Subtyping and Dynamic Dispatch
Or why apples are functions

Marc Sunet*

July 6, 2012

Introduction

Subtyping and dynamic dispatch… you don't really need them. Or well, maybe you do, but perhaps less often than you think.

The purpose of this tutorial is to convince ourselves that we can dodge subtyping and dynamic dispatch using closures. As a Haskell initiate and as someone who had their mind shaped in the object-oriented style, designing solutions in an environment where we can't simply roll a class hierarchy made me feel naked most of the time. This tutorial addresses that particular problem in the context of building a simple game. The information presented here is not new, but the average Joe the C++ coder should still find it enlightening.

An Initial Game Object

Suppose we are building a game. It is quite natural to think about the objects making up a scene as just that, game *objects*. Now, suppose that our game has apples and bananas. Of course, we do not want to limit ourselves to just these two types of objects, as we might decide to add new ones later on.

The natural thing to do in the object oriented paradigm would be to throw in a base class, say, `GameObject`, from which other classes, say `Apple` and `Banana`, would derive. This would allow us to decouple the rest of the game code from the specific types of games objects, giving us the ability to add new ones at a later stage. The following code snippet illustrates this concept:

*My thanks go to Tilo Wiklund for the initial explanation of the concepts leading up to writing this document, as well as for his help in proofreading, copy editing and typesetting it.

```
class GameObject {
public:
    virtual void render () const = 0;
};


class Apple : public GameObject {
public:
    void render () const { cout << "apple" << endl; }
};


class Banana : public GameObject {
public:
    void render () const { cout << "banana" << endl; }
};
```

In Haskell, one might be tempted to use algebraic data types as shown below:

```
data GameObject = Apple | Banana

render Apple = putStrLn "apple"
render Banana = putStrLn "banana"
```

But this approach would not be too adequate, the reason being that the C++ version of the code has the added benefit of giving one the ability to alter the behaviour of the `GameObject` class by extending it with new types of objects without touching existing code. In other words, the `GameObject` class satisfies what is known as the *open/closed principle*, which states that classes should be open for extension, but closed for modification. Unfortunately, this principle does not directly translate to ADTs; we can not add an `Orange` constructor without modifying the existing code, including many of the functions that pattern match on `GameObjects`.

No Data? That must be Functional!

Notice that `GameObject`, `Apple` and `Banana` have no *data*, only *behaviour*. What we just coded in C++ is a bunch of function objects; that is to say simply functions disguised as objects. This is in fact a common C++ idiom. It allows one to pass functions around and swap one for the other as if they were first class citizens to get different behaviours at runtime.

Since our C++ code is actually functional in nature, we ought to be able to do a Haskell "equivalent". To do so, we think about the behaviour exposed by the `GameObject` class. A game object is simply something that can render itself, so by capturing this very notion we get to the following Haskell alternative:

```
data GameObject = GameObject { render :: IO () }

apple  = GameObject $ putStrLn "apple"
banana = GameObject $ putStrLn "banana"
```

How is this better than what we had initially? Well, we can now add oranges without modifying existing code:

```
orange = GameObject $ putStrLn "orange"
```

The open/closed principle is preserved in the new Haskell design. We just had to realise that the C++ class hierarchy we devised was inherently functional to arrive at a better Haskell solution.

Notice that in the Haskell version of the code apples and bananas are not proper types, but functions that construct a `GameObject` value. This is a key point that will accompany us throughout the rest of the tutorial.

Identifying Game Objects with Unique IDs

Suppose that we want to go further by identifying game objects with a unique ID and spit it when rendering an object. The C++ code could resemble the following:

```
class GameObject {
    int _id;

public:
    GameObject (int id) : _id(id) {}

    int id () const { return _id; }

    virtual void render () const = 0;
};


class Apple : public GameObject {
public:
    Apple (int id) : GameObject(id) {}

    void render () const { cout << "apple " << id() << endl; }
};


class Banana : public GameObject {
public:
```

```
    Banana (int id) : GameObject(id) {}

    void render () const { cout << "banana " << id() << endl; }
};
```

But how do we translate that into Haskell?

