Homework 3
Types $\cong$ Theorems

98-317: Hype for Types

Due: 25 Sept 2018 at 6:30 PM

1 Introduction

In class, we discussed the idea that we can use types to express logical propositions, and that creating a value of a particular type corresponds to proving a proposition. In this homework, you will explore this idea further, writing some functions in SML to prove various propositions in logic.

Turning in the Homework You should submit any code files to Autolab by running the Makefile (type the command `make`) in the hw3 directory and submitting the resulting `hw03.tar` file to the Homework 3 assessment. The Autograder for this assignment just checks if your code typechecks, since if it typechecks, it's correct!

2 Logic in SML

In this homework, you will be implementing a structure called `Proofs` so that it ascribes to the signature `PROOFS`. The `PROOFS` signature declares 10 values with types that correspond to logical propositions. In order to implement the signature, you will have to prove the propositions!

For the most part, you can decide what the behavior of the values in the Proof structure should be; they just have to have the correct type. However, there are a few rules:

1. All of the definitions in the `Proof` structure must be *values*. This prohibits code like

   ```
   val demorgan_cong = raise Fail "This typechecks!"
   ```

2. All function definitions in the `Proof` structure must be *total*. That is, they must terminate on all possible inputs. This prohibits code like

   ```
   fun contrapos f = raise Fail "This typechecks!"
   ```

To help you implement these functions, we've given you a couple of definitions in the `Definitions` structure. Its signature is reproduced below.

```
signature DEFINITIONS =
sig
  type ('a,'b) iff = (('a -> 'b) * ('b -> 'a))
  datatype ('a,'b) or = L of 'a | R of 'b
  type void
  type 'a not = 'a -> void
  val abort : void -> 'b
end
```

The `iff` type is meant to represent bi-implication. A value of type `('a, 'b) iff` is a pair of two functions, one of type `'a -> 'b` and the other of type `'b -> 'a`. For example,

```
(fn x => (x, x), fn (x, _) => x) : ('a, 'a * 'a) iff
```

represents a proof of $A \Leftrightarrow A \wedge A$ (that is, a proof of $A \Rightarrow A \wedge A$ and $A \wedge A \Rightarrow A$).

The `or` type is the same as the sum type we discussed in the typechecking lecture, and is meant to represent logical or. A value of this type can either have the L constructor and contain a value of type 'a, or have the R constructor and contain a value of type 'b.

The `void` type is a type for which it is impossible to construct a value. This represents False. If you could somehow construct void, it would allow you to construct a value of any type. This is what `abort` does: it is a total function from void to any arbitrary type 'c. This mirrors how from False, you can conclude anything.

The `not` type is shorthand for `'a -> void`. In logic, $\neg A$ is defined to be "If I assume A, then I can reach a contradiction," so this is also how it is defined in terms of types.

You can use these definitions as much as you'd like in your code. Good luck!

Note: If you spend more than 30 minutes on this assignment, stop working and turn in what you have. We will accept it! [1] Some of the proofs are a little tricky, so if you get stuck, move on and try a different one. This is meant to be a fun puzzle - don't let it become frustrating or stressful!

[1] We will not, however, accept blank submissions or submissions which show a clear lack of effort.