APPROACHES TO EMBEDDABLE FOREIGN LANGUAGE INTERFACES

Author

Maxwell Swadling

Supervisor

Manuel M. T. Chakravarty

Assessor

Ben Lippmeier

Thesis submitted for the degree of
BE Software Engineering

THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

29th October 2013

Abstract

Working with different programming languages within the same program can be difficult, especially if they are to interact non-trivially. It takes a considerable amount of work to get these programs to function correctly, but lets us create programs easier than with just one of the programming languages. This thesis defines two approaches to making this interaction simple and powerful. By using one of these approaches, it is easier for us to make interesting programs with both programming languages. We specifically explore the case of extending Objective-C with Haskell.

Acknowledgements

I would like to thank Manuel M. T. Chakravarty, my supervisor, for all his help and guidance throughout the year.

I would also like to thank everyone in the PLS research group for their help and unique insights along the way.

I would like to thank my family and friends for their support this year.

Finally, I would like to thank Erika Degens, my girlfriend, for all her love and support.

Contents

1 Introduction

Building programs with multiple programming languages can let us more easily create more interesting programs than with just one programming language. Limitations of one programming language can be overcome by utilising another programming language. Sometimes, it may be a simpler to call a function from a *foreign function interface* (FFI) [6] than build an interface in the host programming language's abstractions. When creating a graphical user interface (GUI), it is often easier if we could write the GUI component in the GUI's programming language, but the non-GUI component in a different language more suited to the computation required.

It can be complex and cumbersome to build interfaces using just a foreign function interface. We will focus on using Haskell from C/Objective-C, but these approaches could be implemented for different languages. These approaches are useful in graphical user interfaces, robotics systems, distributed systems or similar problems when we can separate the components with different roles into modules and connect them via interfaces. The two languages may not be different programming languages either, but rather the programming language is used in two distinct ways, with different libraries or in conjunction with an *embedded domain specific language* (EDSL) [10]. An EDSL is a programming language that serves a narrow use case, such as an abstraction to a library, except is provided in the host programming language's abstractions. An example of a DSL is a database query language which executes a string, while an EDSL version would use Haskell's type system to type check the query language expressions.

We want to make the process simple for us to make more interesting programs, allowing us to write programs in both programming languages seamlessly. This simplicity then allows us to extend the host programming language with features of the foreign programming language. To do this, we propose two approaches to working with multiple programming languages and the infrastructure required to provide these tools. Specifically:

- An EDSL based approach to generating Objective-C code from Haskell.
- A new tool for evaluating compile time templates in C using a method similar to *quasi quotation* [14]. Quasi quotation lets us write programs that run at compile time, in any defined syntax. The quasi quoter expression is parsed at compile time and the resulting expression inlined into the program.
- A quasi quotation like approach to *foreign code embedding* in C and Objective-C. Foreign code embedding allows foreign code to be used inside the same files as the host code.

- A simplified build process for creating static libraries from the Glasgow Haskell Compiler (GHC) [7].

Finally, we compare these approaches with other approaches and define heuristics to help decide which approaches are well suited to developing Cocoa applications.

2 Problems and Approaches

First we must look at the motivation behind using these approaches and what problems they solve. Then we divide these problems into categories. Finally we describe categories of approaches (including existing approaches) for interacting between programming languages.

2.1 Motivation

Making programs in just one programming language is not always ideal. Say we want to make an OS X app to do edge detection on a camera feed. We want to utilise the features of Haskell for the image processing component of the program, but Objective-C for the user interface. A multi programming language approach is simpler because we are utilising the strengths of each programming language. If we wrote the whole program in Haskell, writing the user interface is difficult because of the mismatch between Apple's Cocoa framework and Haskell abstractions. Similarly, writing the whole program in Objective-C is difficult because reasoning about the correctness of the implementation is difficult compared to a Haskell implementation. So this dual programming language approach is ideal because we can use each programming language and associated runtimes for problems they are good at; Haskell to develop the multicore edge detection algorithm and Objective-C to develop the graphical user interface. However, when we need the two programming languages to interact it can become complicated.

The interaction between the two programming languages can be complex because of differences in the programming languages' semantics. Objective-C is an object-oriented programming language with extensive runtime type information and reflection [12]. The Cocoa framework makes use of runtime type information and reflection extensively to implement Key Value Observing (KVO)[1] and libraries such as Core Data (a database framework). Another design pattern utilised in Objective-C is evaluating strings at runtime to determine data or control, such as in NSPredicate (a query language) and Auto Layout's Visual Format Language (a GUI layout constraints language). Both these problems can be solved elegantly with a strongly typed functional programming language like Haskell.

For example, instead of using Core Data, we can use the Haskell postgresql-simple[2] library. This library makes use of quasi quotation to perform compile time checks on the SQL queries and generate query objects. Another example is instead of using Auto Layout strings or NSPredicates

[1] KVO [11] lets objects subscribe to changes in other objects and receives notifications when their values change.
[2] http://hackage.haskell.org/package/postgresql-simple

we can compile the strings with Haskell and generate a constant object representation of the string that has been checked to be correct. In both cases, we are rejecting more bad programs than before by adding compile time checks that were previously not possible.

Another motivation for using two programming languages is it may be difficult to represent a library in a foreign programming language in the host programming language. If we consider using Cocoa from Haskell, a library wrapping the Cocoa framework's functions would be large, complex and difficult to maintain and use.

Finally, we can benefit from using Objective-C in conjunction with a foreign programming language by using libraries that help deal with domain specific problems, such as ReactiveCocoa[3]. ReactiveCocoa provides us with an abstraction to simplify reasoning about control in Cocoa applications. By using Objective-C for the components ReactiveCocoa is powerful at representing, we can build on these abstractions instead of rewriting them in a foreign language.

If the approach can bring some of Haskell's type system features, such as parametric types, GADTs, type families, kinds, etc., to Objective-C we can reject more bad programs than previously possible. Even if we used Haskell through a foreign function interface, we can benefit from Haskell's type system.

2.2 Categorisation of Problems

To know if using multiple programming languages is a good approach for a program, we need some criteria to evaluate. One way to identify programs that can benefit is if the program has a function or callback which is where your other programming language should be called. For example, in the edge detection program the foreign function call is located in the function that processes a new image.

Another way to identify these kinds of problems is by structuring our programs such that it is intuitive where the two programming languages integrate. If we use a *design pattern* such as Model-View-ViewModel [9], the *models* should be the only classes where foreign code interfaces are used. We can simplify this further by creating a backend interface or superclass that the models subclass from, which provides wrapped function calls to the foreign programming language.

We also want to decide which of the programming languages is the foreign programming language and is responsible for exporting functions. Most approaches import C functions into Haskell. Language-c-inline [4] is a library that instead exports Haskell functions that are called

[3]https://github.com/ReactiveCocoa/ReactiveCocoa

from C by the foreign function calls into a template. The decision on which tactic to use will motivate which approach is best suited. For example, a program with the GUI component written in Objective-C would benefit from the language-c-inline approach because it allows us to use Haskell functions inside the GUI's functions and callbacks, while a program showing different GUI elements depending on Haskell computations will be better off with the FFI approach because it can create different GUI elements depending on Haskell.

2.3 Categories of Approaches

Currently there are several approaches available, each with their own advantages and disadvantages. By exploring these approaches and considering the motivations, we propose two new approaches that aim to compromise in such a way to optimise the benefits proposed in the motivations.

The foundation of interaction between two programming languages is the foreign function interface [6]. It allows functions to be imported into and exported from the foreign programming language (Haskell). This can be used as the simplest approach, but requires the most work to implement and provides us little additional semantics to help us write programs. Functions must only contain types available in the C language, making *marshalling* a required and verbose step in the FFI. Marshalling is the process of converting the foreign type into a host's type and back again. These interfaces can be tedious to test since they are often impure. In the example of the edge detection program, we want to call a Haskell function that will perform the edge detection computation on an image and give us back a processed image. To do this, we take the following steps:

1. A function to wrap the Haskell function that marshals C types into Haskell types (not required if only C types are used)
2. Export the wrapper function from Haskell to the FFI
3. Include the generated header in C
4. Call the function in C

This extra work making interfaces is tedious and error prone. The process should be as simple and automatic as possible. Changes to code used in other parts of the program should only require minimal manual refactoring of the interface. If possible, it should be done totally by code generation and not require manual refactoring.