Same Data, Different Behaviour

The ID is common to *all* game objects. Our old design adapts to the new situation without any major changes; all we need to do is incorporate the new attribute into the `GameObject` data type:

```
data GameObject = GameObject
    { goID  :: Int
    , render :: IO () }

apple goid = GameObject goid $ putStrLn "apple"
banana goid = GameObject goid $ putStrLn "banana"
```

Adding data common to all game objects is not a problem, we just need to slightly complicate the `GameObject` data type so as to reflect the changes. Notice that since `apple` and `banana` are functions that build `GameObject` values, we must supply them with the game object ID that identifies the game object we are building.

Updating Game Objects

A static game is not all that fun, and eventually we will want to update game objects over the course time. Let us suppose that our apples now have a level attribute, and that it is included in the output when apples are rendered. Furthermore, suppose that they level up on every game tick. Note that we wish our bananas remain static. Of course, having apples level up on every tick might seem like a somewhat contrived example, but it illustrates the next problem at hand without adding too much clutter to the code.

The C++ version of the game could now look something like the following:

```
class GameObject {
    int _id;

public:
    GameObject (int id) : _id(id) {}
```

```
    int id () const { return _id; }

    virtual void render () const = 0;

    virtual void update () {} // Do nothing.
};


class Apple : public GameObject {
    int level;

public:
    Apple (int id) : GameObject(id), level(0) {}

    void render () const { cout << "apple, id: " << id()
                                << ", level: " << level
                                << endl; }

    void update () { level++; }
};


class Banana : public GameObject {
public:
    Banana (int id) : GameObject(id) {}

    void render () const { cout << "banana " << id() << endl; }

    // Default update version inherited from GameObject.
};
```

How do we reflect the changes in Haskell? One might be tempted to do the following:

```
data GameObject = GameObject
    { level :: Int
    , render :: IO ()
    , ... }
```

Recall that bananas do not have levels. We can not push the level attribute all the way to `GameObject` itself, as that would imply that *all* game objects have a level.

Ok then, what about the following:

```
data GameObject = Apple { level :: Int, ... } | Banana { ... }
```

This would once again remove the possibility of adding new kinds of `GameObjects`.

While a `GameObject` is still just something that can render, update, and uniquely identify itself, an apple needs to carry around an additional `level` attribute.

Closures to the Rescue

Sometimes we just need to put the functional hat back on, and this is one of those times. If you are doing object oriented programming most of the time, you might forget about the often underestimated power of *functions*.

The `level` attribute is used internally by our apples, so why not trap it in a closure? With the data centric hat on, we could say that an Apple is a type of GameObject that has a level, that it levels up every certain amount of time, and that as with all game objects, it can render and uniquely identify itself.

Of course, if we are for the most part object oriented programmers we are going to be crunching data every minute, so that definition might seem just fine. In Haskell, however, it is not.

We have to question ourselves what it is we wish to think of as an Apple. If we take the data centric hat off and put the functional one on, we might come up with the following:

> An Apple is a function that takes a game object ID and a level and returns a GameObject identified by the given ID that renders the given level to the output and whose update function returns a new GameObject that does exactly the same thing but with the given level increased by 1.

What do we mean by this? We mean something like the following:

```
data GameObject = GameObject
    { goID  :: Int
    , render :: IO ()
    , update :: GameObject }

banana :: Int → GameObject
banana goid =
    GameObject
    { goID  = goid
    , render = printf "banana %d\n" goid
    , update = banana goid }

apple :: Int → Int → GameObject
apple level goid =
    GameObject
    { goID  = goid
    , render = printf "apple, id: %d, level %d\n" goid level
    , update = apple (level+1) goid }
```

As we have been pointing out, apples are not game objects, but *functions* that assemble game

objects for us. The key here is that what defines an apple is a function applied to whatever attributes we need to represent that apple. Technically speaking, we would say that an apple is actually a *closure*, not the apple-building function on its own, but we can still think of an apple as a way to assemble a game object.

