Programming Languages and Compiler Construction
Department of Computer Science
Christian-Albrechts-University of Kiel

Bachelor Thesis

Design and Implementation of Remote Function Invocation with Template Haskell

Jan-Patrick Baye

SS 2013

Advised by:
Prof. Dr. Michael Hanus
Dipl.-Inf. Fabian Skrlac

Statutory Declaration

I hereby declare that I have authored this thesis independently and that I have only used sources listed in the bibliography, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The thesis in this form or in any other form has not been submitted to an examination body and has not been published.

date *Jan-Patrick Baye*

Contents

1. Introduction **2**

2. Preliminaries **4**
2.1. Template Haskell 4

3. Implementation **6**
3.1. Basic concept 6
3.2. Protocol 7
3.3. Monomorphic functions 8
3.4. Polymorphic functions 8
3.5. Client framework 12
3.6. Generating remote functions 12
3.6.1. Stub signature 13
3.6.2. Stub generation 14
3.7. Server framework 16
3.8. Server functions 16

4. Conclusion **19**
4.1. Performance 19
4.2. Usage 20
4.3. Code Base 20
4.4. Polymorphic Functions 20
4.5. Conclusion 21

A. Installation and usage **22**
A.1. Installation 22
A.2. Testing the installation 22
A.3. Usage 22

B. Examples **24**
B.1. Specifying RFI functions 24
B.2. Starting a server 25
B.3. Using remote functions 25
B.4. A simple chat 26

Bibliography **27**

Listings

2.1. lambda function and abstract syntax tree 4

3.1. stub signature 6
3.2. using a remote functions with multiple servers 6
3.3. genRFI signature 6
3.4. protocol layout 7
3.5. parameter (de-)serialization 7
3.6. MonoRep and simple Mono type class 8
3.7. Mono type class 9
3.8. Mono instance examples 9
3.9. generic Mono instance for Either 9
3.10. generate Mono class instances 10
3.11. generate the Mono type constraints 11
3.12. generate toMono and fromMono for an Mono instance 11
3.13. query function 12
3.14. simple type signature 13
3.15. stub signature 13
3.16. stub definition 14
3.17. generate stub 15
3.18. remote_add example 16
3.19. generate argument variables 16
3.20. generate the tuple's type 17
3.21. creating the server function's body 17
3.22. generate match 18
3.23. server code for add 18

4.1. a function that would benefit from lazy evaluation 19

B.1. RFI functions (examples/usage.hs) 24
B.2. server (examples/server.hs) 25
B.3. client (examples/client.hs) 25
B.4. simple chat application (examples/chat.hs) 26

1. Introduction

This thesis examines the development of an Haskell framework for Remote Function Invocation (RFI) with Template Haskell. The main motivation are resources that are only available on specific machines. We want to provide a way, that one can write the functions to work with these resources, as though they are executed on the same machine and then specify them to be remote accessible functions. Our framework then should generate an application programming interface (API) that provides two sets of functions: One to provide a server, listening for function calls and executing them and the other with client functions, which connect to the server and send it the requests. In Figure 1.1 we see a simple example of what we want to achieve. In this example the *log* function should be executed on the server, but we want to make it callable from client machines. For this our approach generates the *remote_log* function.

Another inspiration for RFI was Java Remote Method Invocation (Java RMI)[7]. Especially in with respect to its simplicity and basic layout. Java RMI allows it, that the programmer does not need to do anything with respect to serialization (except make sure everything he uses is serializeable). A main goal of RFI is to replicate this simplicity in Haskell.

In the next chapter we will introduce Template Haskell, a compile-time meta-programming extension of Haskell, which is used to implement RFI. After that we will discuss the implementation. First we will look at the basic concept we use and our protocol to send our functions calls over the network. Then we take a look at monomorphic and polymorphic functions, what the problems are if we want to provide a generic way to use them over the network and how we solved them. At least we show how we generate

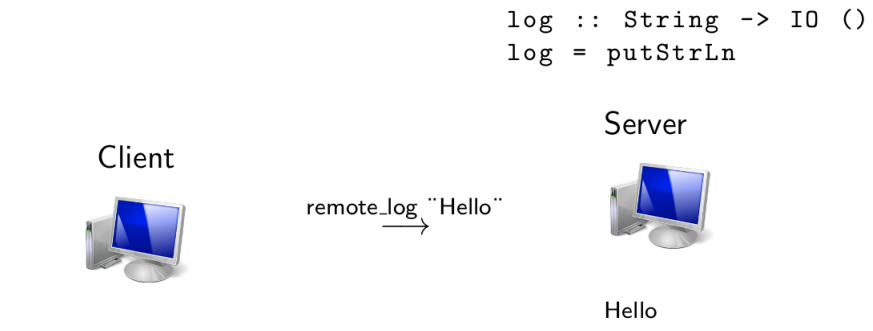

Figure 1.1.: Concept

the client and server code and what interface our module will create. In our last chapter we want to discuss our solution. What it can and can not do and why not. We will also compare our approach with a similar work from Jeff Epstein, Andrew P. Black and Simon Peyton-Jones.