We can categorise approaches for interacting between programming languages into 5 types:

2.3.1 Pre-processor Macros

The simplest approach is by using pre-processor macros to automate code generation for the interfaces. However, this approach does not work well for a few reasons:

- Can not write to other files. We could not perform code generation of interface files for another programming language since we can not write to other files.
- It is difficult to write powerful programs in macros.
- It can modify the C abstract syntax tree (AST) arbitrarily, potentially breaking scopes of functions. It is difficult to debug code that has been generated, so we want to minimise the scope code generation can affect.

For these reasons, macros are not considered an ideal approach.

2.3.2 Binding Libraries

A binding library provides an interface to the foreign programming language in an interface using the host programming language's abstractions. For example, Gtk2Hs[4] is one such implementation for providing GTK+ (a GUI library) bindings for Haskell.

Since we are writing the program in Haskell, we can type check the program in ways C can not, helping us reject more bad programs than if we used C.

A problem with this approach is it requires abstractions in Haskell for every API we want to use in GTK+. For GUI libraries, we must also decide how to implement those abstractions, such as with callbacks or event loops. While this is possible, it can be lots of work and design choices on how to represent some concepts will rule out some valid programs.

This may be infeasible for Cocoa due to the large code base and the frequency at which the API changes. On every iteration (often annually) the APIs may change and ongoing maintenance is required for the library. Ideally, we want to be able to define some rules on how the API is translated into an interface and automatically generate the interface, this approach is known as bridging libraries.

[4] http://hackage.haskell.org/package/gtk

2.3.3 Bridging Libraries

A bridging library provides an interface between the two programming languages usually by implementing the foreign programming language's features, not the library we want an to interface with. We define how the Objective-C programming language features are interfaced with. Then we can use the bridging library to work with foreign libraries in our host programming language. This way, we are defining an interface into the programming language and using that interface to work with foreign libraries.

This is simpler to implement than a binding library in principle because there is a small and finite amount of abstractions available to build a library with. Once all are implemented, any library could be used.

Another added feature of bridging libraries is we can define and work with objects from Objective-C in Haskell and call messages to them (call instance methods).

Four implementations of this for Objective-C are:

- HOC [15] which lets us write and use Objective-C objects from Haskell
- ObjectiveHaskell[5] which is like HOC except provides a facility to splice Haskell functions into Objective-C code
- RubyMotion[6] which is a similar concept except for Ruby. It bypasses a lot of the complexities of code generation by performing object and method interfacing at runtime.
- Wax[7] is another library that lets us write Objective-C programs in Lua. It uses the Objective-C runtime, like RubyMotion, to interoperate with Objective-C.

For convenience these libraries will often also wrap commonly used functions or classes like a binding library will to expose them with a simple API.

In terms of expressiveness, one of the drawbacks of using bridging libraries is it does not let us extend a library with Haskell language features to reject some bad programs. For example, if we wanted a monomorphic[8] array container, we can't express that in Objective-C. We can have an NSArray[9] object that contains NSString[10] objects, then add an NSTextField[11] to it. Then if we

[5]https://github.com/jspahrsummers/ObjectiveHaskell
[6]http://www.rubymotion.com
[7]https://github.com/probablycorey/wax/
[8]All elements in a monomorphic container have the same type.
[9]A `SrcLoc` is the location in the original source code this element occurred.
[10]The Cocoa string class.
[11]The Cocoa class for a text field in a GUI.

attempted to use all the array elements as strings, it would error at runtime when we tried to use the text field as a string. By using a bridging library, interface generator or language templating (as we will see soon) we could check the array only has strings and reject a program that tried to add an NSTextField to the array.

Another downside to this approach is the complexity introduced when interacting between the programming languages. It can be tedious to use the interface in Haskell because of the mismatch between the semantics of Objective-C and Haskell. Objective-C carries implicit information about how to call methods on objects. This makes defining methods on classes and call them verbose in Haskell, because we need to pack in this extra information.

2.3.4 Interface Generators

An Interface Generator is a tool that can generate interface code between languages. These tools make working with FFIs easier, however require some work by the developer to define the interfaces. This simplifies the process of using two programming languages together. Also this can be useful in creating binding libraries.

For example, hsc2hs [13] can be used to process C files to assist in generating interfaces with C functions and C structs. It does this by processing C headers and inserting generated interface code into chs files. A similar approach will be used in the cpppp approach.

c $\longrightarrow$ hs [3] is similar to hsc2hs, however adds several features, most importantly marshalling code generation for C to Haskell. Automated marshalling code generation simplifies the process and reduces the possibility of bugs, which are motivations. Because of this, cpppp can generate C and Objective-C marshalling code as well.

SWIG [1] is another interface generator that works on multiple languages to target C/C++. It enforces a design for the interface and design patterns for the developer to use. This is easier to use than c2hs given your interface following its conventions. However, we must use the interface as described by SWIG, with the SWIG API, which may not be ideal. SWIG's generated interfaces are also hard to debug because of the code generation.

2.3.5 Language Templating

The final approach is by providing code generation using *templating*. This lets us write programs in a programming language (potentially domain specific) that generates the code for the target pro-

gramming languages to work together. For example, an Objective-C file can be parsed, foreign function calls generated and inlined, then re-emitted to be compiled.

One implementation of this is language-c-inline. It parses a quasi quoted language-c-quote expression (Objective-C code), inlines Haskell foreign function calls that are spliced into the Objective-C code and emits Objective-C code with the foreign function interfaces and marshalling code generated and inlined.

The benefit of this approach is we can now use functions from Haskell inside Objective-C code. This is inline with our motivations. However, we can extend this idea in two different ways, describing two new approaches.

We can extend this idea by generating code to inline into the Objective-C code before language-c-inline runs. We can add additional types and other properties to the program before code generation, check they are consistent, then generate code which can not express those properties. We use an EDSL to write the more expressive code before generating Objective-C, so this approach is named the EDSL approach.

We can also extend this idea by using Objective-C as the host programming language instead of Haskell. This approach generates Haskell foreign function interfaces from Objective-C. We can use this approach to automate marshalling and generate foreign function interfaces as they are needed.

3 The EDSL Approach

The first approach we define is derived from extending the language-c-inline idea with an EDSL. Using an EDSL allows us to extend the foreign language. Recall, Objective-C does not have type parameters. We can add type parameters to the embedded domain specific language and type check them before they are discarded and the Objective-C code is generated. This allows us to reject more bad programs than previously possible.

To do this we define a simple abstraction on top of only Objective-C's objects abstraction. We then add semantics where we are interested in adding value to our programs. We use an embedded domain specific language (giving us the name of the approach) in Haskell to make it easy for us to add semantics by using the Haskell type system.

One approach is to define a new kind for all foreign data types. Kinds are sets of data types. By taking this step, foreign data types can not be mixed with Haskell's data types[12]. We call this kind `Ty`.

```
data Ty = NSBool | NSInteger | NSDouble | NSString
        | (~>) Ty Ty -- the function type
        | NSUnit -- C's "void"
        -- CG
        | CGRect
        -- AppKit
        | NSTextField
        | NSScrollView
```

Then we define a *generalised abstract data type* (GADT) to construct new values. This data type is indexed by a `Ty` kind. This lets us construct values in our foreign data type from Haskell values. A possible definition for this would be[13]:

[12] We make use of the GHC extensions `DataKinds` and `KindSignatures` to define the GADT.
[13] `CG` types and functions are geometry related. `AppKit` is for Cocoa GUI related objects.

```
data Func (a :: Ty) where
  FBool :: Bool -> Func NSBool
  FInteger :: Int -> Func NSInteger
  FString :: String -> Func NSString
  FUnit :: Func NSUnit

  FBlockItem :: BlockItem -> Func a
  FFunction :: Exp -> Func (a ~> b)
  FFunctionE :: Exp -> Func a -> Func a

  -- CG
  FCGRect :: (Int, Int, Int, Int) -> Func CGRect
  CGRectZero :: Func CGRect

  -- AppKit
  FNSTextField :: Func NSTextField
  FNSScrollView :: Func NSScrollView
```

For example, a boolean in Haskell is put into the foreign environment by using the `FBool` constructor. It gives us a type `Func NSBool`. The `Func` data type is not limited to functions, as we have seen with bools. However, it is primarily used to wrap foreign functions into Haskell values. For example, the `FBlockItem` constructor creates a function value from any block of C code. For example, if we have a some code that initialises an NSTextField, we wrap the code into a `FBlockItem` and set it's return type to `NSTextField`. We can not verify how the C code is implemented, but if it was identified with the correct type, then it can not be misused.

