Electronic Notes in Theoretical Computer Science 41 No. 1 (2001)

URL: <http://www.elsevier.nl/locate/entcs/volume41.html> 29 pages

Lambada, Haskell as a better Java

Erik Meijer

[emeijer@meijcrosoft.com](mailto:emeijer@meijcrosoft.com)

Sigbjorn Finne

[sigbjorn@galconn.com](mailto:sigbjorn@galconn.com)

**Abstract**

*The Lambada framework provides facilities for fluid interoperation between Haskell (currently both Hugs and GHC using non-Haskell98 extensions) and Java. Using Lambada, we can call Java methods from Haskell, and have Java methods invoke Haskell functions. The framework rests on the Java Native Interface (JNI). The Lambada release includes a tool for generating IDL from Java .class ﬁles (using reflection), which is fed into our existing HDirect to generate Haskell-callable stubs.*

# Introduction

It goes without saying that the ability to interact with other languages is of vital importance for the long-term survival of any programming language, in particular for niche languages such as Haskell or ML. Java is an interesting partner for interoperation for a number of reasons:

* Java comes with an large set of stable, and usually well-documented and well-designed libraries and APIs.
* The interaction with Java via the Java Native Interface (JNI) [8,4] effec- tively allows us to script the underlying JVM. This makes bidirectional interoperability with Java very flexible and vendor independent.
* Because of Java’s platform independence, we hope that Lambada gains wider acceptance than our previous work on COM.

Notwithstanding all these advantages, using JNI in its most primitive form is tedious and error prone. Using raw JNI is like assembly programming on the JVM. *Our mission is to make interoperation between Java and Haskell via JNI as convenient as programming in Java directly*.

*◯*c *2001 Published by Elsevier Science B. V.* Open access under CC BY-NC-ND license.

The architecture of Lambada borrows heavily from our previous work on inter- facing Haskell and COM [6,2,3], in particular the binding for Automation [7]. We refer the reader to those papers for a discussion on the rationale behind the object encoding used for Lambada.

# The Java Native Interface

Simply said, JNI provides a number of invocation functions to initialize and instantiate a Java Virtual Machine (JVM), and defines a COM interface to the JVM (the JavaVM interface), and to individual threads that run inside a particular JVM (the JNIEnv interface). The JNIEnv interface contains roughly 230 methods to interact with objects that live in its associated thread.

* 1. *Calling C From Java and vice versa*

To explain the way C and Java interact via JNI we use a slightly perverse version of Hello World! where a Java method calls a C procedure that calls back into Java to print Hello World!.

To use foreign functions in Java we declare a method as native. Class Hello imports a native method cayHello from the HelloImpl DLL 1 , and defines a normal static Java method sayHello that will print the greeting on the standard output:

class Hello {

native cayHello (); static void sayHello () {

System.out.println("Hello World!");

}

static {

System.loadLibrary ("HelloImpl");

}

}

In Java we cannot just call entries of an arbitrary DLL; Java dictates how native methods should look like. For this example HelloImpl.dll should have entry Java\_Hello\_cayHello with signature 2 :

JNIEXPORT void JNICALL

Java\_Hello\_cayHello ([in]JNIEnv\* env, [in]jobject this)

1 A dynamic-link library (DLL) is an executable file that acts as a shared library of functions on the Win32 platform.

2 Throughout this paper we specify the C signatures for the different JNI procedures using

IDL attributes such as [in] and [out].

In general, each native method implementation gets an additional JNIEnv argument, and a jobject object reference (or a jclass class object for static methods). The JNIEnv argument is an interface pointer through which the native method can communicate with Java, for instance to access fields of the object, or to call back into the JVM. The jobject (or jclass) reference corresponds to the this pointer (or class object) in Java.

