Lab 02

15-150 Fall 2024

Due: Thu 5th Sept, 2024 at 5:00pm

Interview

This assignment will be evaluated in an interview with the course staff on the date above. The points in the tasks are meaningless and submission of solutions is not required.

The instructions for the interview are in the interview guide.

1 Inductive Trees

The set of trees $\mathbb{T}_\mathbb{Z}$ over the domain of the integers $\mathbb{Z}$ is inductively defined as:

$$\begin{cases} \mathit{empty} \in \mathbb{T}_\mathbb{Z} \\ \mathit{node} : \mathbb{T}_\mathbb{Z} \times \mathbb{Z} \times \mathbb{T}_\mathbb{Z} \to \mathbb{T}_\mathbb{Z} \end{cases}$$

And here is the inductive definition of the set of lists $\mathbb{L}_\mathbb{Z}$ over the domain of the integers $\mathbb{Z}$:

$$\begin{cases} \mathit{nil} \in \mathbb{L}_\mathbb{Z} \\ :: : \mathbb{Z} \times \mathbb{L}_\mathbb{Z} \to \mathbb{L}_\mathbb{Z} \end{cases}$$

In other words, a tree is either *empty* or $\mathit{node}(T_1, x, T_2)$, where T_1 and T_2 are trees and $x \in \mathbb{Z}$.

From this definition, we consider two functions on trees. First, the function $\mathit{inorder} : \mathbb{T}_\mathbb{Z} \to \mathbb{L}_\mathbb{Z}$ traverses its argument and collects the data in its node into a list, that is returned. It is defined as follows:

$$\begin{cases} \mathit{inorder}(\mathit{empty}) & = \mathit{nil} \\ \mathit{inorder}(\mathit{node}(T_L, x, T_R)) & = \mathit{append}(\mathit{inorder}(T_L), x :: \mathit{inorder}(T_R)) \end{cases}$$

inorder provides us with the means to reason about the elements of a tree using the structure of a list. The second function we will be using is the familiar $\mathit{size} : \mathbb{T}_\mathbb{Z} \to \mathbb{N}$ that determines the number of *node*s in a tree:

$$\begin{cases} \mathit{size}(\mathit{empty}) & = 0 \\ \mathit{size}(\mathit{node}(T_L, x, T_R)) & = 1 + \mathit{size}(T_L) + \mathit{size}(T_R) \end{cases}$$

Coding Task 1.1 (6 points) The type `tree` is defined for you in the code file for this lab. In this file, complete the code for the functions `inorder` and `size`.

Task 1.2 (10 points) By now, we have defined functions on both lists and trees that determine the size of the structure: that's *length* on lists and *size* on trees. This allows us to show that *inorder* preserves the size of its input.

Prove the following property by structural induction:

$$\text{For all } T \in \mathbb{T}_\mathbb{Z}, \qquad \mathit{size}(T) = \mathit{length}(\mathit{inorder}(T))$$

You may use the following lemma:

Lemma 1

For all $L_1, L_2 \in \mathbb{L}_\mathbb{Z}$, $\mathit{length}(\mathit{append}(L_1, L_2)) = \mathit{length}(L_1) + \mathit{length}(L_2)$.

2 Compiler Optimizations

In this problem, we will be working with the following datatype, which represents an arithmetic expression over integers.

```
datatype exp = Var of string
             | Int of int
             | Add of exp * exp
             | Mul of exp * exp
             | Not of exp
             | IfThenElse of exp * exp * exp
```

Here is the meaning of each constructor in the datatype:

Constructor	Meaning
`Var x`	A variable with name `x`. When evaluating the expression, this variable must be assigned a value in the environment.
`Int n`	A constant integer with value `n`.
`Add (a,b)`	The sum of two expressions, `a` and `b`.
`Mul (a,b)`	The product of two expressions, `a` and `b`.
`Not a`	The "truthy" negation of an integer expression. If `a` is `0`, `Not a` should be `1`. Otherwise, `Not a` should be `0`.
`IfThenElse (i,t,e)`	"Truthy" casing on integers. If `i` is nonzero, the expression evaluates to `t`. Otherwise, it evaluates to `e`.

Here are some example representations of operations.

Arithmetic	`exp`
$2+3$	`Add (Int 2,Int 3)`
$2+x$	`Add (Int 2,Var "x")`
$3x+1$	`Add (Mul (Int 3,Var "x"),Int 1)`
$3 \cdot 2+x$	`Add (Mul (Int 3,Int 2),Var "x")`
"if $x \neq 0$ then y else 8"	`IfThenElse (Var "x",Var "y",Int 8)`
"1 if $x=0$, else 0"	`Not (Var "x")`
"0 if $x=0$, else 1"	`Not (Not (Var "x"))`

We also need a way to represent the "environment", which contains the variable bindings. We will use the `environ` type alias:

```
type environ = (string * int) list
```

Note that there are two helper functions defined in `exp.sml` that you might find useful as you work through the following tasks.