In Haskell, functions are not just pieces of code that take some parameters and return a value. Functions are first class citizens; you can pass them around and return them from other functions, store them, replace them. They allow one to encode ”regular” functions, data structures, infinite lists, and more.

Above, we use functions to represent particular types of game objects. The only data type we actually have is `GameObject`. Particular game objects, such as apples and bananas, are not *subtypes* of a more general `GameObject` type, but *values* of that type.

Now, the terminology can get a bit confusing. When we say ”an apple is a function that assembles a `GameObject`” and ”an apple is a `GameObject`”, we really mean the same thing. The point is that apples are not subtypes of `GameObject`, but values of it. From now on we will apply both perspectives interchangeably.

But something is still not quite right. Aren't apples (and bananas) infinitely recursive? Recall:

```
apple :: Int → Int → GameObject
apple level goid =
    GameObject
    { ...
    , update = apple (level+1) goid }
```

The `GameObject` assembled by `apple` has an apple inside of it with its level increased by 1, and that other apple has another apple inside with the original level + 2, which in turn has another apple that contains another apple...

Indeed, we are building an infinite sequence of apples. Haskell is lazy, so the recursive `apple` call is not evaluated until it is requested. We can build an infinite game object like above just like we could encode the *complete* fibonacci sequence in an infinite list:

```
  fib = 0 : 1 : zipWith (+) fib (tail fib)
```

Laziness gives us the power to deal with infinity.

Towards More Realistic Updates

Up untill now we have had apples level up on every game tick. This has the possibly undesirable effect that apples will level up at different rates depending on available computation resources.

What we seek now is the ability to level up apples based on some time measurement.

Notice that the Haskell code is building a static sequence of game objects. In the case of bananas, this is a sequence of identical bananas, while in the case of apples, each new apple in the sequence gains an additional level with respect to the previous one.

The update function is not reacting to any external input, such as time; it just spawns a static sequence of game objects. What we would like is to generate a *dynamic* sequence of game objects based on some external measurement of time.

Going back to C++, we could let the `update` method take a time delta and let apples level up every 5 seconds:

```
class GameObject {
    int _id;

public:
    GameObject (int id) : _id(id) {}

    int id () const { return _id; }

    virtual void render () const = 0;

    virtual void update (float dt) {} // Do nothing.
};


class Apple : public GameObject {
    int  level;
    float elapsed;

public:
    Apple (int id) : GameObject(id), level(0), elapsed(0) {}

    void render () const { cout << "apple, id: " << id()
                                << ", level: " << level
                                << endl; }

    void update (float dt) {
        elapsed += dt;
        if (elapsed >= 5.0f) {
            elapsed = elapsed - 5.0f;
            level++;
        }
    }
};


class Banana : public GameObject {
public:
    Banana (int id) : GameObject(id) {}
```

```
    void render () const { cout << "banana "<< id() << endl; }

    // Default update version inherited from GameObject.
}
```

Now, how should we go about encoding this behaviour in Haskell? We wish to make use of the lesson we just learned. Apples and bananas are just `GameObjects`, or functions that assemble `GameObjects` for us. So how can we adapt our earlier game object to fit these new requirements?

A game object is still something that can be rendered, updated, and uniquely identified. The definition from last time suits us quite well, except that the `update` function now takes a time delta, just as in the C++ version:

```
data GameObject = GameObject
    { goID  :: Int
    , render :: IO ()
    , update :: Float → GameObject }
```

The definition of banana only varies slightly. Its `update` function ignores the given time delta and returns the original object:

```
banana :: Int → GameObject
banana goid =
    GameObject
    { goID  = goid
    , render = printf "banana %d\n" goid
    , update = const $ banana goid }
```

We now turn to apples. The previous lesson taught us that the `level` attribute could simply be a function argument. We now do the same with the `elapsed` attribute and add that extra logic to get the apples leveled up every 5 seconds:

```
apple :: Int → Float → Int → GameObject
apple level elapsed goid =
    GameObject
    { goID  = goid
    , render = printf "apple, id: %d, level %d\n" goid level
    , update = \dt →
        let elapsed' = elapsed + dt
        in if elapsed' >= 5.0
            then apple (level+1) (elapsed' - 5.0) goid
            else apple level     elapsed'         goid }
```

Now an apple's update function creates a dynamic sequence of apples by reacting to time. In fact, it could react to any external event, such as user input or a network message.

