Leveraging Familiar Game Concepts to Create a Game for Teaching Functional Programming

Implementation Details

Jesse Linossier
Faculty of Information Technology
Monash University
Clayton, Australia
jlin0076@student.monash.edu
ORCID:0000-0001-6782-7019

Tim Dwyer
Faculty of Information Technology
Monash University
Clayton, Australia
tim.dwyer@monash.edu

Shuai Fu
Faculty of Information Technology
Monash University
Clayton, Australia
shuai.fu@monash.edu

1 PURPOSE OF THIS DOCUMENT

This document provides a brief delve into the internals of the Haskell [1] style type system embedded in the C++ [2] programmed game; Looking at the major hurdles to development, as well as the design of the classes and structures used to be able to fully represent a Haskell type in C++ data. A brief overview of this document is already supplied as part of the main thesis as part of Section (3.2) - Type System Basics.

2 IMPLEMENTATION DETAILS

2.1 The difficulties of Haskell in C

Most Haskell code is not valid C++ code [3]. In fact most Haskell constructs are quite difficult to properly represent in C++. Take the type signature of fmap for example:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Listing 1: fmap

There are many incomplete or incorrect ways you could try to represent this type signature in C++. One thing you might think about is using interfaces for the `Functor` Typeclass.

```
template <class a, class b>
IFunctor fmap(b (*function)(a), IFunctor functor);
```

Listing 2: IFunctor

However, this is missing critical information. The above type signature does not specify what type of value is inside the `Functor`. Let's try making IFunctor a templated interface.

```
template <class a, class b>
IFunctor<b> fmap(b (*function)(a), IFunctor<a> functor);
```

Listing 3: IFunctor<b>

Unfortunately this type is too narrow. In Haskell many things can be a `Functor`, even if they don't have a single contained type. For example, functions are a `Functor`.

```
fmap :: (a -> b) -> ((->) c) a -> ((->) c) b
fmap :: (a -> b) -> (c -> a) -> (c -> b)
```

Listing 4: fmap for functions

As we can see, here the `Functor` is `((->) c)`. We cannot represent this with the IFunctor interface as 'c' could be any type including another function, and so we would need an infinite number of instances of IFunctor for each type 'c' could be. This is not possible. You might think that we could swap out the IFunctor interface for a template template as such:

```
template <class a, class b, template <class> typename F>
F<b> fmap(b (*function)(a), F<a> functor);
```

Listing 5: IFunctor as F<b>

However this won't work for three different reasons:

(1) Haskell allows functions to be curried, that is have one input given at a time, returning a new function at each step. Assuming we have a currying function, with this type signature we would have to be able to provide resolution for all of the templates at first call. This goes against currying though as a partially applied fmap would only ever be able to operate on a single known type of `Functor`, even though the `Functor` is the second argument. In Haskell it would be fine to use the same partially applied fmap on two different `Functor` types.
(2) We are assuming we know the concrete type of `Functor` used, but this assumption is not a valid one. In Haskell types are deferred and only resolved as much as needed. For example `fmap (+1) Nothing` is a function call of fmap that operates over a `Maybe a`, but Haskell does not need to know what 'a' is yet while C++ would due to using templates.
(3) Templates are inherently unusable for this type of problem. What we want isn't a way of writing Haskell looking code in valid C++ and being able to compile it, we need a runtime system. In the game, function blocks can be connected and disconnected to any other function block on the canvas. Yet a template system would need to know all types at compile time. As function blocks can pass a runtime type around, it

is impossible to use compile time information to infer what the concrete type is.

As for the other obvious solution, spinning up a Haskell Read Evaluate Print Loop (REPL) in the background might seem appetising, but it still suffers from the same problems. We would have to be able to read out the types from the Haskell REPL output and convert them into concrete C++ types to make use of them. This is not possible unless we allow quite harsh type widening as the same function could return a function, a wrapped value, an empty wrapper, etc. and we are back to the type resolution issue.

Instead of dealing with all these issues with a templated system, this game uses three separate systems in tandem to represent Haskell types in C++. A system for defining Typeclass implementations for Dataclasses, a system for erasing and clarifying types one layer at a time, and a runtime type system.