The two most useful constructors in `Func` are the `FFunction` and `FFunctionE` constructors. They are short for foreign function and foreign function expression. `FFunction` allows us to pack a C expression into a function from `a` to `b` (using our special function type constructor `~>`). `FFunctionE` lets us apply one C expression to a `Func` and gives us back a new `Func` of the same type, but with the C expression applied. We use these two constructors to represent most functions.

Programs are made up of bindings, so we define a `Bind` type. A `Bind` is defined as a GADT that has a location (local or global), bindability (if the expression result can be bound to a variable)

and a type. A `NoBind` is a statement like expression which does not have a return value. We define this with:

```
-- kinds
data TypeListType = Local | Global
data TBool = TTrue | TFalse

data Bind :: TypeListType -> TBool -> Ty -> * where
  Bind :: String -> Func a -> Bind t TTrue a
  NoBind :: Func (a ~> b) -> Bind Local TFalse NSUnit
```

Next, we define a data type for a program (namely `OM`) and its instance of monad. A program has two lists of bindings. One list of bindings is for local variables, the other is for class instance variables. When we perform code generation, we insert the instance variables into the interface of the Objective-C class. We name the monad `OM` and add a function to run it that produces two lists of values by applying functions to the local and global binding lists.

```
data OM a = OM a (TypeList Local) (TypeList Global)

instance Monad OM where
  return x = OM x typeListNil typeListNil
  (OM x ls gs) >>= f = let (OM x' ls' gs') = f x
                       in OM x'
                           (typeListAppend ls ls')
                           (typeListAppend gs gs')

runOM :: OM t
    -> (forall (a :: Ty) (n :: TBool) . Bind Local n a -> [t1])
    -> (forall (a :: Ty) (n :: TBool) . Bind Global n a -> [t2])
    -> ([t1], [t2])
runOM (OM _ ls gs) f g = let
  ls' = typeListMap f ls
  gs' = typeListMap g gs
  in (concat ls', concat gs')
```

A bindings list is parameterised over what types of bindings it may contain. This is defined in the data family for a `TypeList`. Since a global binding must result in a named variable (so it is possible to insert into the class interface), the `Bind` inside the TypeList is restricted to been a `Bind Global TTrue a`. In local bindings, it is a `Bind Local n a`, so it can either be a named variable or just a statement. For example, we can define `Bind` and the `Local` version of TypeList can be defined as:

```
data family TypeList :: TypeListType -> *
data instance TypeList Local
  = TypeListNilL
  | forall (a :: Ty) (n :: TBool) . TypeListConsL
                                      (Bind Local n a)
                                      (TypeList Local)
```

For the implementation of the type list functions and code generation backend see the reference implementation.

Using these data types we can write programs in this EDSL. After GHC type checks the EDSL, we can be sure the types are consistent in the program. We then perform code generation which can not represent anything outside of Objective-C, so some types are erased. However, this erasure does not affect the correctness of the program.

We then add functions to add bindings to `OM` for the different functions and *methods* in Objective-C. A method we represent as a function where one of the arguments is `self`. For example, say we want to make a new NSNumber object from an integer, we would define a function[14]:

```
newInteger :: Int -> OM (Bind Local TTrue NSInteger)
newInteger x = mkOM $ Bind (unsafePerformIO fresh) (FInteger x)
```

The `mkOM` function adds a new binding to the type environment. We generate a fresh[15] variable name and store the Int in the `FInteger` data type. When code generation occurs, the backend pattern matches on the binding to find the correct C code template to use. We use the language-c-quote quasi quoters `citem` and `cexp` to help us generate C code. For example, we can generate C code from local bindings mostly by using the quasi quoters.

[14] This code could be refactored to avoid the use of `unsafePerformIO`, but has not to keep the type obvious.
[15] Fresh variables are unique variable names.

```
mkBlocks :: Bind Local n a -> [BlockItem]
mkBlocks (Bind x (FBool True))  = [[citem|
                          typename NSNumber *$id:x = @(YES);|]]
mkBlocks (Bind x (FBool False)) = [[citem|
                          typename NSNumber *$id:x = @(NO);|]]
mkBlocks (Bind x (FInteger b))  = [[citem|
                          typename NSNumber *$id:x = @($b);|]]
mkBlocks (Bind x (FArray))      = [[citem|
                          typename NSArray *$id:x = [[NSArray alloc] init];|]]
mkBlocks (Bind x (FFunction s)) = [[citem|$id:x = $s;|]]
mkBlocks (NoBind (FFunction s)) = [BlockStm (Exp (Just s) noLoc)]
```

Notice how we destruct the `FBool` type and generate either `@(YES)` or `@(NO)` depending on which Haskell value it was constructed with. Another example is, say we wanted two `NSNumbers` with the values 4 and 5. The code generator pattern matches on the `(Bind x (FInteger b)))` case and returns a C `BlockItem` with the assignment. For example:

```
*ghci> let (block, properties) = codeGen (newInteger 4 >> newInteger 5)
*ghci> putStrLn $ prettyPragma 80 (ppr block)
{
    NSNumber* f_7 = @(4);
    NSNumber* f_8 = @(5);
}
```

Another example of how to define functions, say we wanted to define is `assign`, a function that assigns one variable to another. `assign` does not return anything, so we can not bind a variable to its result. So the `mkOM` function will give us a type error. Instead we need to use `addOM`, which only lets us add statements without changing the type environment. The function could be defined as:

```
assign :: TypeLC s1 => TypeLC s2 => Bind s1 TTrue t -> Bind s2 TTrue t -> OM ()
assign x y = addOM (NoBind (FFunction [cexp|$id:x = $id:y|]))
```

3.1 Type System

Using these semantics and Haskell as the EDSL host, Haskell provides us with type inference. We do not need to declare the types of variables if Haskell can determine them by their usage. Some types have default values (such as NSArray starts empty). We can use this property of some types to provide a type class for default values.

For example, if we have an integer `x` and assign it to another integer `y`, Haskell can determine the other variable's type is the same (because our definition of `assign` requires both arguments to be the same type).

```
x <- newInteger 4
y <- new
y `assign` x
```

This will compile and use `NSInteger`'s `DefTypes` instance to initialise the variable. To add this functionality, we define a type class for `DefTypes`:

```
class DefTypes (a :: Ty) where
  defaultType :: (Func a, Exp)
```

Here the first field of the tuple is the default value. This is also used for pattern matching on a constructor when performing code generation. The second first is the expression used to initialise objects of that type. For example, an instance for an `NSTextField` would be:

```
instance DefTypes NSTextField where
  defaultType = (FNSTextField, [cexp|[[NSTextField alloc] init]|])
```

If we want to allocate a new binding locally, we do not need to use a specific function (like `newInteger`) if the type has an instance for `DefTypes`. Also if we want to make a property (instance variable) in the class of that type, we can use the same type class. We can provide functions to generalise `new` and `newProp` as follows:

```
new :: DefTypes t => OM (Bind Local TTrue t)
new = mkOM $ Bind (unsafePerformIO fresh) (fst defaultType)

newProp :: DefTypes t => OM (Bind Global TTrue t)
newProp = do
  let n = unsafePerformIO fresh
      (dty, din) = defaultType
  addOM (NoBind (FFunction [cexp|self.$id:n = $din|]))
  mkOM $ Bind n dty
```

Recall our example:

```
x <- newInteger 4
y <- new
y `assign` x
```

If we wanted to move the variable `y` into a property (instance variable), we simply change `new` to `newProp` and it will use the `DefTypes` class to determine the type to declare in the interface.

