
2.	`f : Int -> Int`	⊢ `5 : Int`	INT
3.	`f : Int -> Int`	⊢ `f 5 : Int`	APP from 1, 2
4.	`f : Int -> Int`	⊢ `10 : Int`	INT
5.	`f : Int -> Int`	⊢ `(f 5) + 10 : Int`	ADD from 3, 4
6.		⊢ \ `f -> (f 5) + 10 : (Int -> Int) -> Int`	ABS from 5

Example 2: `let f = (\ x -> x) in f (f 5)`

This expression is well-formed. This is confirmed by running a Haskell interpreter on the expression, with `Int` as the final type assigned to it.

First, we show the skeleton of the typing derivation we want, i.e., the derivation without any types.

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

Example 2: `let f = (\ x -> x) in f (f 5)`

This expression is well-formed. This is confirmed by running a Haskell interpreter on the expression, with `Int` as the final type assigned to it.

Second, we insert appropriate types into the skeleton, thus producing a completed typing derivation.

```
1.  f : Int-> Int ⊢ f : Int-> Int                                VAR
2.  f : Int-> Int ⊢ 5 : Int                                      INT
3.  f : Int-> Int ⊢ f 5 : Int                                    APP from 1, 2
4.  f : Int-> Int ⊢ f : Int-> Int                                VAR
5.  f : Int-> Int ⊢ f (f 5) : Int                                APP from 3, 4
6.  x : Int       ⊢ x : Int                                      VAR
7.                ⊢ \ x -> x : Int-> Int                         ABS from 6
8.                ⊢ let f = (\ x -> x) in f (f 5) : Int          LET from 5, 7
```

Example 3: `let f = (\ x -> x) in (f f) 5`

This expression is well-formed. This is confirmed by running a Haskell interpreter, with `Int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules?
If it does, we should be able to insert an appropriate type environment to the left of "⊢" and an appropriate type to the right of the expression — on each line of the skeleton below.

1.	⊢ `f`	VAR
2.	⊢ `f`	VAR
3.	⊢ `f f`	APP from 1, 2
4.	⊢ `5`	INT
5.	⊢ `(f f) 5`	APP from 3, 4
6.	⊢ `x`	VAR
7.	⊢ `\ x -> x`	ABS from 6
8.	⊢ `let f = (\ x -> x) in (f f) 5`	LET from 5, 7

Example 3: `let f = (\ x -> x) in (f f) 5`

This expression is well-formed. This is confirmed by running a Haskell interpreter, with `Int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules?
If it does, we should be able to insert an appropriate type environment to the left of "⊢" and an appropriate type to the right of the expression — on each line of the skeleton below.

1. ⊢ `f` VAR
2. ⊢ `f` VAR
3. ⊢ `f f` APP from 1, 2
4. ⊢ `5` INT
5. ⊢ `(f f) 5` APP from 3, 4
6. ⊢ `x` VAR
7. ⊢ `\ x -> x` ABS from 6
8. ⊢ `let f = (\ x -> x) in (f f) 5` LET from 5, 7

Answer: **NO**, we cannot type-check the expression with the monomorphic typing rules, although we can with the polymorphic typing rules – on the next page.

Example 3: `let f = (\ x -> x) in (f f) 5`

This expression is well-formed. This is confirmed by running a Haskell interpreter, with `Int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules?
If it does, we should be able to insert an appropriate type environment to the left of "⊢" and an appropriate type to the right of the expression — on each line of the skeleton below.

```
1. f : forall a.a -> a ⊢ f : (Int -> Int) -> (Int -> Int)   POLYVAR
2. f : forall a.a -> a ⊢ f : Int -> Int                     POLYVAR
3. f : forall a.a -> a ⊢ f f : Int -> Int                   APP from 1, 2
4. f : forall a.a -> a ⊢ 5 : Int                            INT
5. f : forall a.a -> a ⊢ (f f) 5 : Int                      APP from 3, 4
6. x : b               ⊢ x : b                              VAR
7.                     ⊢ \ x -> x : b -> b                  ABS from 6
8.                     ⊢ let f = (\ x -> x) in (f f) 5 : Int   POLYLET from 5, 7
```

<u>Example 4</u>: `let ones = (1 :   ones) in (head  ones)`

This expression is well-formed. This is confirmed by running a Haskell interpreter, with `Int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules?
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 below. In this example, we cannot hope to type-check the full expression with the rule LET, we must use LETREC instead.

1.	$\vdash$ `ones`	VAR
2.	$\vdash$ `(head ones)`	HEAD from 1
3.	$\vdash$ `1`	INT
4.	$\vdash$ `ones`	VAR
5.	$\vdash$ `(1 : ones)`	CONS from 3, 4
6.	$\vdash$ `let ones = (1 : ones) in (head  ones)`	LETREC from 2, 5

<u>Example 4</u>: `let ones = (1 :   ones) in (head  ones)`

This expression is well-formed. This is confirmed by running a Haskell interpreter, with `Int` as the final type assigned to the expression.

Goal: Does this expression type-check according to the typing rules?
If it does, we should be able to insert an appropriate type environment to the left of "⊢" and an appropriate type to the right of the expression — on each line of the skeleton below. In this example, we cannot hope to type-check the full expression with the rule LET, we must use LETREC instead.

1.	`ones : [Int]`	⊢ `ones : [Int]`	VAR
2.	`ones : [Int]`	⊢ `(head ones) : Int`	HEAD from 1
3.	`ones : [Int]`	⊢ `1 : Int`	INT
4.	`ones : [Int]`	⊢ `ones : [Int]`	VAR
5.	`ones : [Int]`	⊢ `(1 : ones) : [Int]`	CONS from 3, 4
6.		⊢ `let ones = (1 : ones) in (head  ones) : Int`	LETREC from 2, 5