Now take a look at that! The Haskell solution is just as extensible as the C++ version, but without the need of subtyping and dynamic dispatch.

The full source code follows:

```
import Text.Printf

data GameObject = GameObject
    { goID  :: Int
    , render :: IO ()
    , update :: Float → GameObject }

banana :: Int → GameObject
banana goid =
    GameObject
    { goID  = goid
    , render = printf "banana %d\n" goid
    , update = const $ banana goid }

apple :: Int → Float → Int → GameObject
apple level elapsed goid =
    GameObject
    { goID  = goid
    , render = printf "apple, id: %d, level %d\n" goid level
    , update = \dt →
       let elapsed' = elapsed + dt
       in if elapsed' >= 5.0
           then apple (level+1) (elapsed' - 5.0) goid
           else apple level     elapsed'        goid }
```

Reifying Apple

As a final touch, having an actual `Apple` data type will make our code clearer and will allow us to write functions that operate on apples as apples, rather than apples as a kind of game object. The `Apple` data type could look something like the following:

```
data Apple = Apple
    { level   :: Int
    , elapsed :: Float }
```

The point being that it captures all the data needed to represent an apple as something other than a game object. Given this way to represent apples, we can make functions that act directly

on the `Apple` data type. A function responsible for the updating of apples could now be of the form:

```
updateApple :: Apple → Float → Apple
updateApple (Apple goid level elapsed) dt =
    let elapsed' = elapsed + dt
    in if elapsed' >= 5.0
        then Apple goid (level+1) (elapsed' - 5.0)
        else Apple goid level    elapsed'
```

Finally, we can change the apple builder to act directly on an `Apple` and make use of the apple updater we just defined:

```
apple :: Apple → Int → GameObject
apple a@(Apple level elapsed) goid =
    GameObject
    { goID  = goid
    , render = printf "apple, id: %d, level %d\n" goid level
    , update = flip apple goid . updateApple a }
```

The apple builder now defines a kind of explicit upcast operation. Once again, the full source code:

```
import Text.Printf

data GameObject = GameObject
    { goID  :: Int
    , render :: IO ()
    , update :: Float → GameObject }

data Apple = Apple
    { level  :: Int
    , elapsed :: Float }

updateApple :: Apple → Float → Apple
updateApple (Apple level elapsed) dt =
    let elapsed' = elapsed + dt
    in if elapsed' >= 5.0
        then Apple (level+1) (elapsed' - 5.0)
        else Apple level    elapsed'

apple :: Apple → Int → GameObject
apple a@(Apple level elapsed) goid =
    GameObject
    { goID  = goid
    , render = printf "apple, id: %d, level %d\n" goid level
    , update = flip apple goid . updateApple a }
```

```
banana :: Int → GameObject
banana goid =
    GameObject
    { goID  = goid
    , render = printf "banana %d\n" goid
    , update = const $ banana goid }
```

The Lesson We Just Learned

Many times, we might feel tempted to think of new entities as subtypes of a more general type. Using functions, we can encode these new entities as closures and provide a mechanism to build values of that more general type.

When we are doing functional programming we have to stop thinking of functions as just a means of computation. Functions in a functional language are much more than that. Of course, doing so is especially hard when we are constantly going back and forth from an object-oriented language to a functional one, but putting that extra effort will pay us back. Think twice before underestimating the power of such a tool, however primitive it may look. Functions will often go all the way.