Procedure Java\_Hello\_cayHello first gets the class object for the Hello class, then gets the methodID for the static void sayHello() method, and finally calls it:

jclass cls = (\*env) -> GetObjectClass (env, this); jmethodID mid = (\*env) ->

GetStaticMethodID (env, cls, "sayHello" "()V"); (\*env) -> callStaticVoidMethod (env, cls, mid);

In the above example the initiative came from the Java side. It is also possible to start on the C side.

The JNI\_CreateJavaVM function takes an initialization structure and loads and initializes a JVM instance. It returns both a JavaVM interface pointer to the JVM and a JNIEnv interface pointer to the main thread running in that JVM instance:

jint JNI\_CreateJavaVM

([out]JavaVM\*\* v, [out]JNIEnv\*\* e, [in]JavaVMInitArgs\* a);

To call sayHello in this manner, we first load and initialize a JVM us- ing JNI\_CreateJavaVM to obtain a JNIEnv pointer, then call the sayHello method exactly as we did before except that we get the class object via the FindClass function and finish by destroying the JVM:

JNIEnv\* env;

JavaVM\* jvm;

JavaVMInitArgs args;

new\_InitArgs (&args); JNI\_CreateJavaVM (&jvm, &env, &args);

jclass cls = (\*env) -> FindClass (env, "Hello"); jmethodID mid = (\*env) ->

GetStaticMethodID (env, cls, "sayHello" "()V"); (\*env) -> callStaticVoidMethod (env, cls, mid); (\*jvm) -> DestroyJavaVM(jvm);

Even though we have left out many details (including dealing with errors) it is clear that interacting with Java at this level of abstraction is quite mind numbing. In particular, the fact that we have to thread around the JNIEnv pointer in every call is something we would like to get rid of. In addition,

encoding of the result-type in method calls (such as callStatic*Void* Method) and the distinction between static and instance methods is something we want to hide to the user.

# Essential FFI in Four Easy Steps

We will introduce some basic JNI concepts that underpin our JNI binding for Haskell while reviewing the basics of the Hugs/GHC foreign function interface. For a more thorough explanation of the FFI we refer to our ICFP99 paper [2].

* 1. *Importing Function Pointers*

In order to call a foreign function from Haskell, we import externally defined functions into Haskell, either by static linking or by dynamic linking.

For example, we can statically import the LoadLibraryA entry of the Microsoft Windows kernel32 DLL

HMODULE LoadLibraryA([in,string]char\*);

by giving the following foreign import declaration:

type CString = Addr type HMODULE = Int

foreign import stdcall "kernel32"

"LoadLibraryA" primLoadLibrary :: CString -> IO HMODULE

This foreign declaration defines a Haskell function primLoadLibrary, that takes the address of a null terminated string which contains the filename of an executable module, dynamically loads the module into memory and returns a handle to the requested module.

As its name suggests, function primLoadLibrary is rather primitive. Us- ing the functions marshallString and freeCString we can easily write a friendlier version that takes a normal Haskell string as its argument:

marshallString :: String -> CString freeCString :: CString -> IO ()

loadLibrary :: FilePath -> IO HMODULE loadLibrary = \p -> do{

s <- marshallString p; h <- primLoadLibrary s; freeCstring s;

return h;

}

The kernel32 DLL also exports a function GetProcAddress that given a mod- ule handle and a pointer to a null-terminated string containing the desired function name returns the address of that function.

FARPROC GetProcAddressA ([in]HMODULE, [in,string]char\*);

To use function GetProcAddr from Haskell we declare the following foreign import:

type FARPROC = Addr

foreign import "kernel32" "GetProcAddressA" primGetProcAddress :: HMODULE -> CString -> IO FARPROC

Again we can easily construct a Haskell-friendly version of

primGetProcAddress that takes a normal Haskell string as its argument:

getProcAddress :: HMODULE -> String -> IO FARPROC getProcAddress = \h -> \n -> do{

pn <- marshallString n;

pf <- primGetProcAddress h pn; freeCString pn;

return pf;

}

The explicit freeing of CStrings is already getting tedious but don’t despair; in section 3.2 we will instruct the Haskell garbage collector to do this auto- matically.

