

Questions: Semantic and Computational Issues

Veneeta Dayal and Chung-chieh Shan

September 25, 2006

1. Revised course requirements

We have revised what we require of registered students as follows. We intend these requirements to be easier to satisfy.

Everyone should write and present a final paper on some topic related to questions. In addition, cognitive-science students should

1. discuss computational issues in the final paper;
2. implement a linguistic explanation in code and present it on December 15; or
3. a new option: do four sets of recommended programming exercises and present one of the sets a week after it is distributed.

With the latter two options, the final paper can just be a short squib and does not need to be related to the programming. Again, we encourage collaboration.

2. Exercises from September 6

All examples are online at <http://www.cs.rutgers.edu/~ccshan/questions/>

1. Give an example of something to which you can apply `good`, defined by

```
good f = f 0 && f 1
```

What is the type of `good`? Can you apply anything to `good`?

2. Define the proposition (of type `Prop`) that nobody saw anybody. How do you check that your definition is correct?
3. In Hamblin's semantics of questions, the extension of a question is not its set of true answers, but its set of answers. Change `q` in `Karttunen1.hs` and `Karttunen2.hs` to compute the Hamblin extension of *Who did Alice see?*. How do you check that your changes work?
4. The three lines in the definition of `q` in `Karttunen1.hs` is rather repetitive. Define an auxiliary function `f` to capture this repetition, so that the definition of `q` simplifies to

```
q w p = p w && (p == f Alice || p == f Bob || p == f Carol)
```

What is the type of your `f`? Can you simplify this definition of `q` further?

5. A semanticist can read *Two Dozen Short Lessons in Haskell* to learn Haskell. In particular, Chapter 4 explains *list comprehensions*, a convenient way to build lists. Use a list comprehension to make the definition of `q` in `Karttunen2.hs` less repetitive.

3. Algebraic data types

A tuple belongs to a Cartesian product.

```
type World = (E -> Bool, E -> E -> Bool)

reality :: World
reality = ((\x -> ...), (\seer seen -> ...))

saw :: World -> E -> E -> Bool
saw (p,r) seen seer = r seer seen
```

It is good practice to label the tuple, so that `World` is not compatible with any other type. Remember: `type` defines a type *synonym*, whereas `data` defines a *new* type.

```
data World = MakeWorld (E -> Bool) (E -> E -> Bool)

reality :: World
reality = MakeWorld (\x -> ...) (\seer seen -> ...)

saw :: World -> E -> E -> Bool
saw (MakeWorld _ r) seen seer = r seer seen
```

Often the value constructor `MakeWorld` is named the same as the type `World`—a pun.

A binary tree is a disjoint union of Cartesian products. It is *polymorphic* and *recursive*.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Here `Tree` is a type constructor: it takes any type `a` as an argument, so `Tree Int` is the type of a binary tree whose leaves are labeled with `Int` values.

How to count the leaves of a tree? How to compare the shape of two trees? What are the types of these functions?

4. Parametric vs ad-hoc polymorphism

How many functions are there with the type $a \rightarrow a$? What about $a \rightarrow a \rightarrow a$?

Not all types can be conjoined! Partee and Rooth define *conjoinable* types.

```
class Conjoin c where
  conjoin :: c -> c -> c
  disjoin :: c -> c -> c
```

```
> :type conjoin
conjoin :: (Conjoin c) => c -> c -> c
```

We define how to conjoin and disjoin two values of each conjoinable type. Note the recursion below in `conjoin` and `disjoin` to the right of `=`.

```
instance Conjoin Bool where
  conjoin = (&&)
  disjoin = (||)

instance (Conjoin b) => Conjoin (a -> b) where
  conjoin f g x = conjoin (f x) (g x)
  disjoin f g x = disjoin (f x) (g x)
```

```
> pretty (conjoin (\x y -> not x || y) (\x y -> not y || x))
{False: {False: True, True: False},
 True: {False: False, True: True}}
```

More examples of useful type classes can be found in the Haskell Prelude (especially the classes `Eq`, `Ord`, and `Show`) and in the files `Finite.hs` and `Pretty.hs` on the course Web site (especially the classes `Finite` and `Pretty`). These files illustrate more features of type classes: default member definitions; inheritance among classes.

5. Today's assignment: types

1. Write a function of type `Tree Int -> Int`, to sum the leaves of the input tree.
2. How many functions can you define with the type $(a \rightarrow a) \rightarrow (a \rightarrow a)$? (Recall that the type variable `a` here is implicitly universally quantified.)
3. Define Groenendijk and Stokhof's LIFT (Figure 1 on page 447) in Haskell:

```
lift  :: (World -> Bool) -> ((World -> Bool) -> Bool) -> Bool
lift = ...
```

What type does Haskell infer for your definition if you leave out the explicit type annotation above? Does this type make sense?

4. Implement a Haskell function to convert an n -place relation to a question (at the top of Groenendijk and Stokhof's page 442). What is its type? Does the type make sense?
5. Write an additional instance for the `Conjoin` class, for a 2-tuple:

```
instance (Conjoin a, Conjoin b) => Conjoin (a, b) where ...
```

What happens if you remove the context `(Conjoin a, Conjoin b)`? Why?

6. Extend the `Conjoin` class and its instances with Groenendijk and Stokhof's "argument-lifting" operation (example 57 on page 446).

```
class Conjoin c where
  conjoin :: c -> c -> c
  disjoin :: c -> c -> c
  argLift :: (a -> c) -> ((a -> Bool) -> Bool) -> c
```