2. Preliminaries

2.1. Template Haskell

Template Haskell is a part of GHC since version 6. It provides a way to do type-safe compile-time meta-programming and is an extension of Haskell 98. Template Haskell allows us to manipulate the code at compile-time [5]. This is done with abstract syntax trees. These are a representation of Haskell code with Haskell data types. We can convert concrete code to it's syntax tree representation (reify) work with it and then convert it back to concrete code (splice back). This way we can manipulate existing code or even create new code at compile-time. Besides the data types for the syntax trees two functions are relevant for us: *mkName* and *reify*.

mkName :: String -> Name is used to generate a capturable name, whereas *reify :: Name -> Q Info* queries the compiler about information about the *Name*. If we have a function (e.g. *add :: Int -> Int -> Int*) and its name as a string ("add") we can use these two functions to obtain information about *add*. We first generate the *Name* with *mkName* and then query the compiler for information about it with *reify*. If we do this for a function the resulting *Info* variable contains the type signature and the definition as abstract syntax trees. In Listing 2.1 we see a lambda function and the corresponding syntax tree. The *LamE* constructor, which represents a lambda function, takes two arguments, a list of patterns and an expression. In the example the list consists only of a single element representing the x. In this code *x_1* is of type *Name* and we can see how the same name variable is used with different constructors in this syntax tree. First with *VarP* to create a pattern variable and then with *VarE* to use the variable in the expression. If we had obtained this syntax tree with *reify* we could simply change the tree, for example change the operation to decrement x. The *Name* of the $-$ operation can be obtained with *(mkName "-")* and then we only have to substitute the *GHC.Num.+*. For more information on the abstract syntax trees see [4].

```
\x -> x + 1

LamE [VarP x_1]
     (InfixE (Just (VarE x_1))
             (VarE GHC.Num.+)
             (Just (LitE (IntegerL 1))))
```

Listing 2.1: lambda function and abstract syntax tree

To use Template Haskell the Glasgow Haskell Compiler (GHC) version 6 or higher has to be used. Also a compile time flag (*TemplateHaskell*) has to be set for every module that uses it. One drawback of generating code with Template Haskell is that you can not use the generated or modified functions in the same module that you create or modify them

in, respectively. We will use Template Haskell for two tasks, we will query the compiler with *reify*, to get information about the functions we have to generate RFI-code for. We then use Template Haskell to generate the code we need.

3. Implementation

3.1. Basic concept

To make a function remotely callable we have to generate two functions: one function for the client side and one function for the server side. The client function we will call *client stub* or *remote function*. Functions the user wants to use with RFI we will call *RFI functions*. The client stub has to have the same arguments as the original function, send the serialized arguments to the server and finally de-serialize the result. Because the client has to send the arguments over the network the result is wrapped in Haskell's *IO* monad. This is where the function signature differs from the original function. There is one exceptions to this rule, if the function we make remote callable already is an *IO* function, we do not have to wrap the return type. To make the *stubs* more flexible they also take a *RFIHandle* argument to specify the server that should be used. This enables us to let the client connect to multiple servers and use the same client stub to call the function on any one of them. (This principle is shown in Listing 3.2.)

```
inc :: Int -> Int
remote_inc :: RFIHandle -> Int -> IO Int
```

Listing 3.1: stub signature

The server side function has to de-serialize the arguments and then apply the original function. The result then has to be serialized and send back to the client.

In addition there has to be one function for each side to handle the connection. The client version has to establish a connection to a server and then provide the *RFIHandle* to be used by the client stubs. The server version has to both accept connections from clients and then accept their messages and invoke the corresponding functions.

```
% server1 : RFIHandle (connection to a server)
% server2 : RFIHandle (connection to another server)

i <- remote_inc server2 (5 :: Int)
i <- remote_inc server1 i
% i = 7
```

Listing 3.2: using a remote functions with multiple servers

To make this work, we have to provide an interface for the programmer to mark a function for RFI usage (Listing 3.3). The *genRFI* function takes a list of function names as argument. For these functions our module then tries to generate the necessary code in the form of abstract syntax trees.

```
genRFI :: [String] -> Q [Dec]
```

Listing 3.3: genRFI signature

3.2. Protocol

First we have to define a protocol for the communication between server and client. For the underlying network communication *TCP/IP* was chosen. Also between function calls there is not really a need for a continuing connection *TCP* was chosen over *UDP*, because *TCP* already handles lost packets. If *UDP* would be used, this would have to be implemented manually, otherwise a deadlock might occur on client side when a packet is lost because the called function would not return.

On the application layer two things have to be considered: first how to serialize the arguments and return value and second how to encode which function the client wants to call. To serialize the data we choose the *show* and *read* methods, which produce a simple string representation of the data and works for every basic data type and makes debugging easier, because it is still readable. The string representation is then send over the network and as the delimiter for different function calls the newline is used. So every line is one function call with its arguments.

```
-------------------------------------------------------
| functionname | space | (arg1, arg2, ..., argn) | \n |
-------------------------------------------------------
```

Listing 3.4: protocol layout

To identify the called function we simply use its name as a string. The client puts this in front of the serialized arguments. As delimiter we use a space. Haskell function names can not contain any whitespace. Because of this we can safely split the string at the first occurrence of a space, in this way obtaining the function name as the first and argument representation as second part. To represent the arguments we use an n-tuple, where n is the number of arguments. The first argument of the function becomes the first element and the last argument the n-th element. The whole tuple is then serialized with show. On the server we can then de-serialize the tuple with read and obtain the arguments with pattern matching.

```
add : Int -> Int -> Int
add a b = a + b

remote_add : Int -> Int -> IO Int
remote_add a b = read $ send ("add" ++ "␣" ++ (show (a, b)))

server_add : String -> String
server_add s = let (a, b) = read s :: (Int, Int)
               in show (add a b)
```

Listing 3.5: parameter (de-)serialization

In Listing 3.5 we see the basic layout of a client stub for a binary function add. To simplify this code the *RFIHandle* argument was omitted. *Send* represents a function that sends its argument to the server and returns the answer. The listing also shows the basic

layout of the server function. The function mapping was left out, which we will cover later (reference). The argument string is parsed into a tuple to obtain the parameters and then the original function is called. Because this part has to be generated with Template Haskell we also have to provide the exact type of the tuple for the compiler to use the right *read* to parse the string.

3.3. Monomorphic functions

Monomorphic functions have no type variables in its signatures and as long as every argument type and the return type implements the type classes *Show* and *Read* it is possible to serialize the arguments on the client side, de-serialize them on the server side apply the function and send the result back to the client. Far more interesting are polymorphic functions, which we will discuss next.