When interfacing with Java using JNI, we have to deal with different JVM implementations and different versions of the same JVM.Therefore, we don’t want to commit ourselves to a particular VM or version too early. The loadLibrary and getProcAddress functions together with *dynamic* foreign function import allow us to delay the choice of a particular JVM to the very last moment.

The dynamic variation of the foreign import declaration defines a function that may be used to unmarshall an external function pointer into a Haskell function.

We can use a function pointer that represents the JNI\_CreateJavaVM DLL en- try by declaring the dynamic foreign import mkPrimCreateJavaVM.This func- tion will coerce the low-level function pointer into the corresponding Haskell function:

type JavaVM = Addr type JNIEnv = Addr

foreign import dynamic mkPrimCreateJavaVM

:: FARPROC -> IO (JavaVM -> JNIEnv -> Addr -> IO Int)

To write a flexible version of createJavaVM we need several helper functions. Function newJavaVMInitArgs creates a default JVM initialization structure; it comes with a corresponding free routine freeJavaVMInitArgs:

newJavaVMInitArgs :: IO Addr freeJavaVMInitArgs :: Addr -> IO ()

Function newResultAddr allocates space for a return value; it also comes with a corresponding free routine freeResultAddr:

newResultAddr :: IO Addr freeResultAddr :: Addr -> IO ()

And finally, a function deref to shorten a pointer indirection:

deref :: Addr -> IO Addr

Using these helper functions, we can complete the tedious code for

createJavaVM:

type JavaVM = Addr type JNIEnv = Addr

createJavaVM :: FilePath

-> IO (JavaVM, JNIEnv)

createJavaVM = \p -> do{ h <- loadLibrary p;

a <- getprocAddress h "JNI\_CreateJavaVM"; pArgs <- newJavaVMInitArgs;

pJavaVM <- newResultAddr; pJNIEnv <- newResultAddr;

mkPrimCreateJavaVM a pJavaVM pJNIEnv pArgs; freeJavaVMInitArgs pArgs;

jvm <- deref pJavaVM; freeResultAddr pJavaVM; jnienv <- deref pJNIEnv; freeResultAddr pJNIEnv; return (jvm,jnienv);

}

There are numerous other situations where we want to call dynamic function pointers. For instance both the JavaVM and the JNIEnv pointers are a double indirection to a table of function pointers (a *vtable*). The function deref we used earlier and the function getProcAt t i that returns the i-the function

pointer in table t make it quite easy to call vtable entries. We will give an example of this in section 3.3.

* 1. *Interfacing With the GC Using Foreign Objects*

One of the nice things about languages such as Haskell and Java is that the garbage collector silently takes care of freeing unused resources. It seems like a great loss therefore that when we import foreign functions, we import the hassle of explicit resource management as well.

What we would like to do instead is to have a hook into the Haskell garbage collector so that we can tell it how to free external resources once they become garbage. As usual, an extra level of indirection does the job. A *foreign object*

[5] is a smart pointer to the outside world:

data ForeignObject

When constructing a foreign object, we pass it a finalizer action that is called by the Haskell garbage collector once the object becomes unreachable. This gives the object a chance to do its last belch.

new\_ForeignObject :: Addr -> (Addr -> IO ()) -> IO ForeignObject

For example, we can define a smart String marshaller that frees the string automatically once it has become garbage (to avoid confusion, the ‘dumb’ version of marshallString is qualified with prefix Prim):

type CString = ForeignObject

marshallString :: String -> IO CString marshallString = \s -> do{

ps <- Prim.marshallString s; newForeignObject Prim.freeString ps;

}

Note that imported functions can directly take a ForeignObject as argument. Extracting the underlying Addr first, would be rather dangerous as the garbage collector could then free the object before the call is actually made.

