Abstract Sub-Types

by Sven Nilsen, 2018

Path semantics is an extremely powerful notation for function analysis and theorem proving, but since it is constructed using a bottom-up approach, it is desirable to offer a top-down approach found in other theories of mathematics such as category theory. I call the bottom-up approach to mathematics for "concrete" and the top-down approach for "abstract". In this paper I represent a technique to navigate between concrete and abstract levels of path semantics. This means that the concrete and abstract approaches are glued together in a well defined way. Abstract path semantics is used to reason about problems where sub-problems are too hard to comprehend and analyze at first sight, but the style of reasoning is not similar to creating a predictive model (which is captured by the internal language of path semantics). The predictiveness of the abstract level is in the abstract sense, which depends on the definition of abstract sub-types. Since abstract sub-types use overloaded functions depending on context, the predictiveness of the abstract level is also overloaded, and therefore different from the concrete predictiveness within concrete path semantics. As an example I use interpretation of AABB data structures to show how concrete and abstract levels of reasoning can be connected.

An abstract sub-type is a sub-type defined by some function where the function is overloaded depending on the context. With other words, when using an abstract sub-type on a particular problem, it is necessary to specify the missing information in order to have a well defined solution. Abstract sub-types are useful when it is too hard to think about how to add the missing information while outlining the idea, while reasoning depends on inter-relational properties of different abstract sub-types.

Since abstract sub-types are functions (they are overloaded, but still functions), path semantics can be used to describe the inter-relational properties at the abstract level. The abstract level is decoupled from the concrete level, so all existing notation for concrete path semantics can be overloaded as well.

In path semantics, `∃f` means the existential path of `f`. When `f` is not a function, this combination can be overloaded to mean something else. One such useful overloading is `∃x` and `∀x` where `x` is `T` (true) or `F` (false). These abstract sub-types have some useful inter-relational properties that can be described concisely in abstract path semantical notation:

∃T[not → id] <=> ∃F
∃T[id → not] <=> ∀F

not · not <=> id

Here, `not` is not a boolean function! It is called "not" just because it has similar properties to the `not` function in concrete path semantics. What it means here depends on the context of the problem. Likewise the `·` operator does not mean function composition, but composition in the abstract sense.

Using the definition above, it is possible to prove how `∀T` and `∃T` are related in the abstract level:

∃T[not → id][id → not] <=> ∃T[id · not → not · id]
∃F[id → not] <=> ∃T[not → not]
∀T <=> ∃T[not]

Now, I can use this in a concrete situation. An Axis-Aligned-Bounding-Box (AABB) is a common data structure in graphics and game programming. It consists of one minimum coordinate and one maximum coordinate, such that all points between the minimum and maximum corner when moving along all axes belong to the set of points described by the AABB. For example, in Rust you can code:

```
struct AABB<T> { min: T, max: T }

AABB<[f32; 2]>          // AABB for 2D
AABB<[f32; 3]>          // AABB for 3D
AABB<[f32; 4]>          // AABB for 4D
```

An application of AABB is to approximate the location of a complex 3D object in space. Operations performed on AABBs are much cheaper than to check collision between every point on two complex 3D objects. In a such situation, the AABB is interpreted as the minimum box required to contain all points defined by the 3D object.

However, AABBs are useful for many things beside this specific interpretation. By relaxing the constraint of having a minimum box, it can be used for other applications, such as analyzing functions of type `$\mathbb{R}^N \to \mathbb{B}$` (mapping real numbers to boolean values).

One nice mathematical property of AABB is that it is self-similar when increasing the number of dimensions. I call one step down in dimension as the “surface” AABB.

