Homework 3
Types $\cong$ Theorems

98-317: Hype for Types

Due: 6 Februrary 2018 at 11:59 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.

This homework is divided into four parts: Required, Useful, Fun, and Completely Unnecessary But Also Fun. You will receive credit for this homework if you turn in something (not necessarily something working) for the "required" portion.

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. There is an autograder installed in Homework 3, but it is there only for your benefit: we will not base your grade on its results.

You should turn in your written solutions in class.

2 Required

In this section, you'll be translating some logical propositions into types, and then writing proofs of those propositions by writing a value of that type. You should write your solutions in Standard ML.

Example Write the proposition

$$(A \wedge B \to C) \to (A \to B \to C)$$

as a type, and then write an expression of that type.

Solution
Type: `('a * 'b -> 'c) -> ('a -> 'b -> 'c)`
Program: `fn f => fn x => fn y => f (x, y)`

LaTeX Typesetting your solutions isn't required, but if you want to typeset your solutions in LaTeX, here is how we typeset our solution:

```
\paragraph{Solution} \hfill

Type: \texttt{('a * 'b -> 'c) -> ('a -> 'b -> 'c)}

Program: \texttt{fn f => fn x => fn y => f (x, y)}
```

Req Task 1 Write the proposition

$$A \to (B \to B)$$

as a type, and then write an expression (proof) of that type.

Req Task 2 Write the proposition

$$A \wedge (A \to B) \to B$$

as a type, and then write an expression (proof) of that type.

Req Task 3 Write the proposition

$$(A \vee B \to C) \to ((A \to C) \wedge (B \to C))$$

as a type, and then write an expression (proof) of that type.

3 Useful

In class, we talked about how it's impossible to define a function for the proposition $\neg\neg A \to A$. This is because programs represent proofs in *constructive* logic. In constructive logic, a proposition is true exactly when there is a proof of that proposition. This is different from the logic we're all used to, classical logic, where any proposition that can be proven *not false* is also true.

Turns out, we can also write programs that correspond to proofs in classical logic (where $\neg\neg A \to A$ holds) by adding a new language feature called a *continuation*[1]. Continuations allow us to implement the main feature that separates classical logic from constructive logic: proof by contradiction. In particular, continuations allow us to implement two functions:

```
val assume : ('a not -> 'a) -> 'a
val contra : 'a not -> 'a -> 'b
```

The function `assume` derives A from assuming $\neg A$ and proving a contradiction (A). The function `contra` allows you to prove anything (B) from a contradiction ($\neg A$ and A).

These functions are specified in `classical.sig` and implemented in `classical.sml`. Take a look at `classical.sig` and make sure you understand the types in there.

Among the other code files in your handout, there are two files called `proofs.sig` and `proofs.sml`. The first describes a signature called `PROOFS`. Read it over and make sure you understand the types in it.

Useful Task 1 Implement a structure in `proofs.sml` ascribing to the `PROOFS` signature. The values in the structure can have any behavior you want, as long as they have the correct type, with one exception: all functions must be total. That is, the functions must terminate; they may not loop infinitely or raise exceptions. You may assume `contra` and `assume` from `Classical` are total functions.[2]

Some Hints:

1. If you're stuck on an implementation, think about how you would write a proof of the equivalent proposition. This may help guide how your write your code.
2. Make liberal use of `contra` and `assume` from `Classical`.
3. You can use earlier values in your structure to implement later ones. You can also rearrange values in your structure. Use both of these facts to your advantage.

[1] These are not the continuations you learned in 15-150, but they are the continuations taught in 15-312.

[2] Even though a quick glance at the types should tell you that they are *definitely* not. A more precise way to say this would be: you cannot create your own infinite loops or raised exceptions. Only those created by contra and assume from Classical are allowed.

4 Fun

Fun Task 1 Some of the values in this signature can be implemented without `contra` and `assume`. They correspond to propositions that are also true in constructive logic. Conversely, others cannot be implemented without `contra` and `assume`. They correspond to propositions that are true in classical logic, but not in constructive logic. They are:

```
val law_ex_mid     : unit -> ('a, 'a not) or
val demorgan_cong  : (('a * 'b) not, ('a not, 'b not) or) iff
val dne            : 'a not not -> 'a
val contrapos      : ('a -> 'b,'b not -> 'a not) iff
val neg_over_iff   : (('a,'b) iff not,('a not,'b) iff) iff
```

However, it is possible to prove in constructive logic that they are *not false*. Therefore, the following alternative version of these functions can be written without `contra` and `assume`:

```
val cons_law_ex_mid     : (unit -> ('a, 'a not) or) not not
val cons_demorgan_cong  :  ((('a * 'b) not, ('a not, 'b not) or) iff) not not
val cons_dne            : ('a not not -> 'a) not not
val cons_contrapos      : (('a -> 'b,'b not -> 'a not) iff) not not
val cons_neg_over_iff   : ((('a,'b) iff not,('a not,'b) iff) iff) not not
```

Implement these values at the bottom of your Proofs structure.

5 Completely Unnecessary but Also Fun

Unnecessary Task 1 In `classical.sml`, implement a second structure `ClassicalExn`, with the following type definition for 'a not:

```
type 'a not = 'a -> exn
```

Technically, your functions can have any behavior you want, but to make it more interesting, you should imitate the behavior implemented in the Classical signature. That is, what we really want you to do is implement the first-class continuations of SML/NJ using exceptions. A good explanation of what continuations are can be found here:

`http://www.cs.cornell.edu/courses/cs312/2006sp/lectures/lec22-23.html`.

The continuation type is interesting by itself, but sadly we won't get to it in this class.