* 1. *Exporting Closures*

Life would be rather boring if we could only *import* foreign functions. Things start to become interesting when we can masquerade Haskell functions as plain function pointers and pass those the outside world.

A *static* foreign export declaration tells the Haskell compiler to export a de- clared Haskell function behind a C-callable function interface. For example

(assuming we are on the Win32 platform) we can create a DLL with an entry Reverse that reverses a C string by unmarshalling it, reversing the resulting Haskell string and then marshalling the reversed string back to a C string as follows:

type CString = Addr

foreign export stdcall "Reverse" primReverse :: CString -> IO CString

primReverse =

( unmarshallString ## return.reverse ## marshallString )

The double hash operator ## is just reverse monadic composition: f ## g = \a -> do{ b <- f a; g b}. Later, we will also use the single hash operator #, which is just reverse function application a # f = f a.

Haskell *interpreters*, such as for example Hugs, do not naturally support stati- cally exported functions. Fortunately, we can also *dynamically* export Haskell functions as if they are C function pointers 3 . Dynamically exported Haskell functions are extremely powerful and make programming with callbacks very easy, as the following example demonstrates.

The JNIEnv method RegisterNatives dynamically registers new native meth- ods with a running JVM. The RegisterNatives entry takes a reference to a class object in which the native methods will be registered, and an array of JNINativeMethod structures that contain the names, types and implementa- tions of the native methods:

typedef struct { [string]char\* name; [string]char\* signature; [ptr]FARPROC fnPtr;

} JNINativeMethod;

jint RegisterNatives ( [in]JNIEnv\* env

, [in]jclass clazz

, [in,size\_is(nMethods)] JNINativeMethod\* meths

, [in]jint nMethods

);

To be able to call RegisterNatives in Haskell through a JNIEnv pointer, we must first dereference the JNIEnv pointer, fetch the 215-th entry in the function table and coerce that into a Haskell callable function. Then we pass

3 In section 4, we will show how static exports can be simulated using dynamic exports and a statically exported interpreter component.

the resulting function the required arguments to compute the result:

getProcAt :: Int -> Addr -> IO FARPROC

foreign import dynamic mkRegisterNatives

:: FARPROC -> IO (JNIEnv -> Jclass -> Addr -> Int -> IO Int)

registerNatives :: JNIEnv -> Jclass -> Addr -> Int -> IO ()

registerNatives = \env -> \clazz -> \meths -> \n -> do{ f <- (deref

## getProcAt 215

## mkRegisterNatives

) env;

f env clazz meths n;

}

Suppose we want to register a native function with signature void SayHello (JNIEnv\*, jobject) via the registerNatives function. We can easily construct the required function pointer from a Haskell function with type JNIEnv -> Jobject -> IO () by using function mkSayHello. The latter function is defined using a foreign export dynamic declaration:

foreign export dynamic

mkSayHello :: (JNIEnv -> Jobject -> IO ()) -> FARPROC

Thus mkSayHello is a Haskell function that takes *any* Haskell function of type JNIEnv -> Jobject -> IO () and at run-time creates a C function pointer, that when called will call the exported Haskell function:

sayHello\_ :: FARPROC

sayHello\_ = mkSayHello sayHello

sayHello = \env -> \this -> do{ putStr "Hello From Haskell!"; }

Assuming a function marshallJNINativeMethods that builds an array of JNINativeMethod structures from a Haskell list of name/signature/function pointer-triples

marshallJNINativeMethods :: [(String,String,FARPROC)] -> IO Addr

it is not hard to define a function registerHello that registers the Haskell function sayHello as the native implementation of a Java method void sayHello ():

registerHello = \env -> \clazz -> do{

p <- marshallJNINativeMethods [("SayHello","()V", sayHello\_)]; env # registerNatives p 1 clazz;

}

* 1. *Interfacing With the GC Using Stable Pointers*

When we pass Haskell values (such as dynamically exported functions), to the outside world, we must prevent the Haskell garbage collector from moving them around inside the Haskell heap during a collection. Otherwise, an ex- ternal function that holds on to the value will suddenly point to a completely different value. In addition, since we have no control over how many copies are made of exported objects, the garbage collector cannot automatically free them anymore. Haskell values are therefore hidden behind an extra level of indirection called *stable pointers* [5,9]. For the remainder of this paper we don’t need to understand stable pointers in detail except that we know that they exist behind the scenes.