3.4. Polymorphic functions

To provide a way to use polymorphic functions over the network, we have to have some way of serializing the polymorphic arguments. The problem with polymorphic arguments is, that although we know the concrete type on the client side, we do not know it at all on the server side. Thus we could easily serialize the data, but not de-serialize it. One way to provide this information is to encode the data type in the message, but it is also possible that the server does not even know that data type, so it would still be unable to reconstruct the data. To provide at least some support for polymorphic functions we use a new type (*MonoRep*) to represent all polymorphic arguments. We therefor expect every data type, that is to be used with polymorphic functions, to provide a way to convert it into *MonoRep* and back again. To formalize this we can use a new type class *Mono* (Listing 3.6).

```
newtype MonoRep = Mono String
        deriving (Show, Read)

class Mono a where
  toMono :: a -> MonoRep
  fromMono :: MonoRep -> a
```

Listing 3.6: MonoRep and simple Mono type class

If we take a look at *length :: [a] -> Integer*, we see that to use this function over the network we somehow have to serialize the list, but also have to preserve the list structure. So we have to convert the list of *a* into a list of *MonoRep*. In this case we could simply use *map* to convert the list. But let us also take a look at another common data type: *Either*. In the case of *Either* we could also have a type like *Either a Int*. If the function gets an argument constructed with *Left* we have to convert *Left*'s argument to *MonoRep*. On the other hand, if *Right* is used we do not want to convert the *Int* value. To do this with our *Mono* class we have to generate very specific conversion code to map the different data types. It would be better if our *Mono* class could handle it, so that if we

have an argument x of type *Either a Int* we can call *toMono x*, which then converts x to *Either MonoRep Int*.

The *Mono* class

We can realize this requirement using multi-parameter type classes. Simply speaking these enable us to define a relation between multiple types. In our case we can modify our type class to take two parameters and change the type of *toMono* and *fromMono* to convert between these two types. The modified class is shown in Listing 3.7.

```
class Mono a b where
  toMono :: a -> b
  fromMono :: b -> a
```

Listing 3.7: Mono type class

Let us first look at how we can use this to implement our basic case for converting something to *MonoRep* and back again. If we want to convert a type *a* to *MonoRep* we simply have to define an instance of *Mono a MonoRep*. Since we already use *show* and *read* to serialize the arguments of monomorphic functions, we can use them also to convert most data types to *MonoRep*. For every type that implements the type classes *Read* and *Show* we can provide a default instance of *Mono* (Listing 3.8).

```
instance (Show a, Read a) => Mono a MonoRep where
  toMono = Mono . show
  fromMono (Mono str) = read str

instance (Show a, Read a) =>
          Mono (Either a Int) (Either MonoRep Int) where
  toMono   (Left x)  = Left  (toMono x)
  toMono   (Right x) = Right (toMono x)
  fromMono (Left x)  = Left  (fromMono x)
  fromMono (Right x) = Right (fromMono x)
```

Listing 3.8: Mono instance examples

This basic case can now be used to implement the more complex cases like our *Either* example. Remember we tried to convert *Either a Int* to *Either MonoRep Int*. If we look at *Either*'s constructors separately, we notice that we just have to provide a way to convert between *a* and *MonoRep* for the *Left* constructor and between *Int* and *Int* for the *Right* constructor. For types that implement *Show* and *Read* we already have defined the first conversion. What is left is a conversion between *Int* and *Int*. Since we do not want to change anything with this conversion, we can simply use the identity function (*id*) and provide an instance of *Mono*: *Mono Int Int*. Listing 3.8 shows the resulting implementation. We can further generalize this implementation by using the *Mono* type class itself as type constraints. If we want to convert between *Either a b* and *Either c d*, we can do this in the same way as before, if instances for *Mono a c* and *Mono b d* exist. The resulting implementation for *Either* is shown in Listing 3.9.

```
instance (Mono a c, Mono b d) =>
          Mono (Either a b) (Either c d) where
```

```
toMono (Left x) = Left (toMono x)
toMono (Right x) = Right (toMono x)
fromMono (Left x) = Left (fromMono x)
fromMono (Right x) = Right (fromMono x)
```

Listing 3.9: generic Mono instance for Either

Generating *Mono* class instances

Now that we have seen how we can create an abstract *Mono* class instance for an algebraic data type and seen that for every built-in data type like *Int* or *String* we have to define an instance of *Mono a a* to use them with RFI we want to generate the necessary *Mono* instances with Template Haskell. First let us analyze what we have to generate. For the basic data types we only have to generate a simple instance with *toMono* and *fromMono* as aliases of *id*. For algebraic data types we have to do a little more. As the *Either* example shows, we have to generate one *toMono* and *fromMono* version for each constructor and then call *toMono* or *fromMono* for each argument, respectively. For this we also have to add the *Mono* type constraints as shown in Listing 3.9. Listing 3.10 shows the basic layout of *genMonoInstances*. Note that we generate instances for n-tuples separately because they are represented with Template Haskell's *TupX* constructors, but the basic idea of how to do it is the same, although we only need one pattern per function (*toMono*, *fomMono*).

```
genMonoInstances :: [String] -> Q [Dec]
genMonoInstances [] = return []
genMonoInstances names =
  do
    types <- createFunTypeList names
        let typelist = filter (\x -> not $ show x == "GHC.Types.IO")
                       $ removeDuplicates $ concat
                                          $ map listOfDataTypes types
    let tupletypes = removeDuplicates $ concat
                                       $ map listOfTupleTypes types
    infos <- getInfoList typelist
    return $ (map genMonoInstance $ zip typelist infos) ++
             (map genMonoTupleInstance tupletypes)
```

Listing 3.10: generate Mono class instances

To generate all instances we first have to find out what data types are used. For this we obtain a complete list of all RFI function signatures from the compiler. These signatures can be parsed to get a list of all used data types. Since a single data type might be used multiple times, we have to remove duplicates from that list or we would produce a compile time error, when we try to generate multiple instances of the same type. We also have to remove the *IO* monad from that list because it is a special case and we do not serialize it. With a complete list of the used types we can obtain additional information about these types, for example a complete list of all constructors. With all these information the only thing left is to actually generate an instance for each data type. This is done by *genMonoInstance*, which we will discuss next.

