Context Transformation & Learning

by Sven Nilsen, 2017

Difficulty: Expert knowledge of path semantics, including familiarity with asymmetric path notation. Some experience with theorem proving is recommended.

Path generators can add information in a way that describes relationship between existing code as knowledge and new formal intentions of applying the knowledge.

By using a dependently typed language like Idris (http://www.idris-lang.org), one can write detailed program specifications as code, then write the program, then prove that the program follows the specifications, even in a such way that the specifications and the proof can be erased afterwards. The specification is a partial knowledge about the inner workings of a program, but their *formal intentions* must be expressed as path relationships.

Therefore, path semantics is a useful tool for directly expressing intentions of specifications. This is somewhat surprising, because path semantics expresses this in the language of functions. No new quantifiers are required!

A family of functions `f(a)` that vary by context `a` and take `A` as argument can be transformed into a single function `h` that takes `B` as argument by encoding `a` into `B` with `g`.

> f(a): A → C
>
> g(a): A → B
>
> h: B → C
>
> f(a)[g(a) → id] <=> h

For all higher order functions `f` of argument `a` there exists a higher order function `g` of argument `a` such that the path `f(a)[g(a) → id]` exists:

> ∀ f, a { ∃ g { ∃ f(a)[g(a) → id] } }

Notice that `f(a)` returns a function, which means `a` is erased and forgotten. The path generator `g(a)` changes the data representation of the "new" input argument such that `a` is taken into account. It does not tell you how `a` was used by `f`, only that `a` is observed in some way. Therefore, there exists a function that recover the yet-to-be-known behavior of the whole program, because one can just apply the function `f` and get back the functionality.

While this is always true, the useful thing about this law is when we do not know `f`, but only observe outputs of it by describing our partial knowledge as independent functions. If `f` exists, informally thought of as the "intentions of specifications", then there exists some data structure that can be used to encode this intention formally as programming code.

In natural English this can be expressed as:

> *When you learn something, but do not know how to use it, you can learn how later.*

Example: Common sense for machine learning applications

Alice writes a common sense library describing activities performed under various seasons and whether the activity performed is inside or outside a building. She uses Idris for this task. Here are the data declarations:

```
data Inout = Inside | Outside
data Season = Spring | Summer | Autumn | Winter
data Activity = Hockey | Swimming
data Context = MkContext Inout Season
```

Next she defines a function `f` where she does a detailed case-by-case specification:

```
f : Inout -> Season -> Activity -> Bool
f Inside Spring Hockey = True
f Inside Spring Swimming = True
f Inside Summer Hockey = False
f Inside Summer Swimming = True
f Inside Autumn Hockey = True
f Inside Autumn Swimming = True
f Inside Winter Hockey = True
f Inside Winter Swimming = True
f Outside Spring Hockey = False
f Outside Spring Swimming = False
f Outside Summer Hockey = False
f Outside Summer Swimming = True
f Outside Autumn Hockey = False
f Outside Autumn Swimming = True
f Outside Winter Hockey = True
f Outside Winter Swimming = False
```

She defines a path generator for `f` that takes the first two arguments and combines them into a type `Context` that will be used to interpret when an activity is performed, instead of having one argument for each variable:

```
g : Inout -> Season -> Context
g a b = MkContext a b
```

Alice thinks that `f` can be made shorter, and defines a new function `h` that uses `Context`:

```
h : Context -> Activity -> Bool
h (MkContext Inside _) Swimming = True
h (MkContext Outside season) Swimming = case season of
   Spring => False
   Summer => True
   Autumn => True
   Winter => False
h (MkContext Inside season) Hockey = case season of
   Summer => False
   _ => True
h (MkContext Outside Winter) Hockey = True
h (MkContext Outside _) Hockey = False
```

All of a sudden she wonders whether she got `h` right. Did she made any mistake?

Notice the difference in the function signatures:

```
f : Inout -> Season -> Activity -> Bool
h : Context -> Activity -> Bool
```

One can think of `f` as a higher order function, returning case-by-case functions:

```
f : (Inout, Season) -> (Activity -> Bool)
```

Alice could create another function `f` that used `Context` instead of two arguments:

```
f : Context → (Activity → Bool)

f0 Hockey = True
f1 Swimming = True
f2 Hockey = False
f3 Swimming = True
f4 Hockey = True
f5 Swimming = True
f6 Hockey = True
f7 Swimming = True
f8 Hockey = False
f9 Swimming = False
f10 Hockey = False
f11 Swimming = True
f12 Hockey = False
f13 Swimming = True
f14 Hockey = True
f15 Swimming = False
```

The path inferred by Alice is the following (using cross argument asymmetric path notation):

```
f[g × id → id] <=> h
```

However, she might as well interpret the path of `f` as a family of paths over `f(c)`, assuming both `f` and `g` takes `Context` as first argument:

```
f(c)[g(c) × id → id] <=> h
```

This shows that function currying automatically satisfies the natural relationship between context and knowledge modelling.

Alice is lucky, because in Idris she can prove the logical equivalence between `f` and `h`:

```
prove_f : (inout : Inout) ->
          (season : Season) ->
          (activity : Activity) ->
          (f inout season activity = h (g inout season) activity)
prove_f Inside Spring Hockey = Refl
prove_f Inside Spring Swimming = Refl
prove_f Inside Summer Hockey = Refl
prove_f Inside Summer Swimming = Refl
prove_f Inside Autumn Hockey = Refl
prove_f Inside Autumn Swimming = Refl
prove_f Inside Winter Hockey = Refl
prove_f Inside Winter Swimming = Refl
prove_f Outside Spring Hockey = Refl
prove_f Outside Spring Swimming = Refl
prove_f Outside Summer Hockey = Refl
prove_f Outside Summer Swimming = Refl
prove_f Outside Autumn Hockey = Refl
prove_f Outside Autumn Swimming = Refl
prove_f Outside Winter Hockey = Refl
prove_f Outside Winter Swimming = Refl
```

However, when applying machine learning in the real world, she might not get this lucky. The number of variables could get too big to prove, or she might end up with a much larger `Context`:

```
g : Inout -> Season -> Context
g a b = MkContext a b Rain AskingQuestion CarDrivesBy NearSchool HasElectricity …
```

How is it possible to learn that the variables of type `Inout`, and `Season` are more relevant than others? The function `h` might be wrong, or it could be other variables that really should be used instead.

The answer is, that in functional modelling, you need to separate “intentional use” from “realistic modelling”. A function is often too simple to reflect the complexity of the world. What it does, is *encoding the intention of observations from experience*. The function `h` means relative to `f` exactly what it means according to path semantics, nothing more, nothing less. It says a lot more than normal programming code, sure, but it does not say as much compared to real world complexity.

Learning of context never stops, but you can always learn it later.

In Alice’s case she has proven that she could transform 2 variables and still have a consistent view of what these functions do, but she could also repeated the process multiple times and ended up with a more complex `Context` type. Yet, what would be the point of repeating this process? What goal would she achieve? She can do it later anyway!

The nice thing about functional modelling is that it can be used in a narrow context, without dragging the whole world inside. This is safe as long as the context does not change too much. In the field of path semantics there is no goal of modelling the complexity of the world, but to create algorithms that can find paths of functions. It is a much narrower goal than full common sense modelling.