Furthermore, in Objective-C there is a polymorphic `description` method that will give you the description of any object. The type of that function is polymorphic as expected.

```
printDescription :: TypeLC t => Bind t TTrue d -> OM ()
printDescription s = addOM (NoBind (FFunction [cexp|NSLog("%@", $id:s)|]))
```

3.2 Extending the Language

Recall we have only defined types with no type parameters. We want to extend this idea to be able to type check programs we can not express in Objective-C, such as types with type parameters. One such type is the monomorphic arrays, an array that can only contain elements of one type. To do this, first we add the data type to the language:

```
data Ty = ... | NSArray Ty


data Func (a :: Ty) where
...
  FArray :: Func (NSArray a)
```

Here we parameterise the type of an NSArray with a `Ty`. Now we add some new functions to our EDSL to work with NSArrays. First we want a function to take the first element in an array:

```
arrayHead :: TypeLC t
          => DefTypes d
          => Bind t TTrue (NSArray d)
          -> OM (Bind Local TTrue d)
arrayHead xs = let n = unsafePerformIO fresh
  in mkOM $ Bind n
          $ FFunctionE [cexp|[$id:xs firstObject]|]
          $ fst defaultType
```

The C expression to get the first element in an NSArray is `[x firstObject]`. So we wrap this C code into an `FFunctionE`. But we must also provide a constructor for `FFunctionE` to lookup the type of the return value. this is where we lookup the type representation in our

Notice how the type of the function argument is an `NSArray d` and returns a binding to a single `d` object. This is not possible conventionally in Objective-C.

The next useful thing to do with arrays is to add objects to them. We define the following function for that purpose.

```
addObject :: TypeLC t1
          => TypeLC t2
          => Bind t1 TTrue d
          -> Bind t2 TTrue (NSArray d)
          -> OM ()
addObject x xs = addOM
               $ NoBind
               $ FFunction [cexp|[$id:xs addObject:$id:x]|]
```

This is similar to `arrayHead` except it has no return value. Since we do not need to define a type to return into, we use `FFunction` instead of `FFunctionE`. Then we use the language-c-quote quasi quoter to generate code to call `addObject` on the array's identifier with the new object's identifier.

Using these functions, we can demonstrate using type inference to infer the type of the array. For example:

```
example :: OM ()
example = do
  xs <- new
  x <- new :: OM (Bind Local TTrue NSInteger)
  addObject x xs
  h <- arrayHead xs
  printDescription h
```

`xs` is determined to be an `NSArray` because it is used as the argument to `addObject`. `x` is a local variable we say is an NSInteger. We are required to fix the type of `x`. If we left the type annotation off `x` then it would be a type error, since the type of `NSArray t` is not known. Also if we tried to insert an `NSInteger` then an `NSBool` or `NSString`, it would be a type error.

3.3 Extending with Libraries

We can extend our EDSL with other libraries. One such library we could add is the Reactive-Cocoa library. ReactiveCocoa is a functional reactive programming library for Objective-C. In functional reactive programming changes in values over time are represented as *signals* of values. An example signal could give you a text field's string value which changes as a user types into the field. However, since ReactiveCocoa is implemented in Objective-C, signals are not parametric over a type. Instead, all signals are of type RACSignal regardless of the types they hold. To help us write programs that use ReactiveCocoa, we can use our EDSL to add a type parameter to the RACSignal type. Then we can know by type checking the EDSL that we have not mixed up RACSignals of different types. The first step to adding this is adding types for signals and tuples:

```
data Ty =
...
  -- RAC
  | RACSignal Ty
  | RACTuple Ty Ty

data Func (a :: Ty) where
...
  FRACSignal :: Func (RACSignal a)

  FRACTuple :: Func (RACSignal a)
            -> Func (RACSignal b)
            -> Func (RACTuple (RACSignal a) (RACSignal b))

  FRACFunctionT :: Exp
                -> Func (RACTuple (RACSignal a) (RACSignal b))
                -> Func (RACTuple (RACSignal a) (RACSignal b))
```

Here we define two new types in the kind `Ty`: `RACSignal Ty` and `RACTuple Ty Ty`. Then we add constructors to the `Func` data type to create new objects using them. Our first is `FRACSignal`, which can be used to introduce new signals. The next is `FRACTuple`, which wraps up two signals into a tuple. The last is `FRACFunctionT` which we recall is similar to our `FFunctionE` type from before except it is specialised to a `RACTuple`. The difference is, the one expression now has two bindings (the fields of the tuple).

Returning a `Tuple` to Objective-C is not trivial. To get around this a new type of binding is added that unpacks the tuples.[16]

Now we have defined the data types required, we can implement functions such as `rac_textSignal`, which create a signal of NSString values from a `NSTextField`.

[16] For implementation details, please see the reference implementation.

```
rac_textSignal :: TypeLC t
               => (Bind t TTrue NSTextField)
               -> OM (Bind Local TTrue (RACSignal NSString))
rac_textSignal x = mkOM $ Bind (unsafePerformIO fresh)
                        $ FFunctionE [cexp|[$id:x rac_textSignal]|] FRACSignal
```

Note the return type is `RACSignal NSString`. If this binding was used as an argument to a function that took a `RACSignal NSNumber` it would be a type error.

Finally we can use type classes to represent inheritance. We define the function `rcl_frameSignal` which returns a signal of `CGRects` as the frame of an NSView changes.

```
rcl_frameSignal :: TypeLC t
                => View v
                => Bind t TTrue v
                -> OM (Bind Local TTrue (RACSignal CGRect))
rcl_frameSignal x = mkOM $ Bind (unsafePerformIO fresh)
                         $ FFunctionE [cexp|[$id:x rcl_frameSignal]|]
                         (FRACSignal)
```

Since this only works on NSViews, we add the type constraint `View v`. If we used the type `NSView` instead, then using an `NSTextField` would be a type error. To define this, we add an empty type class and instances for Objective-C types that are subclasses. For example:

```
class View (a :: Ty) where
instance View NSView
instance View NSTextField
instance View NSScrollView
```

3.4 Extending with Context

Combining the tools we have developed, we can write full Mac OS X applications using this approach. Since we want to be able to use Haskell functions inside our Objective-C code we make use of the language-c-inline library. Since our code generator emits a C AST, we can

splice this into a language-c-inline template. For example, the Layout module in our example exports the `props` and `onLoad` functions that are the properties and C AST generated from code generation. We can inline them into an NSWindowController (a GUI window controller) template and compile it to produce an application.

```
{-# LANGUAGE TemplateHaskell, QuasiQuotes #-}
module AppWindow (objc_initialise) where

import Language.C.Quote.ObjC
import Language.C.Inline.ObjC

import Layout (props, onLoad)

objc_import [ "<Cocoa/Cocoa.h>"
            , "<AppKit/AppKit.h>"
            , "<Archimedes/Archimedes.h>"
            , "<ReactiveCocoa/ReactiveCocoa.h>"
            , "<ReactiveCocoaLayout/ReactiveCocoaLayout.h>"]

objc_interface [cunit|
@interface AppWindow : NSWindowController
@end
|]

objc_implementation [] [cunit|

@interface AppWindow ()
$ifaces:props;
@end

- (void)windowDidLoad
{
  [super windowDidLoad];
  $stm:onLoad;
```

```
}

- (id)init {
  return [self initWithWindowNibName:@"AppWindow"];
}
|]

objc_emit
```

Furthermore, we can implement the ReactiveCocoaLayout[17] library (a GUI layout framework that uses ReactiveCocoa) in our EDSL and use it to write GUIs. Then we can use the EDSL instead of XIBs[18], letting us write both the GUI and code in the EDSL.

[17] https://github.com/ReactiveCocoa/ReactiveCocoaLayout
[18] Apple's interface layout file format.

4 The Cpppp Approach

The second approach is derived from bringing the ideas from language-c-quote [4] over to C. Cpppp is a simpler approach we will define soon that aims to offer an expressive abstraction without the complexity of DSL based code generation. Recall from our introduction, quasi quotation [14] is a feature of Haskell that allows us to write expressions in an arbitrary programming language that is parsed and executed at compile time, then the result inlined into the program to be compiled. This functionality would provide us with the infrastructure required to implement cpppp in the C programming language. However, quasi quotation provides features we do not need that add to the complexity of using quasi quotation. So we need to make a compromise between functionality and simplicity.

