Closed Sub-Types of Sets

by Sven Nilsen, 2018

A set can be used as a generalized data structure for computation where one speaks about collection of values instead of single values. An interesting application of sets is when the set is unknown. This is closely related to many concepts expressed in natural language, because natural language evolved to be used across a wide variety of contexts while preserving easily recognizable sounds.

For example, in the sentence “do you see the car?”, the concept “the car” refers to a set of type “car” of size 1. In the sentence “do you see the cars?” the concept “the cars” refers to a set of type “car” of size more than one.

|{car}| : (= 1)	“the car”
|{car}| : (> 1)	“the cars”

The kind of concepts used in the example above is a sub-type of a set. It is not a sub-type of a variable of type “car” because one can not speak of more than one car when there is just one car. On the other hand, one can speak of a single car from a set of cars. Therefore, speaking of sets is a generalization of speaking of variables.

There are some advantages with using sub-types of sets as the basic fundament of words. For example, when a physical objects moves between many states over a long time, it gets very hard to keep track of all that information. One way is to think of the history of an object as a separate thing from the object itself. This is necessary when thinking about the object as a single variable. However, by thinking of the physical object as a sub-type of a set that might contain all physically possible states, one can ignore the order of how events occurred during the history. Instead, one can think of the “recent history” as a physical object on its own that is related to the physical object in a larger context. Figuring out what happened becomes a problem-solving exercise to map possible histories to the set sub-type, which does not define the history accurately but to some degree we might find useful.

When I say “do you see the car?”, the person I talk to tries to figure out what I mean by “the car” to understand what I am talking about. If there are no cars nearby, then the person will believe I am talking non-sense, or perhaps believe I am talking above another kind of “car”, like an illustrated car in an image.

This notion of figuring out what words mean on the fly is important for the use of language. There is a trade-off in how easily the sound of the word “car” can be recognized and precision of language. I could point to the car through language by giving its chemical composition, but it would be much harder to understand. It is therefore interesting to take a look at what can be said about sets that is easy to transmit through a slow communication channel. One can also see this as a challenge to figure out what concepts makes it easy to imagine things: Not too complicated, but not too vague either.

So far I have only talked about sub-types of sets that say something about its size. The things one can say about the size of a set is useful when the objects I am talking about is in the immediate surroundings. Now I want to talk about sub-types of sets that are useful localize objects.

Imagine that you have a map which splits an area up into regions. Every point on the map belongs to exactly one region. If the map contains a finite number of points, then there is a finite number of regions. One can also have a map containing infinite points, but which every point belongs to a region and the number of regions is finite.

When there is a finite number of regions and every point belongs to a region, then speaking about a set of points relative to these regions can only be expressed in terms of what is called a “closed sub-type”.

A closed sub-type has the property that what it means has a dual representation: It is not its own inverse. While this seems obvious at first, it becomes practical and non-trivial when eliminating the ability to speak of the inverse directly.

a type = not some other type	(some A and some B are not exclusive)
some type = not another type	(an A and a B are exclusive)

For example, Alice traveled somewhere in the world and Bob wonders where she has been. When Bob talks to Alice, they communicate sentences like “have you been in Germany?” and “I was in Sweden”. Just because Alice was in Sweden, does not mean she could not have been in some other country too. By default, when Alice says she has been somewhere, she does not mean to exclude other places.

In type systems for most programming language it is more natural to use the dual representation. Instead of saying “this variable `x` had type `A`” we say “this variable `x` has either type `A` or the empty type”. When we code `x : A` we mean by default that `x` has no other type. However, it might be that there no solution for `x`, which is the empty type.

It turns out that these two ways of thinking are logically equivalent to each other, but only when sub-types are closed. Everything that can be said in one system can be said about the other.

For either system, it takes only a single bit together with an index to tell which bit, to communicate the default representation in that given system. Yet, to the say the same in the other system requires one bit for every other closed sub-type.

Notice that in the second case, we define a type directly. However, this is within the language of the type system. If we needed to talk about how the type system works, then we need a language designed to speak of sub-types of sets of types! In a such scenario, it makes sense to talk about contexts where sets of types are finite, and therefore one can use the theory of a “closed sub-type” to talk about them. In some programming languages, one can take the union of two types, e.g. `x : A | B`. In the type inference algorithm of such type systems, it could be useful to derive the type of `x` from how it is used in the code, which is more similar to the previous example with Alice and Bob. In other words, the opposite representation is more useful when reasoning in the opposite direction.