GenMonoInstance basically has three cases, the first two map against simple data type or type aliases (e.g. String) and simply generate an instance, where both *toMono* and *fromMono* are an alias for *id*. These two cases we will not look at any further. The third case maps against the algebraic data types. From the abstract syntax tree representation of the data types we can extract two pieces of information. How many different type variables are used and the format of the different constructors.

Let us take a look at how we generate the *Mono* type constraints. When we look back at Listing 3.9, we see that for every type variable used in the definition we have to generate a *Mono* type constraint with a new type variable. For this we generate two lists of type variables called *a1, ..., an* and *b1, ..., bn* to represent the types in the constraints. We can combine the two list with *zip* to a single list of pairs and Listing 3.11 shows how to generate the type constraints from that list.

```
genMonoConstraints :: [(Type, Type)] -> Cxt
genMonoConstraints [] = []
genMonoConstraints ((a, b) : xs) =
  ClassP (mkName "Mono") [a, b] : genMonoConstraints xs
```

Listing 3.11: generate the Mono type constraints

We will skip how to generate the rest of the type signature and look at how to create the *toMono* and *fromMono* functions for an instance. Because those two functions are identical except for the small difference of the function name and which function they have to call for each argument, we can use a single function to generate the *Clauses* for both cases. Listing 3.12 shows how we generate these functions. *genFun* takes a list of constructors as argument and the generates the two functions. Since we have to create one *Clause* for every constructor we can use *map* to generate the list of *Clause*'s needed for the *FunD* constructor.

The basic idea of *genClause* is to distinguish between two types of constructors. Constructors with and without arguments. The first pattern matches against the second type of constructor and the resulting *Clause* is accordingly simple. Since no arguments are involved, all we have to do is call the same constructor.

The second case is more interesting. We can extract a list of variable names from the constructor definition. With this list in hand, it is possible to create the correct constructor pattern (*ConP*) and then recreate the constructor call while wrapping each argument within the function *f*, which in our case is either *fromMono* or *toMono*. *GenConCall* does exactly that. To generate the constructor call it has to create a chain of *AppE* expressions, which are similar to the *AppT* expression we already know from the type signatures.

```
genFuns :: [Con] -> [Dec]
genFuns xs = (FunD (mkName "toMono")
                   $ map (genClause $ mkName "toMono") xs)
             : [FunD (mkName "fromMono")
                     $ map (genClause $ mkName "fromMono") xs]

genClause :: Name -> Con -> Clause
genClause _ (NormalC name []) = Clause [ConP name []]
                                       (NormalB (ConE name))
```

```
                                          []
genClause f (NormalC name xs) =
  let varnames = map (\(_, (VarT n)) -> n) xs
  in Clause [ConP name (map (\x -> VarP x) varnames)]
            (NormalB $ genConCall varnames f (ConE name))
            []
```

Listing 3.12: generate toMono and fromMono for an Mono instance

With this we can generate instances for the basic data types, algebraic ones and n-tuples. Lists are also supported, but an instance for lists is statically defined in *Monomorph.hs*, since it works for nested lists.

3.5. Client framework

For the client side we provide one function to initiate a connection with a server and for every function the programmer wants to use with RFI we generate one remote function as specified before in section 3.1. The connection to the server is established using the *startRFI_* function, which takes a host name or IP-address and a port number and then returns the corresponding *RFIHandle*. This handle can than be used as argument for the remote functions. Because we use Template Haskell, the programmer has to define the RFI functions in an own module and there call *genRFI*. To re-export *startRFI_* from that module, a wrapper function (*startRFI*) is generated with Template Haskell. Because, if everything of that module is exported by default, all generated functions are also exported and thus the programmer can use *startRFI*. To send to and receive data from the server the query function (Listing 3.13) is used.

```
query :: (Show x) => String -> RFIHandle -> x -> IO String
query name h p = let handle = sHandle h
                 in (hPutStrLn handle $ name ++ "␣" ++ (show p))
                    >> hGetLine handle
```

Listing 3.13: query function

Query takes three arguments, the name of the original function as identifier as specified in the protocol (3.2). The RFIHandle specifying the server and the functions arguments already joined as tuple. It then assembles the message according to the defined protocol and sends the request to the server. Finally it returns the servers answer. Thus this function is used by every remote function to communicate with the server. It is important to note that *query* does not de-serialize the return value, this has to be done by the calling function.

3.6. Generating remote functions

For each RFI-function we have to generate one remote function with its corresponding header. To generate the signature we first need the original function's type signature. We then can obtain all other information we need from the signature. For example the number of arguments, which we need to generate the n-tuple. We can also check if we

are able to use that function with Remote Function Invocation. This includes whether the function uses type constraints or is a higher-order-function, which we can not use.