Using cpppp we can write foreign code inside of the host code. In our example, this allows us to write Haskell inside of C code and code generation will generate the FFI required code required. We do not need to decide which functions we want to export from the FFI ahead of time. But to define this approach, first we must extend the C language with a new feature, named *Quoted Initialisers*. While I provide examples for C, the framework could be converted to work in other programming languages.

4.1 Quoted Initialisers

Quoted initialisation is not unlike quasi quotation. However, quoted initialisers can only appear as the value in a typed *A-normal form* [8] expression. A-normal form is an assignment expression where all arguments are variables, not other expressions. In the C language, we restrict this to the initialisation value of a variable. An example of a quoted initialiser lets us initialise the value of the C int `z` by adding two C ints `x` and `y` with the Haskell function `(+)` is:

```
int z = @|c_hs|"(+) %d %d"|x, y|;
```

There is a lot of information baked into this expression that we will understand soon. To decide on why we use this representation, we first isolate some desirable properties of representing foreign code:

1. The quoted initialiser can not modify the AST outside of it. This simplifies the effect of code generation on the AST since it is contained, unlike a macro which can mutate the AST

in any fashion (for example, closing a function and opening a new function from inside of a loop is possible, we want to rule that possibility out).

2. The return type is defined. For marshalling purposes, we may want to know the type we are returning into. Say the Haskell function is returning an Double, it may be desirable to marshal it into a different number representation in Haskell before returning in.
3. All arguments must be evaluated trivially. We are guaranteed they are already values and their evaluation order has already been taken care of.

These properties can be provided by using an A-normal form expression.

The reference implementation of quoted initialisers was made in Haskell by using the language-c-quote library[19]. It performs two tasks: traversing the AST and performing *transforms*. When a `QuotedInitializer` is encountered in the AST, a transform is performed on it to transform it from a `QuotedInitializer` into an `ExpInitializer` (something that exists in standard C). The transform to use is based on the name used in the `QuotedInitializer`. Transformers are given an expression to evaluate and a list of ids of the arguments to splice in.

After the transforms are complete, the transformed C code is pretty printed into a file, ready to be compiled by a C compiler. This is where the name of the approach comes from, these transforms are applying pre-processing before the C pre-processor is ran. So it is a C pre-processor pre-processor (cpppp).

By observing the definition of the C AST in language-c-quote[20]:

```
data InitGroup = InitGroup DeclSpec [Attr] [Init] !SrcLoc
               | TypedefGroup DeclSpec [Attr] [Typedef] !SrcLoc

data BlockItem = BlockDecl InitGroup
               | BlockStm Stm
```

A function is made of a list of block items. Each block item may be a statement or declaration. If the declaration is an `InitGroup`, then it contains a list of `Inits`.

[19] http://hackage.haskell.org/package/language-c-quote
[20] A `SrcLoc` is the location in the original source code this element occurred.

```
data Init = Init Id Decl (Maybe AsmLabel) (Maybe Initializer) [Attr] !SrcLoc

data Initializer = ExpInitializer Exp !SrcLoc
                 | CompoundInitializer
                       [(Maybe Designation, Initializer)] !SrcLoc

data DeclSpec = DeclSpec [Storage] [TypeQual] TypeSpec !SrcLoc
```

Each `Init` may contain an initialiser. Recall the desired properties of quoted initialisers and it is clear the position of `Initializer` matches for the C programming language because it:

1. Will not allow mutation of the outside AST.
2. Has a defined return type (contained in the `InitGroup`'s `DeclSpec`, the `TypeSpec`).
3. Since the arguments can only be `Id`s, they are trivially evaluable (they can only be variables for values, not other expressions).

By adding `QuotedInitializer` to the `Initializer` data type, the type becomes:

```
data Initializer = ExpInitializer Exp !SrcLoc
                 | QuotedInitializer Id String [Id] !SrcLoc
                 | CompoundInitializer
                       [(Maybe Designation, Initializer)] !SrcLoc
```

The `QuotedInitializer` data type contains (from left to right):

1. A `quoter_name`. The `id` (name) of the language transform to use (e.g. c_hs, objc_hs, sql, regex, etc.).
2. A `quoter_expression`. The expression to process (a string).
3. A `quoter_args`. The list of ids which are the arguments to the expression.

This is similar to a Haskell quasi quoter where `[name|EXPRESSION|]` matches the first two fields, except a quoted initialiser takes a list of arguments, which are names of variables.

Next we add the syntax to language-c-quote's parser (Parser.y):

```
initializer :: { Initializer }
initializer :
  assignment_expression
  '@' '|' identifier '|' STRING '|' identifier_list '|'
    { QuotedInitializer $3 (snd . getSTRING $ $5)
      (rev $7) (srclocOf $1) }
  | assignment_expression ...
```

This parses the following syntax:

```
@|c_hs|"(+) %d %d"|x, y|
```

By parsing that code fragment, it produces the following AST (with `srclocs` omitted):

```
QuotedInitializer (Id "c_hs") "(+) %d %d" [(Id "x"), (Id "y")]
```

We can not observe the types of arguments from the argument list. However, since we provide types inside the quoter expression, the code generated will expose functions from Haskell with defined types and the C compiler will reject any arguments that are not the correct C type. In the case of `c_hs`, the type information is included into the expression string, this is explained in depth later on.

Finally the implementation of this tool requires some infrastructure. Recall we traverse the AST to replace `QuotedInitializers` with `ExpInitializers`. Since this tool can be used in other problems than code generation, it is more flexible if the transformer used the `IO` monad, so it may access files. Another feature the transformers should have is a setup and finish phase to run code before and after the AST is traversed (to create and clean up files).

To satisfy this, transformers are given the following data type to implement:

```
data Transform = forall a . Transform
    { _init :: IO a
    , _trns :: a -> String -> [Id] -> SrcLoc -> IO (a, Initializer)
    , _fin  :: a -> IO ()
    }
```

Stepping through the components, `_init` is some `IO` function to run before the AST is traversed, `_trns` is some function to run on each occurrence of that transformer and `_fin` is some function to run when the AST has finished traversal. It is also given a value "a" to carry around if it wants to track some state information (such as file handles to write auxiliary files, or building an AST while traversing and only writing it at the end).

The transforms are stored in an associated list of strings with transforms. We lookup which transformer to use based on the string given. It then calls the `_trns` field and returns the carried a value (some state) with the new initialiser. The type of the function that performs the transforms looks like:

```
transform :: [(String, Transform)]
          -> C.Definition
          -> IO ([(String, Transform)], C.Definition)
```

As described earlier, using cpppp is not limited to foreign interface generation. Several example usages of this tool are:

1. More expressive macro code generation by using code templates.
2. Checking and compiling regex, SQL or AutoLayout expressions.
3. Generally problems that the Haskell quasi quoter is used for.

As a simple example, say we wanted to transform any occurrence of the `int_four` quoted initialiser with the C integer constant `4`, we could provide the following implementation:

```haskell
{-# LANGUAGE QuasiQuotes #-}
module Language.C.CPPPP.Transforms.Int4 (mkInt4) where

import Language.C.Syntax
import Data.Loc (SrcLoc)
import Language.C.Quote.C

mkInt4 :: (IO (),
      () -> String -> [Id] -> SrcLoc -> IO ((), Initializer),
      () -> IO ())
mkInt4 = (return (), mkInt4', const . return $ ())

mkInt4' :: () -> String -> [Id] -> SrcLoc -> IO ((), Initializer)
mkInt4' _ expression vars l = return ((), [cinit|4|])
  -- alternatively, (ExpInitializer (Const (IntConst "4" Signed 4 l) l) l)
```

Here the `_init` is `return ()`, the `_trns` is `mkInt4` and the `_fin` is `const . return $ ()`. This will do nothing when the code generation begins, return a C initialiser for the integer `4` when a `int_four` expression is encountered and do nothing when code generation ends.