Now that we have covered the basics of interacting with the outside world and of JNI, we can start looking at the main subject of this paper, the bridge between Java and Haskell via the Java Native Interface (JNI).

# Calling Java From Haskell and Haskell from Java

Calling Java from Haskell and Haskell from Java is in principle no different from calling Java from C and calling C from Java.

Calling Java from Haskell poses no problems as long as we have static and dynamic foreign import. In order to let Java call Haskell, the only provision is that Java can either load a DLL that contains Haskell implementations of the required native methods (to implement native methods directly), or some other DLL that can dynamically register the native methods using the JRegisterNatives JNI entry we have seen in section 3.3.

The *DietHEP* component [11] leverages on the foreign function interface we defined for Haskell in previous papers [3] and allows us to view *any* Haskell module as an ordinary DLL.Clients of DietHEP see no difference (except for the extra flexibility of specifying the calling convention at runtime when using GetProcAddressEx) between using an ordinary DLL via the kernel32.dll or using the DietHEP primitives. Going through the extra level of indirection of DietHEP allows us to abstract from the underlying Haskell implementation (provided of course that it supports the DietHEP interface).

The LoadLibrary function takes the name of a Haskell module (or GHC compiled binary), loads it and returns a *handle* to the module. Function GetProcAddress takes that handle and a function name and returns a ‘foreign export’-ed version of the requested Haskell function.

[dllname("DietHEP.dll")]

module DietHEP { typedef enum {stdcall, ccall} CALLCONV;

HMODULE LoadLibrary ([in,string]char\* n);

FARPROC GetProcAddress

([in]HMODULE m, [in,string]char\* n);

FARPROC GetProcAddressEx

([in]CALLCONV c, [in]HMODULE m, [in,string]char\* n);

};

To register native Haskell methods via DietHEP, we wrap the DietHEP com- ponent into a Java class DietHEP in the expected way:

class DietHEP {

static native int LoadLibrary (String n);

static native int GetProcAddress (int m, String n);

...

}

We also assume that we have a class JNI that reflects (amongst oth- ers) the JNI entry registernatives back into Java as the native method RegisterNatives:

class NativeMethod {

String name; String signature; int fnPtr;

...

}

class JNI {

static native void RegisterNatives (Class c, NativeMethod[] ms);

...

}

Using these two classes, the Hello example of section 2.1 can be written to use the Haskell function Hello.sayHello by first loading the Haskell module Hello that contains the definition of the sayHello function, and then regis- tering the dynamically exported function pointer for sayHello with the Java Hello class:

class Hello {

public static native void SayHello (); static {

int h = DietHEP.LoadLibrary ("Hello");

int sayHello = DietHEP.GetProcAddress (h,"sayHello");

JNI.RegisterNatives ( Hello.class

, new NativeMethod[]{

new NativeMethod ("SayHello","()V", sayHello)

}

);

}

}

# A User Friendly Library

At this stage, we have shown the bare bones of the integration between Haskell and Java. We will make the binding more user friendly in a number of steps. First we will abstract away commonly occurring patterns. Second, we hide the explicit threading of the JNIEnv argument through all JNI related code. Next, we will use overloading to hide the encoding of result-types in JNI methods.