3.6.1. Stub signature

Because of the transformation to monomorphic functions on the server side it is essential to generate a complete function signature for the remote function. Otherwise the compiler does not know which *toMono* and *fromMono* to call.

```
Int -> IO Integer

AppT (AppT ArrowT (ConT Int)) (AppT (ConT IO) (ConT Integer))
```

Listing 3.14: simple type signature

Listing 3.14 shows the signature of an simple *IO* function and its corresponding representation using Template Haskell's syntax trees. Albeit a little simplified because instead of the type names (*Int*, *Integer* and *IO*) *Name*s would be required. These can be obtained using *mkName("typename")*. The two important parts in this representation are the constructors *AppT* and *ArrowT*. *AppT* connects two *Type* objects and *ArrowT* representing the '->' connection. In this way we can construct a type signature as a tree of *AppT* constructs, the second argument being applied to the first one. This way *ArrowT* can be seen as a binary function, so *AppT ArrowT Int* correlates to *Int ->*, thus still missing the second argument. In this case *AppT (ConT IO) (ConT Integer)*.

```
-- generate the stubs type
-- put the RFIHandle in front of the original signature
-- (polysig_help does the rest)
polysig :: Type -> Type
polysig t | isHigherOrderFunc t =
    error (name ++ ":␣higher-order-functions␣are␣not␣possible␣with␣RFI")
polysig (ForallT a [] t) =
  let names = map (\(PlainTV a) -> a) a
      constraints = map (\x -> ClassP (mkName "Mono")
                                       [VarT x,
                                        ConT (mkName "MonoRep")])
                        names
  in ForallT a constraints (AppT (AppT ArrowT
                                       (ConT (mkName "RFIHandle")))
                                 $ polysig_help t)
polysig (ForallT _ _ _) =
    error (name ++ ":␣type␣constraints␣are␣not␣possible␣with␣RFI")
polysig t = AppT (AppT ArrowT
                       (ConT (mkName "RFIHandle")))
                 $ polysig_help t

-- add the IO to the return type
-- or if it already is an IO function, leave the return type unchanged
polysig_help :: Type -> Type
polysig_help (AppT (AppT ArrowT t1) t2) = AppT (AppT ArrowT t1)
                                               $ polysig_help t2
polysig_help t@(AppT (ConT a) x) | (showName a) == "GHC.Types.IO" = t
```

```
polysig_help x = AppT (ConT (mkName "IO")) x
```

Listing 3.15: stub signature

In Listing 3.15 we see the code that generates the stub's signature. If we remember the basic idea described in 3.1 the signature has to be modified at two places. First an extra argument of type *RFIHandle* has to be supplemented and second if it is not already an *IO* function the return type has to be wrapped inside the *IO* monad. The rest of the signature remains unchanged. As mentioned before Haskell code is represented in the abstract form of syntax trees using Haskell data types in Template Haskell. The *polysig* function takes the representation of the original function's signature and returns the modified signature of the client stub. Two of the patterns (the first and third) check if the function is a valid RFI function, meaning that it is not a higher-order-function or requires type constrains. The definition of *isHigherOrderFunc* is left out in the listing. It uses the type signature and pattern matching to search for function signatures embedded in the signature and returns *True* if it finds any otherwise it returns *False*. The second pattern matches polymorphic functions. The first two arguments of *ForallT* being the type variables and the type constraints and the third the argument and return types. In this case we first use the list of type variables (a) to obtain the *Name*s and then generate the needed *Mono* class constraints. *Polysig_help* is used to wrap the function's return type into the *IO* monad if it is not already an *IO* function. We return the modified signature and also add the *RFIHandle* argument at the front. The last pattern matches monomorphic functions, so we do not have to do as much as for polymorphic ones. We simply prepend the *RFIHandle* argument and use *polysig_help* again for the return type. *Polysig_help* has to modify the return type. As we have seen in Listing 3.14 the signatures are represented in a tree like manner using the *AppT* constructor. We have also seen that the second parameter can always be seen as an parameter to the first. Thus we can deduce that at the top level the function signatures must map against *AppT (AppT ArrowT t1) t2*, where *t1* represents the function's first argument type. In this way we can construct the new signature by leaving the first part unchanged and simply modify *t2* if necessary. We achieve this with a recursive call to *polysig_help*. When we reach the return type it is either already of type *IO* _, which the second pattern matches and in which case we leave the type unchanged, or it has any other type and we wrap it in the *IO* monad. In either case our modified function signature is complete.

3.6.2. Stub generation

For every RFI function we have to generate a remote function. This function has to join all its n arguments into a single n-tuple, send the request to the server with *query* and the de-serialize the return value.

```
FunD stubname [Clause [VarP handle]
                      (NormalB (genStub (siglist monoSig) 0 []))
                      []]

stubname handle = result_of_genStub
```

Listing 3.16: stub definition

In Listing 3.16 the basic stub definition is shown. *FunD* defines a new function with name *stubname*. The list of *Clauses* represents multiple possible function definitions. In this case only one function definition is created. The function takes one argument (*handle*), which is necessary for all remote functions and the functions body is created by *genStub*. Because the number of arguments for the stub is defined by the original function (plus the one argument for the handle) and not fixed for all stubs, *genStub* generates an appropriate chain of lambda expressions.

```
genStub :: [Type] -> Int -> [Exp] -> Exp
genStub [t] n acc =
  let parse_arg = mkName "p"
  in DoE [BindS (VarP parse_arg)
                (AppE (AppE (AppE (VarE (mkName "query"))
                                  (LitE (StringL name)))
                            (VarE handle))
                (TupE (reverse acc))),
          LetS [ValD (VarP (mkName "r"))
                           (NormalB (SigE (AppE (VarE (mkName "read"))
                                                (VarE parse_arg))
                                          (parseSig t)))
                           []],
          NoBindS (AppE (VarE (mkName "return"))
                        (AppE (VarE (mkName "fromMono"))
                                    (VarE (mkName "r"))))]
genStub types n acc =
  let x = (mkName $ "x" ++ (show n))
  in LamE [VarP x] (genStub (tail types)
                            (n + 1)
                            $ (SigE (AppE tomono (VarE x))
                                    (head types)) : acc)
```