By using language-c-quote, it is easy to write new transforms by utilising the language-c-quote quasi quoter. This is what we did in the `int_four` example. However, we could use the AST if desired, as shown in the comment on the last line.

4.2 Compilation Process

Before going into the details of the `c_hs` transformer, we must first look at how to run the tool. Firstly, we want to run the cpppp tool on the C code. This will emit new C code and auxiliary Haskell code (containing the auto generated FFI).

Next we want to compile all the Haskell code (including the auxiliary FFI generated code from cpppp). One such way to compile is with `ghc -staticlib Main.hs` where `Main.hs` contains your `Main` module with the `main` function. Typically, renaming the C main function to `c_main` and calling that function from Haskell's main module will simplify initialisation of the Haskell runtime. One recommended implementation is:

```
foreign import ccall safe "c_main" c_main :: IO ()
main :: IO ()
main = c_main
```

In GHC 7.8+, the `-staticlib` flag (described in the Build Process section) produces a statically linked library archive (in our case, named `Main.a`). This simplifies the final step, linking the Haskell archive with the C objects. Say we had several object files from compiling with `clang -c`, we can now link them all with the Haskell `Main.a` with `clang c_file1.o c_file2.o ... Main.a` (plus any linker flags that may be required).

4.3 c_hs Transformer

Now that we have the infrastructure required to generate foreign function interfaces inline in C, we now need a transformer for Haskell. The `c_hs` transformer is such a transformer. The desired properties of such a transformer is to generate the Haskell foreign function interface code required to export the Haskell function used, as well as generating the C code initialiser to call the function. To do this, we emit the Haskell code into an auxiliary file to be compiled and linked with the other Haskell code (using the compilation process described previously).

Our carried state in this case is a file handle to an auxiliary Haskell file, that we will call `HSAux.hs`. In this file, we write each Haskell FFI wrapper that is needed. To open the file before the AST is traversed, one such implementation for `c_hs` would be:

```
initHs = openFile "HSAux.hs" WriteMode
```

Conversely, when traversal is finished, the finish function would be:

```
closeHs = hClose
```

While traversing, we want to emit Haskell code to support our C code. For example:

```
@|c_hs|"(+) %d %d"|x, y|
```

Here we want to create a new function in `HSAux.hs` to define the function that takes 2 C integers and uses them as arguments to the `(+)` Haskell function. So our `HSAux.hs` should append to the file:

```
foreign export ccall hs_ffi_1 :: CInt -> CInt -> CInt
hs_ffi_1 x_0 x_1 = (+) x_0 x_1
```

Where `hs_ffi_1` is a fresh function name and the arguments are also given fresh variable names. To do Haskell code generation, Template Haskell [16] is utilised. However instead of splicing the code we pretty print it into a file. This file is compiled in the next step of the build phase, making template-haskell not a requirement of compiling generated code. This is important when working with cross compilers since using GHC as a cross compiler does not support template-haskell yet. Because we do not require template-haskell in the generated code, this approach can be used with the `ghc-ios` [2] cross compiler.

To generate the C and Haskell code, we must inspect the expression string. The information we need is:

1. Which Haskell function to call.
2. What type the arguments are.

One representation of this information that is hopefully familiar to both C and Haskell programmers are format strings, so we utilise format strings for the expression value in `c_hs` quoted initialisers. In the above example `@|c_hs|"(+) %d %d"|x, y|`, `(+)` is the Haskell function to call, followed by the C types of the arguments, in this case 2 C integers.

The reference implementation works by building a `FCall` type which contains the Haskell function name, the fresh FFI wrapper function name and a list of arguments to call the function with. For example:

```
mkChs' :: Handle -> String -> [Id] -> SrcLoc -> IO (Handle, C.Initializer)
mkChs' h ex vars loc = (fcall ex vars :: IO (FCall CLang)) >>= emit h loc

emit :: (FormatString a)
     => Handle
     -> SrcLoc
     -> FCall a
     -> IO (Handle, C.Initializer)
emit h loc (FCall fname ffi_name args) = do
    let carg_list = map snd args
        cexpr = FnCall (Var (Id ffi_name loc) loc)
                       (map (flip Var loc) carg_list)
                       loc
    hs_expr <- runQ $ mkHsExpr ffi_name fname args
    hPutStrLn h $ pprint hs_expr
    return (h, [cinit|$cexpr|])

-- args look like: [(CType CInt, Id "x" ),(CType CInt, Id "y" )]
mkHsExpr :: String -> String -> [(Arg a, C.Id)] -> Q Dec
mkHsExpr ffi_name fname args = do
  names <- mapM newName $ replicate (length args) "x"
  let body = NormalB $ applyT (reverse names) (mkName fname)
  return $ FunD (mkName ffi_name) [Clause (map VarP names) body []]
```

We call `fcall` on the arguments to wrap them into the `FCall` type and augment the type with a programming language (in this case `CLang`). Then we use `emit` to create a new initialiser while writing to the auxiliary Haskell file. To build the expression to write to the auxiliary Haskell file we traverse the arguments and build a function definition from them.[21]

This step is where we can include marshalling. We can inspect all the types and define transforms for the programming language into Haskell and back. For example:

```
int n = @|c_hs|"factorial %d"|x|;
```

[21] Some function definitions have been omitted for brevity, see the reference implementation for the full source code.

Our `factorial` function uses `Int` types, so `fromIntegral` calls are inserted to marshal the input `x` into an `Int` from a `CInt`, and the return value marshalled back into a `CInt`. For example:

```
foreign export ccall f_123 :: CInt -> CInt
f_123 f = fromIntegral (factorial (fromIntegral f))
```

By extending the format string parser to parsing non-arguments, we can use constants inside of the arguments. For example, we could write `@|c_hs|"writeFile \"test.log\" %s"|str|` to write the `str` C string variable to the file `test.log` using Haskell's `writeFile` function. Other extensions could include integer constants, other function names, etc.

By using this approach, we can eliminate boilerplate code. For example, in an iOS App that compiles Elm [5] code to Javascript, the Elm compiler can be imported and used via the `c_hs` transformer. We add a Haskell function to define the default build parameters and return if a build was successful.

```
compileFileBackend :: String -> String -> IO Int
compileFileBackend freeDir str = build
                                 $ Flags True [] Nothing False False False
                                   [] False False
                                   (freeDir </> "cache") (freeDir </> "build"))
                                   str
```

This function takes a file path to the directory to write compiled file into and a file path to the input file. It sets the default values for the Elm compiler and calls `build`. To call it from C or Objective-C we can use:

```
char *docDir = "/Users/maxs/Documents/";
char *filen = "/Users/maxs/Documents/test.elm";
int status = @|c_hs|"compileFileBackend %s %s"|docDir, filen|;
```

The `QuotedInitializer` would be replaced by a function call similar to:

```
int status = hs_ffi_1(docDir, filen);
```

And the auxiliary Haskell code would look something like this:

```
foreign export ccall hs_ffi_1 :: CString -> CString -> IO CInt
hs_ffi_1 :: CString -> CString -> IO CInt
hs_ffi_1 freeDir' cstr = do
  str <- peekCString cstr
  freeDir <- peekCString freeDir'
  compileFileBackend freeDir str
```

Notice here we pull out the C Strings into Haskell Strings then calls the original function used in the expression.

4.4 objc_hs Transformer

Using the previous definition for generating C foreign function interface code, it is trivial to extend it to support Objective-C. Simply add marshalling to the `mkHsExpr` for each data type in the format string. For example, a `%s` occurrence would map to a `NSString` in Objective-C instead of a `char *`.

Say we wanted to draw a tree in an OpenGL context. We could write a function in Gloss[22] to draw the tree, but it needs a context to draw in. We use Objective-C to get the context from Cocoa then use cpppp to call the draw function from Gloss when the draw frame callback is called.

