## Contents

1 Datatypes Lab .................................................. 2
   1.1 Totality and Valuability .................................. 2
       1.1.1 A Totally Awesome Proof .......................... 2
   1.2 Inductively-Defined Datatypes ............................ 3
       1.2.1 Some Built-Ins ................................. 3
       1.2.2 Infinite Power ................................... 3
   1.3 Structural Induction ...................................... 3

2 Having fun is valuable! ........................................ 4
   Task 1. ....................................................... 4
   Task 2. ....................................................... 4
   Task 3. ....................................................... 4
   Task 4. ....................................................... 4
   Task 5. ....................................................... 4
   Task 6. ....................................................... 4
   Task 7. ....................................................... 4
   Task 8. ....................................................... 4

3 Listen Up ....................................................... 5
   Task 9. ....................................................... 5
   Task 10. ...................................................... 5
   Task 11. ...................................................... 5
   Task 12. ...................................................... 6
   Task 13. ...................................................... 6
   Task 14. ...................................................... 6
   Task 15. ...................................................... 6

4 “Trees Are Never Sad Look At Them Every Once In Awhile They’re Quite Beautiful”
   —Jaden Smith (@jaden), 9/19/2013 ............................ 8
   Task 16. ...................................................... 8
   Task 17. ...................................................... 8

5 Proving Totality ................................................. 10
   Task 18. ...................................................... 10
1 Datatypes Lab

1.1 Totality and Valuability

Definition (Valuable). A well-typed expression $e$ is *valuable* if there exists a value $v$ such that $e \Rightarrow v$. We also say that $e$ *evaluates to a value*.

Definition (Total). A well-typed expression $e_1 : t_1 \rightarrow t_2$, for types $t_1$ and $t_2$, is *total* if for all values $e_2 : t_1$, the expression $e_1 \ e_2$ is valuable.

1.1.1 A Totally Awesome Proof

```haskell
fun foo x = foo x + 1
```

We will now prove that $0 \cong 1$.

*Proof*. Obviously, by math, we have $(\text{foo } x - \text{foo } x) \cong 0$, for any $x$.

Additionally, we derive the following:

$\text{foo } x - \text{foo } x \cong (\text{foo } x + 1) - \text{foo } x$  
\hspace{1cm} \text{(definition of foo)}

$\cong 1 + (\text{foo } x - \text{foo } x)$  
\hspace{1cm} \text{(math)}

$\cong 1 + 0$  
\hspace{1cm} \text{(math)}

$\cong 1$  
\hspace{1cm} \text{(math)}

Therefore, since $0 \cong (\text{foo } x - \text{foo } x) \cong 1$, we have $0 \cong 1$. \qed
1.2 Inductively-Defined Datatypes

Datatypes can be inductively defined, each declaring one or more constructor.

1.2.1 Some Built-Ins

There are lots of types which are built into SML. For example:

- The int list type:
- The unit type:
- The int option type:

1.2.2 Infinite Power

We can define our own datatypes using the datatype keyword:

1.3 Structural Induction

Structural induction inducts on the structure of a datatype.

Let’s prove that two summing functions are equivalent.

fun treeSum (Empty : tree) : int = 0
| treeSum (Node (L,x,R)) = treeSum L + x + treeSum R

fun inorder (Empty : tree) : int list = []
| inorder (Node (L,x,R)) = inorder L @ (x :: inorder R)

fun listSum ([]) : int list = 0
| listSum (x :: xs) = x + listSum xs

Lemma 1. For values xs : int list, ys : int list, we have listSum xs + listSum ys ∼= listSum (xs @ ys).

Lemma 2. inorder is total, i.e., for all values T : tree we have inorder T reduces to a value.

We want to prove that for all trees T : tree, treeSum T ∼= listSum (inorder T).
2 Having **fun** is valuable!

In this section we will explore totality and valuability in more detail.

Let $t_1, t_2, t_3$ be non-function types (that is, they can be, for instance, `int`, `string`, `int list`, or `int list list`, but not `int -> int` or `int list -> int`).

State whether each of the following statements are **true** or **false**. If you say a statement is true, give a justification. If you say a statement is false, give a counterexample.

For some of these problems, you will need to recall the `fact` function:

```haskell
fun fact (0 : int) : int = 1
| fact n = n * fact (n-1)
```

**Task 1.**
Let $f : t_1 -> t_2$ be total. Then for all values $x : t_1$, $f x$ is valuable.

**Task 2.**
The function $\text{fact} : \text{int} -> \text{int}$ is total.

**Task 3.**
Let $f : t_1 -> (t_2 -> t_3)$ be total. Then for all values $x : t_1$, $f x$ is total.

**Task 4.**
If $f : t_1 -> t_2$, then $f$ is a value.

**Task 5.**
Let $f : t_1 -> t_2$ be a value. Then $f$ is valuable.

**Task 6.**
The function $\text{fact} : \text{int} -> \text{int}$ is valuable.

**Task 7.**
$\text{fact} x$ is valuable for all nonnegative $x : \text{int}$.

**Task 8.**
Let $f : t_1 -> t_2$ be a value. If for all values $x : t_1$, $f x$ is valuable, then $f$ is total.
3 Listen Up

Define the following functions in code/lists/lists.sml.

Task 9.
Define a function

\[
\text{head} : \text{int list} \rightarrow \text{int option}
\]
REQUIRES: true
ENSURES: head L \(
\Rightarrow
\) SOME x if L is non-empty, where x is the first element, or NONE if L is empty

Task 10.
Prof. Neumann has decided to reward the TAs handsomely for their hard work. He wants to add a bonus $3 to the February paycheck for each of the TAs. Write a function

\[
\text{credit} : \text{int list} \rightarrow \text{int list}
\]
REQUIRES: true
ENSURES: credit L \(
\Rightarrow
\) L’ where each element is $3 more

that takes in a list representing the amounts payable to each of the TAs for the month of February. Credit $3 to each of the TAs and make their day!