Listing 3.17: generate stub

The *genStub* function takes three arguments, a list of types of the arguments and as last element the type of the return value, a counter to create distinct local variables and a list of expressions. This list is used as an accumulator to store the already generated code parts. Looking back at our protocol (3.2) we remember that we have to put all n arguments of the function into one n-tuple. We can solve this by recursively calling *genStub* each time generating one new argument and pushing the call to *toMono* onto the accumulator list. When we reach the last element of the type list we reached the return type and can put the actual body together. The second pattern generates a lambda expression with one parameter and its body being generate with a recursive call to *genStub*. This generates a chain of lambda expressions. The accumulator list is used to store the parts that make up the n-tuple. Thus each entry corresponds to one element in the tuple. The first pattern matches, if only one element in the type list is left. This means that we reached the type of the return value and we can put everything together. We generate a *do*-statement with three lines. Firstly we use *query* to send the request and receive the result. As shown earlier (Listing 3.13) *query* expects a tuple

with the arguments as its third parameter. Since we already have all the tuples entries accumulated as list (*acc*), we can directly generate the tuple from that list using the *TupE* constructor, which takes a list of elements as argument. In our case we use the reversed list, since our protocol specified that the tuple's i-th element correlates to the functions i-th argument and our list list has the reversed order. Secondly we de-serialize the return value with read and lastly we use *fromMono* before we return the result, this ensures that if our return type is polymorphic the right type is restored. Listing 3.18 shows the resulting code for a simple add function (also defined in that Listing).

```
add : Int -> Int -> Int
add = (+)

remote_add h
      = \ x0
          -> \ x1
               -> do { p <- query "add" h (toMono x0 :: Int,
                                           toMono x1 :: Int);
                       let r = read p :: Int;
                       return (fromMono r) }
```

Listing 3.18: remote_add example

3.7. Server framework

The server side has to fulfill the task of opening a port and listen for incoming function calls. These the server has to process and return the result to the client. To do this the server has to map the message it receives to the right function. For this it first has to extract the function name from the message. We than use a simple *case of* to map that name to the corresponding function. Since the *case of*'s matches depend on what functions we want to use with RFI we have to generate that part with Template Haskell. To start an RFI server we provide *startServer :: Int -> IO ()*. This function takes a port number which it should listen on, opens it and then processes any incoming messages.

3.8. Server functions

The core of our server function is a function map of type *String -> (String -> IO String)* which maps the function names to a corresponding generated wrapper function. These wrapper functions are generated lambda expressions which take the serialized argument tuple de-serialize it and then return the serialized result. Since we allow *IO* functions to be used with RFI all other functions also are transformed into an *IO* function to gain a consistent type signature.

To generate the wrapper function we first have to generate a list of variable names to de-serialize the argument tuple.

```
-- list of names for the arguments
arguments :: [Name]
arguments = argumentsFromSig sig 0
```

```
-- generate the list from the functions signature
argumentsFromSig :: Type -> Int -> [Name]
argumentsFromSig (AppT (AppT ArrowT _) t) n =
  (mkName $ "x" ++ (show n)) : (argumentsFromSig t $ n + 1)
argumentsFromSig _ _ = []
```

Listing 3.19: generate argument variables

As shown in Listing 3.19 we again use the functions signature to generate the list. Pattern matching against the signature allows us to count the number of arguments and generate a list of locally unique *Name*s.

```
-- generate the type of the tuple
tupleType :: Type
tupleType = tupleType_help (siglist sig) (TupleT (length arguments))
  where
    tupleType_help [t] acc = acc
    tupleType_help (t:ts) acc = tupleType_help ts $ AppT acc t
```

Listing 3.20: generate the tuple's type

Listing 3.20 shows another important part of generating the server side code. To correctly de-serialize the argument tuple we have to provide a complete type definition of the tuple. Since polymorphic arguments are all converted to *MonoRep* before serialization we have to use the monomorphic type signature of the RFI function. Meaning the type signature, in which all type variables are replaced by *MonoRep*. This type signature is necessary as soon as we have more than one argument, even if the original function is monomorphic or the original function is polymorphic. To make our life easier we always generate the type signature, independent of the tuple size and the function type. *Siglist* converts a function signature into a list of its arguments' types, but also includes the return type as the last element, which is why we have to stop the recursion when only one element is left.

```
-- create the body of the Match
-- arguments: list of names for the arguments
--            accumulator
matchBody :: [Name] -> Exp -> Exp
matchBody [] acc = (body acc)
matchBody (x:xs) acc = matchBody xs (AppE acc (VarE x))

body :: Exp -> Exp
body x = if isIOFun sig
         then DoE [BindS (VarP (mkName "tmp")) x,
                   NoBindS (AppE (VarE (mkName "return"))
                                 (AppE (VarE (mkName "show"))
                                       (VarE (mkName "tmp"))))]
                        else (AppE (VarE (mkName "return"))
                                   (AppE (VarE (mkName "show")) x))
```

Listing 3.21: creating the server function's body

MatchBody and *body* are the two functions, which generate the actual call to the original function. First we use *matchBody* with the list of argument variables and the

function we want to call. It then generates the abstract syntax tree, which applies the function to the arguments. At last *body* is used to serialize the result and if necessary wrap the result into the *IO* monad.

```
Match (LitP (StringL match))
      (NormalB (LamE [VarP (mkName "x")]
                     (LetE [ValD (TupP argumentTuple)
                                 (NormalB (SigE (AppE (VarE (mkName "read"))
                                                      (VarE (mkName "x")))
                                                tupleType))
                                 []]
                     (matchBody arguments (VarE funName)))))
      []
```

Listing 3.22: generate match

Listing 3.22 is using all this information to generate the match we need for the *case of*. The function name is used as match pattern and as body we generate a lambda expression with one argument. The body of the lambda expression is a simple *let*, which is used to de-serialize the arguments and then call the original function.

In Listing 3.23 we show what the server generates for our *add* example.

```
"add"
  -> \ x
    -> let (x0, x1) = read x :: (Int, Int)
       in return (show (add x0 x1))
```

Listing 3.23: server code for add

4. Conclusion

In this chapter we want to discuss our solution, what it can and can not do and compare it to another approach and look for future improvements.

