Simply Typed Lambda Calculus

Deadline: Thursday, March 28th 2024, 5pm

- this assignment has 75 points (they'll scale to a percentage score like all other homeworks).
- Work must be uploaded to gradescope per-question; text boxes and file uploads are ok.
    - uploading pdfs/images are acceptable; other filetypes (such as docx) are not, and will have up to 5pt deduction assessed. It slows down grading significantly. Similarly, rotate images to be correctly upright. Thank you! ☺

We will write proofs and code by hand, this time about the simply typed lambda calculus (with a few extensions given: booleans, numbers, lists, and recursion).

The entire rules and definition for our calculus are available here:
https://cs.gmu.edu/~marks/463/calculi/stlc_rules.pdf

You can use simple ASCII shorthands to keep your writings more compact. No bonus points for presentation style, so just make your life easy! Perhaps make it pretty later on.

- Z in place of $\mathbb{Z}$; B in place of $\mathbb{B}$; [x] or TyList x in place of $[\![x]\!]$.
- -> in place of $\rightarrow$; |- in place of $\vdash$
- *Here are just a bunch of cut-pastable symbols:* $\rightarrow \vdash \mapsto \Gamma \lambda \mathbb{Z} \mathbb{B} [\![ ]\!]$

---

1. Proof Trees (25pts)

For each of the following, write out an entire proof tree, including naming the rules like our classroom examples, to show that the term is well-typed (and what its type is).

1. `2*(3+4)`
2. `if (0>1) 2 3`
3. `(((λx:ℤ.λy:ℤ.x+y) 8) 9)`
4. `((λx:ℤ.x+1) 5)`
5. `(head ( nil [[ℙ]]))` *hint: don't think about evaluation!*

---

2. Type Errors (10pts)

For each of the following ill-typed terms, write out a proof tree (including type rules used) as far as possible until the error prohibits further proving; then point out that part and explain why the type rule cannot apply. E.g., "this use of Ty-App expects a function as the left subterm, but we only have ℤ." Your proof tree must have the evidence in it that you describe in your description.

1. `cons true (nil [[ℤ]])`
2. `((λx:ℤ.x>2) true)`

---

3. Language Extensions (20pts)

For each of the following, you will consider the given language extension and write out one type rule per term. The full t/v/T definition is given to help you understand the exact expected implementation/type usage. When in doubt, ask for clarification; these are just like the definitions presented in class though.

3.1 Maybe: there are two kinds of maybe-values; maybe we have the value, maybe we don't. When there is a value, we can extract the value from the maybe-value.

```
t ::= … | just t | nothing T | isjust t | unjust t
T ::= … | Maybe T
v ::= … | just t | nothing T
```

This is somewhat like the following Haskell definition of Maybe. (except we have to record the optional value's type in our `nothing T` terms such as `nothing 𝔹`, since we aren't doing type inference).

```
data Maybe a = Just a | Nothing

isJust :: Maybe a ⟶ Bool
isJust (Just v) = True
isJust Nothing  = False

unJust :: Maybe a ⟶ a
unJust (Just v) = v
```

3.2 Either: there are two kinds of either-values; either a **`left`**-labeled value of some type **`a`**, or a **`right`**-labeled value of some type **`b`**. We always need to remember what two types we've possibly got here, even though exactly one is present, so there's a type in place of the "missing" side. (This is a "tagged union"). We can check which kind we've got (**`isleft`**/**`isright`**), and we can try to extract the side we think we've got. (**`getleft`**/**`getright`**).

3.3 *NOTE: left and right record the "other" side's type; two example values of the same type:*

```
left  3 𝔹    :: Either ℤ 𝔹
right ℤ true :: Either ℤ 𝔹
```

```
t ::= … | left t T | right T t | isleft t | isright t | getleft t | getright t
T ::= … | Either T T
v ::= … | left t T | right T t
```

This is somewhat like the following Haskell definition of Either:

```
data Either a b = Left a | Right b

isLeft :: Either a b ⟶ Bool
isLeft (Left _) = True
isLeft _        = False

isRight :: Either a b ⟶ Bool
isRight (Right _) = True
isRight _         = False

getLeft :: Either a b ⟶ a
getLeft (Left v) = v

getRight :: Either a b ⟶ b
getRight (Right v) = v
```

4. Encodings (20pts)

We will encode various definitions in the simply typed lambda calculus, with all our extensions present (including the two from the previous question). Some require recursion (review the fix extension and its typing/evaluation rules). Each of these may likely be just a short expression to a couple of lines if you space things out; you can use all the extensions in our language to write out your encoding.

- See class slides for guidance on these definitions.
- *Hint: you can still use helper functions to solve tasks.*
- *Hint: if you're inspecting a (recursively defined) list, you'll almost certainly need recursion (fix)*

`Non-recursive Definitions`

1. `nand` given two Booleans, answer the not-and of those values.
   (remember, we have if-expressions! Focus on the types.)
2. `thrice` given a function `ℤ→ℤ`, and an int, call the function three times composed on the int.
   example: `(thrice (λx:ℤ.x+1) 10) == 13`
3. `cubed` given an int, cube its value (raise it to the third power).
4. `modify` given an `Either ℤ 𝔹`, this function either increments the int, or negates the Boolean, and then puts it back into an either. Examples:
   - `modify (left 3 𝔹 ) ===> (left 4 𝔹 )`
   - `modify (right ℤ true) ===> (right ℤ false)`
5. `safeAdd` given two `Maybe ℤ` values, add their ints; any `nothing` should be treated like a `just 0`.
   Hint: a helper function may be a good idea!

`Recursive Definitions`

6. `all` accepts a list of Booleans, and answers if all the items in the list are `true`.
   (Same idea: "there are no `false` values present in this list.")
7. `sumlist` accepts a list of ints, and returns the sum. When the list is empty, `0` is returned.
8. `takeN` accepts an int n, a list of ints xs, and returns the list that has the first n items of xs.
   Output may be shorter than n if there weren't enough items in the original input list.
9. `filter` accepts a function of type `ℤ→𝔹`, a list of ints, and returns the list of elements for which the function returned true
10. `zipwith` accepts a function of type `(ℤ→ℤ→ℤ)`, then two int lists, and applies the function across each position in the lists collecting answers, the same as Haskell's `zipWith` on ints.

---

5. Extra Credit (+5)

Encode the `maxlist` function, which accepts a list of ints and returns `Maybe ℤ`. An empty list returns `nothing`, and a non-empty list returns `just` the largest int that was present.

- *Hint: think about how you would write this in Haskell first.*