* 1. *Combining JNI calls*

Calling an instance method void Foo() on a Java object obj and a JNIEnv

pointer env is a three step process:

1. Get the class object for obj using the JNI method GetObjectClass: cls <- env # getObjectClass obj
2. Look up the methodID for the method via its name and type description using the JNI method GetMethodID:

mid <- env # getMethodID cls "Foo" "()V"

1. Call the method using the JNI method Call*<t>* Method where *<t>* is the result type of the method, passing the object, the methodID and the actual arguments:

env # callVoidMethod obj mid nullAddr

To call a static or instance method, or to get or set a static or instance field, we have to go through the same steps; fetch a class object, look up a method or field ID based on the name and signature, and eventually perform the de- sired operation. For the sake of presentation however, we will ignore invoking static methods until section 5.8. Space limitations preclude us from explaining (static) field access, but they don’t require new concepts to be dealt with.

Our first step in simplifying interaction with JNI is to combine the basic 3-step JNI call sequence into a family of functions call*<t>* Method. These functions take a Jobject pointer, the name and signature of the method or field, a ForeignObject pointer to an array of arguments, and return a value of *<t>* (where as usual, we qualify the primitive call*<t>Method* with Prim):

call*<t>* Method

:: Jobject -> String -> String -> ForeignObject

-> JNIEnv -> IO *<t>*

call*<t>* Method =

\this -> \name -> \sig -> \a -> \env -> do{ cls <- env # getObjectClass this;

mid <- env # getMethodID cls name mid; env # Prim.call*<t>* Method this mid a;

}

Our next step will be to eliminate the JNIEnv parameter.

* 1. *Hiding JNIEnv*

In previous examples we have seen that most interaction with Java goes via the JNIEnv interface pointer, and that all methods of the JNIEnv interface also take a ‘this’ pointer (or a class object pointer) as an additional argument. Having to thread env around is rather tedious, so we are willing to move a few mountains at this moment to save a lot of work later.

The JNIEnv pointer is only valid in its associated thread, so we cannot put it in a global variable just like that. The JavaVM pointer however *does* remain valid across different threads and we can safely put that in a global variable. To do that, we need to get hold of the JavaVM pointer in which the current thread is running, and we postulate that (current and future) JNI implementations provide some way of doing this. We will follow one of the suggestions 4 of Liang ([8], section 8.1.4) that relies on the fact that the current JVM releases do not support the creation of more than one JVM instance inside a single process ([8], page 254).

The JNI library function getCreatedJavaVMs returns a list of all cur- rently running JVMs and as we argued above, we may safely assume that getCreatedJavaVMs always returns a singleton list containing the single run- ning JavaVM. Hence getCreatedJavaVMs is a pure function and thus we can use unsafePerformIO to create a global ‘variable’ javaVM that contains the pointer to the running JVM:

javaVM :: JavaVM

javaVM = unsafePerformIO $ do{ [javaVM] <- getCreatedJavaVMs; return javaVM;

}

The JavaVM interface entry attachCurrentThread returns the JNIEnv pointer for the current thread. Using global variable javaVM we can now easily and safely fetch the currently valid JNIEnv pointer in any context:

4 The book gives several other suggestions, but this one is the simplest to implement. Another possibility would be to capture the current JVM in the JNI OnLoad event handler.

attachCurrentThread :: JavaVM -> IO JNIEnv

getJNIEnv :: IO JNIEnv

getJNIEnv = javaVM # attachCurrentThread

Now that we can obtain the right JNIEnv in any context, we don’t have to sup- ply the JNIEnv argument to call*<t>* Method we defined previously any more, since we can fetch it when we need it (again we qualify the more primitive version of a function with Prim):

call*<t>* Method

:: String -> String -> ForeignObject -> Jobject -> IO *<t>*

call*<t>* Method = \name -> \sig -> \args -> \this -> do{ env <- getJNIEnv;

env # Prim.call*<t>* Method this name sig args;

}

* 1. *Overloading argument types*

Before we start using overloading to simplify JNI, we have to make a short digression on marshalling. In section 3.1, we assumed that we had a function marshallString :: String -> IO CString that takes a Haskell string and returns a pointer to a null terminated array of characters.