4.1. Performance

Our approach has several drawbacks related to performance. The first being our protocol. We use string representations for serialization. For most data types this is not an efficient method of serialization, but on the other hand it allows for an easier debugging. For all basic data types the string representation of *show* is an easily readable format. So we can debug by reading the messages or even sending our own to the server, for example with telnet. A second drawback is, that since we somehow have to send the arguments of the function to the server, we first have to serialize them. More importantly to do that we first have to evaluate all arguments. We have the same disadvantage again on the server side. The client expects the server to return the result of the function. To do that we have to evaluate the function. This means that with our remote functions we loose Haskell's lazy evaluation.

Here multiple improvements may be possible. The protocol can be improved by using another serialization method at the cost of readability. A form of byte representation comes to mind, an example for that would be the *ByteString* type from *Data.Serialize*. The second problem is more complicated. One could try to make the protocol more complex and allow the server to send requests back to the client and enable the server to ask the client for specific arguments. In this way, the client would only send which function it wants to call and then wait for the server to specify which arguments are to be send. Important for this method would be that the server can make multiple independent requests per function. Meaning if we have a function like shown in Listing 4.1, the server could ask for the first argument and then after evaluation of the condition it can request either the second or third argument, depending on which branch is taken.

```
lazy :: Bool -> Int -> Int -> Int
lazy True a _ = a
lazy _ _ b = b
```

Listing 4.1: a function that would benefit from lazy evaluation

The problem with this approach is that not only would we have to extend our protocol, but also modify the original function, to request the arguments as needed.

4.2. Usage

A main goal of our solution was to make generating remotely callable functions as simple as possible. With only one function to call to generate the remote functions and server function (*genRFI*), one function to start the server (*startServer*) and one function to establish a connection from the client (*startRFI*) our statically generated function set is very small. To use RFI functions from the client the user simply has to use the corresponding remote function with an additional *RFIHandle* argument. In this way the user does not need to specify anything about the network layers or data serialization, which enables an easy usage.

4.3. Code Base

Normally one would want to generate two modules, one which contains all functions to use as server and one for all client functions. Contrary to that our approach only generates a single module, because of the usage of Template Haskell (section 2.1). This means that the same module can be used both to program a server and a client. So if we want to use RFI to create an API for a server application we also have to publish the server functions.

4.4. Polymorphic Functions

To make some polymorphic functions usable with RFI we defined a new type *MonoRep* and used this type to convert all polymorphic functions into monomorphic ones (section 3.4). This enables us to use all polymorphic functions as long as they are not based on type constraints or are higher-order-functions. In both cases the problem is, that we have no way of de-serializing the data on the server. We have no possibility to de-serialize types, from which we only know they implement a specific type class. A classic example would be the type class *Show*. We have instances for *Show* for both *String* and *Int*, but they are two completely different types and we can not serialize and de-serialize them in the exact same way. Since we want to provide a polymorphic remote function, we do not know anything about the polymorphic type variables at the time of compilation. The same issue we have with higher-order-functions. We are not able to serialize and de-serialize an arbitrary function, to use it on the server.

In 2011 Jeff Epstein, Andrew P. Black and Simon Peyton-Jones presented a paper ("Towards Haskell in the Cloud") [3] in which they also have to solve similar problems. Their main focus is on providing a message passing communication model. In their model they found a way to serialize closures over the network. The main drawback of their solution is that to pass messages with Cloud Haskell you have to use *Channel*s. Each channel can only pass along a single data type and the type that is passed has to be specified at compile time. This makes a usage of their solution for our framework impossible since we do not know the type at compile time and want to provide polymorphic remote functions. Our approach enables us to provide that interface but at the cost of not

allowing constructs like closures. And since we do only loose the exact type information on the server side we can work with arbitrary functions meeting our restrictions. Simple examples for this are *length :: [a] -> Integer* and *append :: [a] -> [a] -> [a]*, which can be used with RFI, since no data is lost during serialization.

The support for polymorphic functions is another item, that might be further improved. For example one could try to support higher-order-functions, by allowing callbacks to the client. In this approach the client would not serialize the function argument, but send some kind of callback identifier. The server then has to ask the client to calculate that function when it is needed. For an example let us take a look at the *map :: (a -> b) -> [a] -> [b]* function. We would still transform it into a monomorphic function on the server side, but also replace the function argument with a placeholder (let us call that type *Function*). The resulting signature for the server function would be *Function -> [MonoRep] -> [MonoRep]* and from the server perspective Function represents a function of the type *MonoRep -> MonoRep*. The server then would have to use the client to calculate the result of *Function* for each element of the list. Since the client does know the original type it can convert the *MonoRep* argument back to its original type, call the function and return the *MonoRep* representation of the result to the server. This approach would require bidirectional communication between client and server depending on the function could dramatically increase the network traffic. In our *map* example the server would have to make n callbacks for a list of n elements. It would also be necessary to modify the original function to make those callbacks. On the other hand it would provide higher-order-function support and would not make the usage of RFI more complicated.

4.5. Conclusion

All in all we accomplished the goal of creating an easy to use library for remote functions. The main drawbacks are, that higher-order-functions and lazy evaluation are not possible with Remote Function Invocation. But on the other hand we can retain polymorphic functions, as polymorphic and by means of generating the serialization functions section 3.4 automatically at compile time for algebraic and basic data types, polymorphic RFI functions work with a variety of data types out of the box.

A. Installation and usage

A.1. Installation

The installation of our RFI module can be done with Cabal [1]. The following instructions assume that you have a Linux based system and the source archive. For instruction on other Operating Systems, take a look at the cabal documentation [2]. To install the package you first have to unpack the source code. Under Linux this can be done with:

```
tar -xvf RFI-version.tar.bz2
```

Now you have to change into the package directory.

```
cd RFI-version
```

After that three more commands are necessary the third has to be executed as *root* if you want to make the package available for all users. One way to do this is prepend the *sudo* command to the line.

```
cabal configure
cabal build
cabal install
```

A.2. Testing the installation

The RFI package includes three example applications. A server and the corresponding client and a simple chat application, which is both server and client. These can also be used to test the RFI installation. Simply change into the *exampels/* directory, which contains the applications. It also includes a *makefile*. So a simple

```
make
```

should build all three applications. If you run these, make sure you start the servers before the client. Especially when you start the chat, you have to switch to a second instance after specifying the server port or use the same port number twice and so connect to the same instance.