There are four interpretations of AABB that are useful in particular for analysis:

∃T – When there exists a solution in all surfaces (holds for a minimum box)
∃F – When there exists a non-solution in all surfaces
∀T – When all points on all surfaces have a solution
∀F – When no points on all surfaces have a solution

Still, even an AABB can be interpreted in these four ways, the problem is how to obtain such boxes in the first place. An easy way to do this is to start with a large box, then shrink it gradually until some condition is satisfied or the box is empty. Since there are four conditions, there are four functions:

$f : [AABB_N \times (\mathbb{R}^N \to \mathbb{B}) \to opt[AABB_N]] \wedge [len]\ 4$
$f_0 : \exists T$
$f_1 : \exists F$
$f_2 : \forall T$
$f_3 : \forall F$

Notice that I used the symbols here as abstract sub-types. This means that the function `∃T` is overloaded with the function `$f_0$`, the function `∃F` is overloaded with `$f^1$`, and so on.

One might try substituting `∃T` and `∃F`, but this leads to a type error:

```
∃T[not → id] <=> ∃F
f₀[not → id] <=> f₁          THIS IS A TYPE ERROR (`not : bool → bool`)
```

The reason substitution does not work is that the symbols are connected *in the abstract sense*.

So, what does it mean here that `∃T[not → id] <=> ∃F`?

Notice that when you have written `$f_0$` in code, one can use it to define `$f_1$` for all functions. In the same way, when you have written `$f_2$`, it can be used to define `$f_3$`. This is very useful because this saves a lot of work when using AABBs of many dimensions:

$$f_0(x, \text{not} \cdot g) = f_1(x, g)$$
$$f_2(x, \text{not} \cdot g) = f_3(x, g)$$

$$g : \mathbb{R}^N \to \mathbb{B}$$

Therefore, one can go from the abstract level to the concrete level:

∃T[not → id] <=> ∃F	abstract level
$f_0(x, \text{not} \cdot g) = f_1(x, g)$	concrete level

Here, the `not` function used on the abstract level means that the `not` boolean function is used to compose the function argument on the concrete level. It is natural to use this on the input of the asymmetric path on abstract level, because this operation happens on the function input at the concrete level.

Then, what does the following mean?

∃T[id → not] <=> ∀F

Remember that an AABB interpreted as `∃T` means there exists a solution for all surfaces.

Using logic, one can write this:

∀ s : Surface { ∃ p : s { g(p) } }

However, here it is natural to think about an AABB as an abstract property `P` such that:

¬(∃ P) <=> ∀ ¬(P)
¬(∀ P) <=> ∃ ¬(P)

Because, if there exists no solution on any surface, it means the same as all surfaces only contain non-solutions. This is natural when considering all surfaces as one, but it not directly describable using the rules of logic alone. It is a higher order idea, where you consider the logic expression as constructed from some higher order information where this transformation is a valid logic operation. With other words there is an outer composition that translates into an inner composition that depends on the logical expression:

∀ s : Surface { ¬(∃ p : s { g(p)) } }
∀ s : Surface { ∀ p : s { ¬(g(p)) } }

The logical expression describes what the AABB means, but not how it is obtained, so this transformation is valid in the abstract sense by modifying code. When you have written the code for `$f_0$`, it takes only a few changes to get `$f_3$`.

Still, because one defines the following abstract operation of an AABB:

$$\neg(\exists\ P)$$

One can think of this as a composition:

$$not \cdot h$$

Where `h` describes the interpretation of the AABB.

Now, looking at:

$$\exists T[id \to not] <=> \forall F$$
$$\exists T[id \to id \cdot not] <=> \forall F$$
$$not \cdot \exists T[id \to id] <=> \forall F$$
$$not \cdot \exists T[id] <=> \forall F$$
$$not \cdot \exists T <=> \forall F$$

In the abstract sense, this has the same form as composition of the output, therefore:

$$\exists T[id \to not] <=> \forall F$$
$$not \cdot \exists T <=> \forall F$$
$$f_3 : \forall F <=> f_3 :\ not \cdot \exists T$$
$$f_0 : \exists T$$