- `lookup (env,x)` finds the integer bound to variable `x` in environment `env`.
- `vars e` evaluates to a list of all variable names in expression `e`.

Full specifications and implementations can be found in the file itself.

2.1 Evaluation

Before we do anything, it would be helpful to define an evaluation function for our datatype. Our function will take in an environment, which is just a list of `string * int` pairs representing which integer each variable name is bound to. You may assume the list has no duplicates.

Coding Task 2.1 (5 points) In `exp.sml`, define the function

```
environ * exp -> int
```

such that `eval (env,e)` returns an integer `n`, where `n` is the integer representing the value of `e` given environment `env`. You may assume that each variable in `e` has one entry in `env`.

For example, it should be the case that:

```
eval ([("x",45),("y",3)],Add (Int 15,Mul (Var "x",Var "y"))) ↦ 150
```

If you attempt to view deeply-nested `exp` values, the REPL will stop printing a few layers in, instead showing a `#` sign. You can (temporarily) increase the print depth by typing the following into the REPL:

```
Control.Print.printDepth := 100
```

or by running in the terminal:

```
PRINT_DEPTH=100 ./smlnj exp.sml
```

2.2 Constant Fusion

Many programming language implementations have optimizations so that the code you write will run faster. For example, if you write the Python program

```
def foo(x):
    return x + (2 * 3)
```

an optimization called *constant fusion* may be applied, transforming the function to

```
def foo(x):
    return x + 6
```

In this case, the constant-fused expression has precomputed `(2 * 3)`, so we can avoid the multiplication step at runtime when the function is called. We have implemented a similar optimization[1] in `fuse.sml` that fuses multiplications for our integer expression language:

```
fun fuse (Var x) = Var x
  | fuse (Int n) = Int n
  | fuse (Add (a,b)) = Add (fuse a, fuse b)
  | fuse (Mul (a,b)) = (
      case (vars a,vars b) of
        ([],[]) => Int (eval ([],a) * eval ([],b))
      | _       => Mul (fuse a,fuse b)
    )
  | fuse (Not a) = Not (fuse a)
  | fuse (IfThenElse (i,t,e)) = IfThenElse (fuse i,fuse t,fuse e)

val Add (Var "x",Int 6) = fuse (Add (Var "x", Mul (Int 2,Int 3)))
```

[1] This is just one of many possible optimizations! For example, you could consider a similar optimization for addition, or optimizing expressions like `Add (Add (x,x),x)` to `Mul (Int 3,x)`. We will only consider the given optimization for constant multiplication on multiplication in this problem, but feel free to experiment with coding (and proving) other optimizations.

However, before enabling this optimization, we want to ensure that the constant-fused code is actually equal[2] to the original code. Thankfully, we can prove this fact and sleep soundly for the rest of our lives.

The next four tasks will refer to the validity of `fuse`.

Theorem 1 (Validity of `fuse`). *For all values `env` and `e` satisfying the REQUIRES for `eval`,*

$$\texttt{eval (env, fuse e)} = \texttt{eval (env, e)}$$

You may make use of the following lemmas:

Lemma 1 (Totality of `vars`). *The `vars` function is total.*

Lemma 2 (Totality of Fusion). *The `fuse` function is total.*

Lemma 3 (Idempotence of Fusion). *For all `e : exp`:*

$$\texttt{fuse e} = \texttt{fuse (fuse e)}$$

Lemma 4 (Closed Expression Valuability). *For all `e : exp`, if `vars e = []`, then for all `env : environ`, we have that `eval (env, e)` is valuable.*

Lemma 5 (Environment Independence of Closed Expressions). *For all `e : exp`, if `vars e = []`, then for all `env : environ`, we have that `eval (env, e) = eval ([], e)`.*[3]

Feel free to abbreviate `IfThenElse` as `ITE`.

Note that this proof may depend on your implementation of `eval`. Make sure you are convinced of its correctness!

Task 2.2 (1 points) Identify which cases are base cases for the structural induction proof of the theorem.

Task 2.3 (4 points) State the inductive cases and their IHs for the structural induction proof of the theorem.

Task 2.4 (5 points) Prove the `IfThenElse` case for the structural induction proof of the theorem.

Task 2.5 (11 points) Prove the `Mul` case for the structural induction proof of the theorem.

Note that during the lab interview, while you may use abbreviations to shorten your proof, make sure this is *not* at the expense of correctness clarity. In particular, please make the structure of your proof is clear: be sure to *explicitly* state all cases and inductive hypotheses.

[2] You should assume that the two programs are equal when they evaluate to the same value. As you will later see in the course, this is the concept of program equivalence.

[3] Note that this is a special case of a stronger lemma. Namely, given `e : exp`, if `env` and `env'` bind all variables in `e` to the same integers and satisfy the REQUIRES of `eval`, then `eval (env, e) = eval (env', e)`.