A.3. Usage

To use RFI you have to create one module which contains all RFI functions. This module you can then in turn use to write your client and server applications. Since we need a

[1] http://www.haskell.org/cabal/
[2] http://www.haskell.org/haskellwiki/Cabal/How_to_install_a_Cabal_package

lot of compiler flags to generate the RFI functions it is recommended that you copy *rfi_template.hs* from the packages base directory to your working directory and use it as a starting point. It already contains all necessary compiler flags and includes the RFI module. You basically have to do only three things to generate the RFI functions. Set a module name, define the functions you want to use with RFI and add these functions' names to the list of *genRFI*'s argument. You can then compile the module and include it in your client and server applications. To use your functions with RFI, the server application has to call *startServer* with an appropriate port number and the clients have to establish a connection with *startRFI* and the host and port number of a running server as arguments. After that you can use the generated functions. The functions use the original names, but preceded by *remote_*. As mentioned before in section 3.1, these functions take the *RFIHandle* returned by *startRFI* as their first parameter. If not sure what to do, take a look at the examples in Appendix B.

B. Examples

B.1. Specifying RFI functions

```
{-# LANGUAGE
    MultiParamTypeClasses
  , FlexibleInstances
  , OverlappingInstances
  , FlexibleContexts
  , NoMonomorphismRestriction
  #-}
module Test where

import RFI

add :: Int -> Int -> Int
add = (+)

echo :: String -> String
echo = id

inc :: Int -> Int
inc = (+1)

fac :: Int -> Int
fac n | n == 0 = 1
      | otherwise = n * fac (n - 1)

put :: String -> IO ()
put = putStrLn

data Test a = Test a a | Test2 a
        deriving (Show, Read)

f_test :: Test a -> a
f_test (Test _ a) = a
f_test (Test2 a) = a

f_either :: Either Int a -> Either a Int
f_either (Left n)  = Right (n+1)
f_either (Right x) = Left x

f_maybe :: Maybe a -> Maybe a
f_maybe = id

lengthPlusX :: [a] -> Int -> Int
lengthPlusX z x = length z + x
```

```
f_map :: [a] -> (a -> b) -> [b]
f_map = \x y -> map y x

f_tuple :: (a, b) -> a
f_tuple = fst

-- generate RFI - Code

genRFI ["f_test", "echo", "inc", "add",
        "fac", "put", "lengthPlusX",
        "f_maybe", "f_either", "f_tuple"]
```

Listing B.1: RFI functions (examples/usage.hs)

B.2. Starting a server

```
import Test

main = do startServer 55556
```

Listing B.2: server (examples/server.hs)

B.3. Using remote functions

```
import Test
import Monomorph

main = do
  handle <- startRFI "localhost" 55556

  remote_echo handle "test␣echo" >>= putStrLn

  remote_inc handle (2 :: Int) >>= putStrLn . show

  remote_add handle (3 :: Int) (2 :: Int) >>= putStrLn . show

  remote_fac handle (3 :: Int) >>= putStrLn . show

  remote_put handle "test␣print"

  remote_f_either handle (Left 2 :: Either Int String)
      >>= putStrLn . show

  remote_f_either handle (Right "Test" :: Either Int String)
      >>= putStrLn . show

  remote_lengthPlusX handle ([1, 2, 3] :: [Int]) (2 :: Int)
      >>= putStrLn . show

  remote_f_maybe handle (Just "test") >>= putStrLn . show

  remote_f_test handle (Test (2 :: Int) (3 :: Int))
```

```
      >>= putStrLn . show

  remote_f_tuple handle (2 :: Int, 3 :: Float) >>= putStrLn . show
```

Listing B.3: client (examples/client.hs)

B.4. A simple chat

```
import Test
import Control.Concurrent

main = do
  putStrLn "port␣number␣(incoming):␣"
  tmp <- getLine
  forkIO $ startServer (read tmp)
  putStrLn "port␣number␣(outgoing):␣"
  tmp <- getLine
  handle <- startRFI "localhost" (read tmp)

  clientloop handle
  where
    clientloop handle = do
      msg <- getLine
      remote_put handle msg
      clientloop handle
```

Listing B.4: simple chat application (examples/chat.hs)

Bibliography

[1] Epstein, J.; Black, A. P.; Peyton-Jones, S.: *Towards Haskell in the Cloud*, 2011

[2] O'Sullivan, B.; Stewart, D.; Goerzen, J.: *Real World Haskell*, O'Reilly Media, 15.05.2009

[3] *Cloud Haskell*, http://www.haskell.org/haskellwiki/Cloud_Haskell, 24.09.2013

[4] *Language.Haskell.TH*, http://hackage.haskell.org/packages/archive/template-haskell/2.8.0.0/doc/html/Language-Haskell-TH.html, 24.09.2013

[5] *Template Haskell*, http://www.haskell.org/haskellwiki/Template_Haskell, 24.09.2013

[6] *Multi-parameter type class*, http://www.haskell.org/haskellwiki/Multi-parameter_type_class, 24.09.2013

[7] *Java Remote Method Invocation API*, http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html, 24.09.2013

[8] *Java RMI Tutorial*, http://www.eg.bucknell.edu/ cs379/DistributedSystems/rmi_tut.html, 24.09.2013

[9] *The Java Tutorials, Trail: RMI*, http://docs.oracle.com/javase/tutorial/rmi/index.html, 24.09.2013