2.2 The Typeclass System

To implement the function definitions required for a Typeclass, a Dataclass defines a series of singleton subclasses that inherit from Typeclasses interfaces, and the Dataclass provides a means of accessing them. What this means, is that if we have the Typeclass '`Functor`' and the Dataclass '`Maybe`', we start by defining the `Functor` Typeclass:

```
typedef const VStar& VStarRef;
// Functor Interface
class IFunctor {
public:
  const auto fmap() const {
    return curry([this](VStarRef f, VStarRef f_a ) -> VStar {
      return this->_fmap ( f, f_a );
    });
  };
private:
  virtual VStar _fmap (VStarRef f , VStarRef f_a ) const = 0;
};
```

Listing 6: IFunctor definition

'VStar' is discussed further in Section (2.3), but can be thought of as a type-safe 'Any' type for now. We can see that we define a curried version of 'fmap' that uses a private virtual uncurried version of 'fmap' that the subclasses can override to implement its functionality. We can then let the Typeclass system know about 'IFunctor':

```
// Typeclass Struct
struct Typeclass {
  const IFunctor* Functor;
};
// ITypeclass Interface
class ITypeclass {
private:
  virtual const Typeclass _GetTypeclass() const = 0;
public:
  const Typeclass GetTypeclass() const;
};
```

Listing 7: ITypeclass definition

Now we can define the 'Maybe' class and have it provide the 'ITypeclass' interface, and have a nested class provide the 'IFunctor' interface.

```
// The Maybe Class
class Maybe : public virtual ITypeclass {
private:
  // Class Data
  VStar _value;
  // Typeclass Information from ITypeclass Implementation
  virtual const Typeclass _GetTypeclass() const override {
    // We have a Functor Instance
    Typeclass Instances;
    Instances.Functor = &FunctorInst;
    return Instances;
  }
  Maybe(VStar InValue) : _value(InValue) {};
public:
  // Constructors
  Maybe Just(VStar a) { return Maybe(a) };
  Maybe Nothing() { return Maybe(VStar()); };
  // The Functor Nested Class
  class Functor : public virtual IFunctor {
    Functor() = default;
    virtual VStar _fmap(VStarRef f, VStarRef f_a) const override;
  }
  // The Functor Instance of the Maybe class
  inline static const Functor FunctorInst = {};
}
```

Listing 8: `Maybe` definition

Finally, we can implement the 'IFunctor' interface.

```
// (simplified) Fmap Implementation
inline VStar Maybe::Functor::_fmap(VStarRef f, VStarRef f_a) const {
  return f_a._value.Valid() ? f(f_a._value) : f_a;
}
```

Listing 9: `Maybe Functor` implementation

and use it

```
// Just 1 and Nothing
Maybe justOne = Maybe::Just(1);
Maybe nothing = Maybe::Nothing();
// Get the fmap implementation
auto fmap = justOne->GetTypeclass().Functor->fmap();
// fmap (+1) over each maybe
Maybe justTwo = fmap(plus_one)(justOne); // Just 2
Maybe stillNothing = fmap(plus_one)(nothing); // Nothing
```

Listing 10: Using Maybes with fmap

While some special syntax around the handling of the 'VStar' type and its creation through the 'curry' function has been omitted for brevity, this is otherwise entirely functional and complete code. VStar requires some special handling as VStar is the 'Stratified Type Erasure' system that allows types to be resolved or erased one layer at a time. This paper will omit the specific implementation details for brevity, and will instead focus on the use of the system, as this system is the backbone of this entire project, and what makes it possible.

2.3 Stratified Type Erasure

If we had a highly nested type, such as `example :: Maybe (Int -> Maybe Int)`, to represent this in C++ without VStar, we would need to write:

```
// Without VStar, the entire type must be known
TMaybe<TFunction<Int, TMaybe<Int>>> example = Maybe::Just([](Int x) { return Maybe::Just(x); });
TFunction<Int, TMaybe<Int>> f = fromJust()(example);
TMaybe<Int> maybeResult = f(Int(1));
Int result = fromMaybe()(Int(-1))(maybeResult);
```

Listing 11: Without VStar

However with VStar, we only have to be aware of as much type information as we need. We may only need some function, rather than knowing what types the function inputs and outputs, we may only need `Maybe` as a box rather than knowing what type is inside of it, we may even only need some `Functor` without caring what type it actually is; we only specify what we use and not what we have:

```
// With VStar, we can Wrap / Unwrap one layer at a time, and not know the full type
Maybe example = Maybe::Just([](Int x) { return Maybe::Just(x); });
Function f = *fromJust()(example).ResolveTo<Function>();
Maybe maybeResult = *f(Int(1)).ResolveTo<Maybe>();
Int result = *fromMaybe()(Int(-1))(maybeResult).ResolveTo<Int>();
```

Listing 12: Using VStar

In addition, VStar is type safe while still being Type Erased. If we try to 'ResolveTo' the wrong type, we get a nullptr, which we can easily check for. This has added benefits when we consider how this has to work in the Visual Programming Langauge (VPL)where types and values are runtime dependant.

REFERENCES

[1] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson *et al.*, "Report on the programming language haskell: a non-strict, purely functional language version 1.2," *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.

[2] B. Stroustrup, "A history of c++ 1979–1991," in *History of programming languages—II*, 1996, pp. 699–769.

[3] M. B. Francesco Mazzoli, "inline-c: Write haskell source files including c code inline. no ffi required." 2015. [Online]. Available: https://hackage.haskell.org/package/inline-c