The primitive string marshall function marshallString itself is defined in terms two more primitive functions. Function marshallByRefChar :: Char -> [Addr -> IO ()] returns (in this case a singleton) list of functions that writes a character at the indicated addresses. Function writeAt takes a list of Addr -> IO () functions and a start address and calls each function in the list to write the value at subsequent addresses:

writeAt :: [Addr -> IO ()] -> Addr -> IO ()

To marshall a string, we allocate enough memory to hold the string, generate a list of marshall functions for each character in the string and then write each one into the allocated memory:

marshallString :: String -> IO Addr marshallString = \s -> do{

let cs = s++[chr 0];

p <- malloc $ length cs;

writeAt (concat (map marshallByRefChar) cs) p; return p;

}

The free function for strings simply calls the system function

free :: Addr -> IO (), which deallocates a memory block that was

previously allocated by a call to malloc:

freeString :: Addr -> IO () freeString = \s -> do{ free s; }

In general, we have a family of functions marshall*<t>* , marshallByRef*<t>* ,

free*<t>* for each *<t>* that we can marshall from Haskell to Java:

marshall*<t>* :: *<t>* -> IO Addr marshallByRef*<t>* :: *<t>* -> [Addr -> IO ()] free*<t>* :: Addr -> IO ()

At this moment bells should start to ring and alarms should go off: “ad-hoc overloading”, this is *exactly* what Haskell type classes were invented for!

So we define a new type class GoesToJava with three methods marshall, marshallByRef and free. To be able to overload free as well, we add an extra phantom argument to the free function in the class:

class GoesToJava a where { marshall :: a -> IO Addr;

marshallByRef :: a -> [Addr -> IO ()]; free :: a -> Addr -> IO ();

}

and make all types that can be marshalled from Haskell to Java (Char, Int, Integer, Double, Float, Jobject, String, ....) an instance of the GoesToJava class:

instance GoesToJava *<t>* where { marshall = marshall*<t>* ; marshallByRef = marshallByRef*<t>* ; free = \a -> free*<t>* ;

}

The marshallByRef instances for Double and Integer return a two element list with the second entry being \a -> do{ return () } to ensure that the marshall function will allocate enough memory to hold a double or a long.

We continue to overload GoesToJava for tuples to encode Java methods with zero, or more than one argument. This is impossible if we use curried functions to represent multiple argument Java methods in Haskell.

instance GoesToJava () where { marshall = \() -> do{ malloc 0; }; marshallByRef = \() -> [];

free = \() -> \p -> do{ free p; };

}

instance

(GoesToJava a, GoesToJava b) => GoesToJava (a,b) where {

marshall = \(a,b) -> do{

let ab = marshallByRef a ++ marshallByRef b; p <- malloc $ length ab;

writeAt ab p; return p;

};

marshallByRef = \(a,b) -> marshallByRef a ++ marshallByRef b;

free = \(a,b) -> \p -> do{ (deref ## free) p;

(deref ## free) (incrAddr p); free p;

}

By using the GoesToJava class we don’t have to marshall arguments to a JNI call into an array anymore, but we can simply write (again qualifying the previous version of call*<t>* Method with Prim):

call*<t>* Method :: GoesToJava a =>

String -> String -> a -> Jobject -> IO *<t>*

call*<t>* Method =

\name -> \sig -> \a -> \this -> do{ p <- marshall a;

r <- newForeignObject p (free a); env <- getJNIEnv;

env # Prim.call*<t>* Method this name sig r;

}

* 1. *Overloading result types*

The signature of JNI calls above obviously cries out to be overloaded as well by means of a type class ComesFromJava with an instance for each *<t>* ((), Jobject, Bool, Byte, Char, Short, Int, Integer, Float, Double) that can be returned by a JNI method invocation:

class ComesFromJava b where { callMethod :: GoesToJava a =>

String -> String -> a -> Jobject -> IO b;

}

instance ComesFromJava *<t>* where { callMethod = call*<t>* Method;

}

At this point we have reduced the complexity for JNI calls quite significantly when compared with the most primitive form, but there are still ample possi- bilities to improve. The next thing that we are going to attack is the explicit passing of type descriptor strings, which at the moment is rather unsafe.