Task 11.
Write a function

\[
\text{evens} : \text{int list} \rightarrow \text{int list}
\]
REQUIRES: true
ENSURES: evens L \(
\Rightarrow
\) L’ where all odd elements of a list are filtered out, but the order of the original list L is preserved

For example,

\[
\begin{align*}
\text{evens } [0,0,4] &= [0,0,4] \\
\text{evens } [] &= [] \\
\text{evens } [0,0,4,9,3,2] &= [0,0,4,2]
\end{align*}
\]
Task 12.
Write a function

```ocaml
def lastPositive : int list -> int option

REQUIRES: true
ENSURES: lastPositive L =⇒ SOME x where x is the last positive number in the list, if it exists, or NONE if no such x exists
```

Task 13.
Define a function

```ocaml
def sequence : int option list -> int list option

REQUIRES: true
ENSURES: sequence L =⇒ NONE if L contains at least one NONE or SOME [x₁, x₂, ..., xₙ] if L is of the form [SOME x₁, SOME x₂, ..., SOME xₙ]
```

Task 14.
In code/lists/lists.sml, write a function

```ocaml
def bitAnd : int list * int list -> int list

REQUIRES: A and B only contain 1s and 0s
ENSURES: bitAnd (A,B) =⇒ l where l is a list with a logical ‘and’ performed on the corresponding elements of the two lists interpreting 1 as true and 0 as false. If the end of either list is reached, we stop evaluating
```

For example,
```
val [1] = bitAnd ([1],[1])
val [0,1,0] = bitAnd ([1,1,0],[0,1,0])
val [] = bitAnd ([],[[]])
val [1,0] = bitAnd ([1,0,1],[1,1])
```

Task 15.
In code/lists/lists.sml, write a function

```ocaml
def interleave : int list * int list -> int list

REQUIRES: true
ENSURES: interleave (A,B) =⇒ l where l is a list built by alternating between the elements of A and B, until we reach the end of one of the lists, after which we take the remaining elements from the other list
```
For example,

\[
\text{val } [2,4] = \text{interleave } ([2],[4])
\]

\[
\text{val } [2,4,3,5] = \text{interleave } ([2,3],[4,5])
\]

\[
\text{val } [2,4,3,5,6,7,8,9] = \text{interleave } ([2,3],[4,5,6,7,8,9])
\]

\[
\text{val } [2,3] = \text{interleave } ([2,3],[])
\]


We will take a look at some trees (notice, they are indeed quite beautiful).

Recall the definition of trees from lecture:

```ml
datatype tree = Empty
             | Node of tree * int * tree
```

Recall from lecture that recursive functions on trees usually have the following form:

```ml
fun foo (Empty : tree) = ...
             | foo (Node(L, x, R)) = ...
```

that is, in order to specify a function by recursion on the structure of a tree, it suffices to specify the value of the function at `Empty` and the value of the function at `Node(L, x, R)` in terms of the value of the function on `L` and `R`.

For example, recall the function `size` which gives the number of `Node`s in a tree:

```ml
(* size : tree -> int
 * REQUIRES: true
 * ENSURES: size T = > the number of nodes in T *)
fun size (Empty : tree) : int = 0
             | size (Node(L, x, R)) = 1 + size L + size R
```

To practice writing tree functions, we’ll have you implement the function `depth` which gives the max depth of a tree (the max depth of the empty tree is 0, and the max depth of a nonempty tree is the maximum depth of all the nodes in a tree):

**Task 16.**

In `code/depth/depth.sml`, define

```ml
depth : tree -> int
REQUIRES: true
ENSURES: depth T = > the maximum depth of T
```

For example,

```ml
val 0 = depth Empty
val 1 = depth (Node (Empty, 1, Empty))
val 2 = depth (Node (Empty, 4, Node (Empty, 5, Empty)))
```

**Task 17.**

Now we’ll write a more complex function to operate on trees.

We define the `leaves` of a tree to be nodes which have the form `Node (Empty, x, Empty)` for some `x : int`. 
Define the function:

```ocaml
leaves : tree -> int list
REQUIRES: true
ENSURES: leaves T \rightarrow the values at the leaves of T
```

For example,

```ocaml
val E = Empty
val T = Node (E, 3, E)
val T' = Node (Node (E, 1, E), 2, T)

val [] = leaves E
val [3] = leaves T
val [1, 3] = leaves T'
```

Because we don’t care about the order of the returned list, your solution may behave differently than the examples in that regard. You are allowed to use standard library functions, such as \@.
5 Proving Totality

Consider the function \( \text{foo} : \text{int list} \rightarrow \text{int list} \) given by

\[
\begin{align*}
\text{fun foo} \; (\[] : \text{int list}) \; : \text{int list} &= [] \\
| \text{foo} \; (x::xs) &= x :: \text{foo}(\text{rev} \; xs)
\end{align*}
\]

where \( \text{rev} \) is a given function of type \( \text{int list} \rightarrow \text{int list} \) such that for all integer lists \( L \), \( \text{rev}(L) \) evaluates to the reverse of list \( L \).

Task 18.

Prove the following theorem by induction on the length of \( L \):

**Theorem**: For all values \( L : \text{int list} \), \( \text{foo} \; L \) evaluates to a value.

You may NOT make any assumptions about how \( \text{rev} \) is implemented. Instead, you may make use of the following lemmas:

**Lemma 1**: For all values \( L : \text{int list} \), \( \text{rev} \; L \) evaluates to a value.

**Lemma 2**: If \( L : \text{int list} \) has length \( n \), then \( \text{rev} \; L \) has length \( n \).