It should therefore be possible to make a few changes to `$f_0$` to get `$f_3$`:

$$\neg(f_0) <=> f_3$$

Notice that `$f_3 :\ not \cdot \exists T$` is also an abstract sub-type, because when you compose a two functions where one is abstract, then both must be abstract. Composition is a constrained function with a trivial path requiring arguments to have same abstract value (it means composition is a partial function), so it is sufficient to know whether either the first or the second argument is abstract:

$$compose : function \times function \to function$$
$$\forall compose <=> eq \cdot (abstract, abstract)$$

$$compose : [abstract]\ a \times [abstract]\ a \to [abstract]\ a$$
$$compose[abstract] => \{fst, snd\}$$

$$abstract : function \to bool$$

What I did above is creating a model in concrete path semantics, which is very different from using abstract path semantics. In concrete path semantics, `eq · (abstract, abstract)` means, literally, function composition using the standard definition of `eq`. In abstract path semantics you weaken the standard definition to some specific properties, such that functions like `not` and `eq` should not be interpreted in the concrete sense.

One can reason about combinations of concrete and abstract levels using concrete models.

Since what the abstract symbols are defined using the language of path semantics, one can argue that the language of path semantics is itself disconnected from path semantics. This seems intuitively true at first, but can not be true because it leads to a paradox: How can a language be disconnected from itself?

A more accurate expression of a the idea is that path semantics seems applicable also when used in the abstract sense. From the example I demonstrated with AABBs, this is obviously useful. However, is the concept of abstract reasoning separate from path semantics? Is it fundamentally different or less trustworthy?

All of concrete path semantics builds upon the single axiom of path semantics that tells when two symbols are identical, then there are two collections of knowledge, about how the symbols are used, that also are identical. Concrete path semantics assumes first that the axiom is interpreted as atomic functions, but then it constructs new ideas at higher and higher levels where the interpretation of the axiom takes on a new meaning each time that has not occurred before.

When moving from concrete reasoning alone to combined concrete and abstract reasoning, each symbol is assigned an additional truth value defined by the `abstract` function. A symbol is abstract when it occurs in a context where it depends on or is being depended on some abstract value. Therefore, the concrete level and abstract level of reasoning is separate. Only by using abstract sub-types it is possible to glue these two kinds of reasoning together.

Abstract path semantics has no computational property of concrete values. For example, the `not` function when used in abstract sense has no argument `false` or `true` which it can compute `true` or `false`. The only computational property at abstract level is that of abstract functions. To derive this kind of computation, one must use theorem proving. Therefore, at the abstract level, theorem proving is not redundant as a predictive tool, but necessary.

The abstract level of reasoning can be trusted to the degree that abstract sub-types are trusted, and to the degree one believes the abstract theorem proving to be correct. This means in principle, one can assign a very high confidence in the combined reasoning, but of course the abstract level remains abstract. The very fact that the abstract level denies concrete values means there is a weakening of the assumption that the theory is built upon atomic functions. It is this weakened assumption that makes it feel less grounded in reality. Still, it would be more precise to formulate it as overloaded syntax for specific contexts, but still just as grounded in reality as concrete path semantics. The easy error to make here is to assume concrete path semantics is strongly grounded, which is not correct. It is only grounded in the sense that it consistently predicts identity of symbols!

Therefore, grounding of truth is a question of consistent reasoning and the ability to check the answers. This applies at all levels. What is different this time, when extending reasoning to both concrete and abstract levels, is that the abstract levels make overloaded predictions of the concrete level. There is no way to check these predictions without first defining the concrete level. However, since the concrete level is grounded, the overloaded predictions are in principle just as grounded. They just are assigned weaker subjective beliefs to be true because our ability to check answers is limited.

One way to describe this process, is as if the confidence in concrete path semantics inspire confidence in abstract path semantics, because abstract path semantics is a useful predictive tool for concrete path semantics for various problems. Some people hesitate to believe abstract ideas, which is rational when you are not able to check answers yourself, but since the language of path semantics is reused, this technique should be assigned high confidence by practitioners of path semantics.