`drawTree` is a Haskell function that takes a pointer to be used by OpenGL to render the tree by Gloss. In this case, Gloss will need to know how to use this pointer. Its type may be `Ptr () -> Int -> Int -> IO ()`. We would call this with:

```
int n = @|objc_hs|"drawTree %@ %d %d"|glconext, height, width|;
```

The transformer used in this case is the `objc_hs` transformer. It will use only NSObject types. So when the format string `%d` is used, it is a NSNumber. When `%@` is used, it is a NSObject. The quoter will generate the appropriate marshalling to marshal the NSNumber into a Haskell Int and the NSObject into a Ptr. A possible generated implementation could be:

[22] Gloss is a 2D graphics library for Haskell http://gloss.ouroborus.net

```
import DrawTree
import ObjCMarshaling
foreign export ccall hs_ffi_1 :: Ptr NSObject
                              -> Ptr NSNumber
                              -> Ptr NSArray
                              -> IO CInt
hs_ffi_1 p x xs = drawTree (nsObjToPtr p) (nsNumToInt x) (nsNumToInt xs)
                >> return 0
```

It may look strange that this requires a return value, even though it returns `IO ()`. Recall that quoted initialisers must in A-Normal form, so it must have a return value. Since the unit type does not exist in C, we substitute `IO ()` with `IO CInt` and `return 0`.

4.5 Implementation Considerations

Since we want to include Haskell modules outside of `Prelude` into our programs, we can write to the top of the auxiliary files `module HSAux where\nimport HSExport\n`. Then create a `HSExport.hs` where we import and re-export all the functions we want available to C/Objective-C from the different Haskell modules in the project. This explicitly lists in one file everything available to the programmer, making the process simpler.

At the time of writing, the reference implementation does not preserve macros or ingest macro includes to add new C types to the parser's types. So while the examples work as specified, using the tool requires inserting the outputted C/Objective-C code back into a template with the required macros. Once language-c-quote is extended to support this functionality, the tool should integrate seamlessly into a build process. Adding the build step to a Makefile is usually done by adding one rule for `FILENAME_cpppp.m` → `FILENAME.m` and `FILENAME_hs.hs`. Adding this step to Xcode is usually done by adding a "Build Phase" to call `cpppp FILENAME_cpppp.m`, then adding the emitted `FILENAME.m` and `FILENAME_hs.hs` to "Build Sources".

Also, the reference implementation does not use the return type of an expression. This is omitted from the `Transform` data type. The previous examples do not require this feature. However, if so desired, it is trivial to add to this feature to cpppp.

Finally, the reference implementation does not yet parse multiple files at once. This feature could be added if such a feature was desired. It is usually not useful because most programs are

structured such that cpppp will only interact with the other programming language from one Class or Module. This is to simplify the design of the program, however it could be added.

4.6 Extended Usages

Since cpppp lets us write code to run before compiling the program, we can solve problems similar to problems we solve with quasi quoters in Haskell. For example, consider the NSPredicate class. It uses a string to perform queries. To search for all people with a name that starts with `Bo` you might use `@"people.name like \"Bo*\""`. If you had something wrong with this string, you will not know until run time. By writing a transformer for NSPredicates we can check this string at compile time by using an quoted initialiser such as `@|nspred|"people.name like \"Bo*\""||`.

Auto Layout has a representation of layouts as strings. These strings could be type checked at compile time with quoted initialisers. However, the same string could be used instead to generate code that describes the layout at compile time, instead of generating the layout at run time. Furthermore, this same string could then generate code for describing that layout in other frameworks, such as ReactiveCocoaLayout.

Regular expressions and SQL statements are usually expressed as strings. By extending cpppp their validity can be checked at compile time, and in some cases optimisations may be performed or warnings provided to the programmer.

5 The Build Process

One of the complexities with compiling programs written in multi programming languages is the build process. Conventionally [17] it is complex to link the binaries produced by both compilers. Especially if we are using a package manager (like Cabal in Haskell) since we need to locate and add libraries to our linker options.

For example, repa-examples' beholder[23] program uses a script to pull in all the Haskell static libraries from the compiler's directory (`libHSrts_tthr.a`, `libHSffi.a`, `libHSghc-prim.a`, `libHSinteger-gmp.a`, `libHSbase.a`, `libHSbytestring.a`, `libHScontainers.a`, `libHSpretty.a`, `libHStemplate-haskell.a`) and cabal dependencies (vector, repa, repa-bytestring, primitive). We need to list all the file names and versions of each dependency. It then symlinks them into a `hslib` directory and the compiler then link them all.

For example, to add the `vector` dependency, we need to add the following code to the script:

```
ln -s $LIB_CABAL/vector-0.7.0.1/$GHC_VERSION/libHSvector-0.7.0.1.a
```

ObjectiveHaskell improved on this by generating this information. The script[24] queries Cabal for the archive file paths. The programmer still must incorporate this into the build process and provide a list of dependencies.

Since these processes are tedious and do not fit well into the Haskell build process, I added a new flag to GHC 7.8 to support statically linking libraries into non-executable archives. It uses the cabal lookup internal to GHC to link dependent libraries automatically. For example, by running:

```
ghc -staticlib Main.hs
```

This produces a statically linked archive in a fashion similar to the `--make` flag. For example, beholder can now use `ghc -staticlib beholder/Process.hs` then use the emitted `Process.a` when linking the executable. This simplifies working with Haskell in any cases where linking the executable with the C compiler (not GHC) is preferable, usually when the IDE has a complex build process and uses a wide range of compiler options, such as Mac OS X Apps, iOS Apps, distributing libraries, etc. This is easy to include in an IDE's build process, since adding this usually only requires a new "run script" build phase before linking and including the emitted archive in the list of objects to link.

[23] http://code.ouroborus.net/beholder/beholder-head/
[24] https://github.com/jspahrsummers/ObjectiveHaskell/blob/reboot/script/flagsForPackage.hs

6 Related Works

6.1 HOC

HOC [15] is a Haskell to Objective-C binding library, with the capability to operate with, and define, Objective-C objects from Haskell while remaining type-safe. HOC provides the programmer a way to use Haskell but requires the programmer to create code to provide the interface between the two languages. This approach requires the programmer to define how the two representations should work. This extra work is overhead the approaches here can remove.

6.2 ObjectiveHaskell

ObjectiveHaskell[25] is similar to HOC, however also provides code generation for the foreign function interface and an extensive build system along with the bridging library. The shortcoming of using a bridging library, as per HOCs, is the separate representations of your program. In order to use a function from Haskell, the developer must export the function from Haskell using template-haskell, then import the function in Objective-C. While ObjectiveHaskell provides some code generation for the Haskell interface, cpppp will perform all the code generation without requiring template-haskell for building the Haskell static library. However, cpppp does not have a class method or object representation, instead only functions may be represented.

6.3 RubyMotion

RubyMotion[26] is an implementation in Ruby, a dynamically typed object-oriented programming language. It benefits from the ability to use its classes as Objective-C classes, so there is no disconnect between the programming language paradigms. It provides a sophisticated build system as well, like ObjectiveHaskell. The main idea is Objective-C objects and Ruby objects work similarly. The developer is not required to export or import functions or methods. While Haskell and Objective-C have significantly different programming language paradigms, the approaches provided in this thesis should match the convenience RubyMotion provides by allowing seamless interpolation between objects, but by using different abstractions that could be utilised to extend the programming language.

[25] https://github.com/jspahrsummers/ObjectiveHaskell
[26] http://www.rubymotion.com

6.4 Xamarin

Xamarin[27] uses a combination of code generation, binding libraries and runtime queries to provide a C# library abstraction. It is more expressive than Objective-C by extending the language with C# language features. It has multiple platform targets allowing apps to be written in C# and compiled to iOS, Android, Windows or OS X. While this approach is powerful it is also time consuming to build. Every new feature needs to be considered and implemented correctly, similar to the EDSL. But since the EDSL has strong type constraints and only performs code generation, it should be simpler to add new functionality to the EDSL than to Xamarin. This is not a practical problem in using Xamarin, just implementing or extending Xamarin.

6.5 libextobjc

While libextobjc[28] does not provide foreign function interface generation, it does provide compile time checks by using macros that we could only otherwise get by performing code generation. This solves one of the motivations of this thesis without using two programming languages. This library could then be used in conjunction with an approach to further extend our ability to reject bad programs.

[27] http://xamarin.com
[28] https://github.com/jspahrsummers/libextobjc

7 Results and Conclusions

At the beginning this thesis the aim was to provide a simple and powerful method for using Haskell and Objective-C to write more powerful Cocoa applications. By exploring current approaches and how the problem is solved in other programming languages, two approaches are proposed by extending previous approaches. We now evaluate how each of the contributions function in practice.

7.1 EDSL

The reference implementation of the EDSL approach is available at: `https://github.com/maxpow4h/language-objc-edsl`.

While this approach is appealing because of the utility provided by adding additional type information to the program to help development, but I encountered several trade offs in using this approach.

The main advantage is we can write programs and type check them in ways not available in normal Objective-C. Also we can utilise the type inference engine to write polymorphic code and then specialise with a type class (such as by using `new` and `newProp`).

One trade off is the work required to implement the APIs is not automate-able since every function and type needs to be reasoned about and the correct types selected. This is complex and time consuming.

The other main trade off is it complicates the build process to incorporate multiple phases of template-haskell code generation. In some cases it is easier to just write the Objective-C and use the interface because the main value with your program is in the computation written in Haskell, not in the GUI to drive it.

Finally type errors in any EDSL are complex and not easy to understand. They can be tricky to understand and not usually obvious.

7.2 Cpppp

The reference implementation of the cpppp approach is available at: `https://github.com/maxpow4h/cpppp`.

By using the cpppp approach I could eliminate boilerplate code. Most of my FFI related code was deleted. It was quick to extend existing Objective-C programs with Haskell features. It was also useful when an implementation already exists in Haskell, since I can import it and use it directly, most of the time with only slight modification.

It was easier to use than a FFI. Whenever I would of had to perform typical marshalling, I added it to the transformer so it would be automatic for any occurrence of that type. However, when encountering my own defined types, it did require some knowledge of the FFI in order to pull out the fields from the structs. Potentially, `c2hs` integration could automate this process too, but it was not a common problem I encountered while using cpppp.

The main disadvantage is the program is not seamless and needs to be added to the steps in the build process. This is should instead be a drop in replacement for a C pre-processor, but it is not possible to do that yet.

A potential disadvantage is you can not use it to call from Haskell into Objective-C. You need another library to do that, such as HOC. However, by designing programs in the MVVM design pattern, this was not necessary.

7.3 Comparison of EDSL and Cpppp Results

I have found the two approaches excel in different situations. Depending on the approach selected, you are promoting the capabilities of some component. For the EDSL we are simplifying the Objective-C side of the program. For cpppp we are offloading the work to a code generator, simplifying the FFI usage.

Which approach to use depends on what is most valuable to the program. If the value of the program is some computation or backend which is written in Haskell and implementing the GUI in Objective-C is fine, it is simpler to interface with cpppp. However, if the value of the program relies on the correctness of the GUI or you want to use Haskell functions to control the GUI, then the EDSL approach may be a simpler.

Another consideration is the EDSL approach requires extending the library when adding new types or functions that do not exist in the library yet. This can be time consuming work, so cpppp may be more beneficial since it is automated by using code generation.

Another consideration is if you are using a pre-existing Haskell library or Objective-C program. It may take little to no code to add the Haskell library to an Objective-C program via cpppp. If

you were using the EDSL approach, significant refactoring on the Objective-C code would need to occur to create classes that can interface with the Haskell library.

7.4 Build Process

The changes to the build process made compiling Haskell with C/Objective-C programs simpler. The process is a huge improvement and should be used when appropriate. It also is easy and seamless to add to IDEs, and has proven especially useful for Mac App development.

7.5 Conclusion

At the beginning of this thesis we identified some difficulties with using multiple programming languages in the same program. By evaluating the motivations behind why we would want to do this, we categorised existing approaches and identified two new fields we could explore.

One approach simplified the Objective-C programming language's object abstraction, then extended it with features unavailable in the language. We were able to use this to write more interesting Objective-C programs in our EDSL.

The other approach let us write normal Objective-C but use Haskell functions inline. We were able to use this approach to easily build on Haskell libraries in Objective-C.

Finally we considered how the two languages should be compiled and linked. We identified a shortcoming of the current build process and corrected it.

Using and experimenting with the tools developed throughout the course of this thesis has provided insight into how developing programs (primarily Cocoa applications) can be simplified and extended. While the implementations are not fully featured yet, the tools are powerful enough to create new kinds of programs using these simpler approaches. Both approaches are a step forward conceptually towards a simpler and more powerful process to developing complex programs.

7.6 Future Work

- Combining the EDSL approach with the cpppp approach would let us benefit from both approaches. A transformer to parse monad syntax and load the AST into the code generator could check correctness of the expression and then inline the generated code.

- Quoted initialisers have many applications in other domains, such as SQL strings, regular expressions and other problems that can be simplified with compile time code generation. Future work could explore the possibility of utilising quoted initialisers for other domain problems.
- The reference implementation of cpppp must be run as a step in the programmer's build process. This is suboptimal since it adds a new step to the build process. Future work could be to make cpppp able to be used as a replacement for CPP, allowing it to work transparently.
- Adding a C++ transformer to cpppp would help write C++ programs. Future work could explore this possibility.
- Adding the ability to nest pure Haskell functions as values inside of the cpppp expressions would be convenient and could be considered for future work.
- The EDSL could be extended to utilise other front ends, such as the Elm [5] compiler. A programmer could compile an Elm program to Objective-C.

References

[1] David M Beazley et al. "SWIG: An easy to use tool for integrating scripting languages with C and C++". In: *Proceedings of the 4th USENIX Tcl/Tk workshop*. 1996, pp. 129–139.

[2] Stephen Blackheath and Luke Iannini. *Building a GHC cross-compiler for Apple iOS targets*. url: `http://ghc.haskell.org/trac/ghc/wiki/Building/CrossCompiling/iOS`.

[3] Manuel M. T. Chakravarty. "C ⟶ HASKELL, or Yet Another Interfacing Tool". In: *Implementation of Functional Languages*. Ed. by Pieter Koopman and Chris Clack. Vol. 1868. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 131–148.

[4] Manuel M. T. Chakravarty. *language-c-quote Motivation*. 2013. url: `https://github.com/mchakravarty/language-c-inline/wiki/Motivation`.

[5] Evan Czaplicki and Stephen Chong. "Asynchronous functional reactive programming for GUIs." In: *PLDI*. 2013, pp. 411–422.

[6] Manuel M. T. Chakravarty (Ed.) *The Haskell 98 foreign function interface 1.0*. Tech. rep. 2002.

[7] Simon Marlow (Ed.) *Haskell 2010 Language Report*. Tech. rep. 2010.

[8] Cormac Flanagan et al. "The essence of compiling with continuations". In: *ACM Sigplan Notices*. Vol. 28. 6. ACM. 1993, pp. 237–247.

[9] John Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. 2005. url: `http://blogs.msdn.com/johngossman/arcNve/2005/10/08/478683.aspx`.

[10] Paul Hudak. "Building Domain-Specific Embedded Languages". In: *ACM Computing Surveys* 28 (1996).

[11] Apple Inc. *Key-Value Observing Programming Guide*. 2012. url: `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.pdf`.

[12] Apple Inc. *Objective-C Runtime Reference*. 2013. url: `https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/ObjCRuntimeRef.pdf`.

[13] Marcin Kowalczyk. *Writing Haskell interfaces to C code: hsc2hs*. url: `http://www.haskell.org/ghc/docs/7.2.1/html/users_guide/hsc2hs.html`.

[14] Geoffrey Mainland. "Why it's nice to be quoted: Quasiquoting for Haskell". In: *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. Freiburg, Germany: ACM, 2007, pp. 73–82.

[15] André T. H. Pang and Manuel M. T. Chakravarty. "Interfacing Haskell with Object-Oriented Languages". In: *Implementation of Functional Languages*. Ed. by Philip W. Trinder, Greg Michaelson, and Ricardo Pena. Vol. 3145. Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 20–36.

[16] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, 2002, pp. 1–16.

[17] John Velman. *Using Haskell in an Xcode Cocoa project*. 2009. url: `http://www.haskell.org/haskellwiki/index.php?title=Using_Haskell_in_an_Xcode_Cocoa_project&oldid=31345`.