* 1. *Generating type descriptors*

The problem with the current version of function callMethod is that there is no connection between the *value* of the type descriptor string, which denotes the Java type of a field or method, and the corresponding Haskell *types* of the argument and result of the callMethod function. For example, the following definition is accepted by the Haskell type-checker:

foo :: () -> Jobject -> IO Int

foo = \() -> \obj -> do{ obj # callMethod "foo" "(I)V" (); }

Unfortunately, it leads to a runtime error because the foo method is called as if it has Java type void foo(int) instead of at the correct type int foo() (which corresponds to the type descriptor string should be "()I").

We will generate type descriptors by a family of functions descriptor*<t>* ::

*<t>* -> String 5 . As we did in the previous sections, we define a Haskell type class, in this case called Descriptor, that witnesses for which *<t>* we can obtain a descriptor:

class Descriptor a where { descriptor :: a -> String; }

instance Descriptor *<t>* where { descriptor = descriptor*<t>* ; }

The type descriptor for the primitive JNI type Bool is "Z": descriptorBool :: Bool -> String

descriptorBool = \z -> "Z"

The other basic types are mapped as follows: Byte *'→* "B", Char *'→* "C", Short *'→* "S", Int *'→* "I", Integer *'→* "L", Float *'→* "F", Double *'→* "D".

Type descriptors for methods arguments are formed by concatenating the type descriptors of the individual arguments:

instance Descriptor () where { descriptor = ""; }

instance (Descriptor a, Descriptor b)

5 This trick is used in the Haskell module Dynamic as well, and is attributed to the legendary hacker Lennart Augustsson.

=> Descriptor (a,b) where {

descriptor = \(a,b) -> descriptor a ++ descriptor b;

}

Type descriptors for methods are formed by enclosing the type descriptor for the arguments inside parenthesis followed by the type descriptor of the result type, with the exception that V is used for the void return type:

methodDescriptor

:: (Descriptor a, Descriptor b) => a -> b -> String methodDescriptor = \a -> \b ->

concat

[ "(", descriptor a ,")"

, if null (descriptor b) then "V"

else descriptor b

]

To deploy function methodDescriptor we need to jump one more hurdle. In order to build the method descriptor string, we need to know the type of the result of a call. Since the methodDescriptor function does not need the value of its arguments to compute a type descriptor, we can recursively build the right method descriptor from a call which will take that descriptor as an argument:

callMethod

:: ( Descriptor a, GoesToJava a

, Descriptor b, ComesFromJava b

) => String -> -> a -> Jobject -> IO b callMethod = \name -> \a -> \this

-> let{

*sig* = methodDescriptor a (unsafePerformIO *mb* );

*mb* = this # Prim.callMethod name *sig* a

} in *mb*

There are several things to note about the above code. First of all, it inti- mately relies on lazy evaluation, function methodDescriptor only depends on the *types* of its arguments and not on their values. Secondly, the call to unsafePerformIO is there solely to strip the IO from the result type of the callMethod function, which is completely safe.

* 1. *Encoding Inheritance using phantom types*

The callMethod function as we have reached by now is not yet as safe and flexible as it could be. In particular, we do not reflect the fact that in Java classes satisfy an inheritance relation. We can however encode inheritance

using the same trick we used for COM components [6].

For the Java root class Object we define a phantom type Object\_ that con- tains the low level untyped Jobject pointer, and immediately hide Object\_ using type synonym Object (we will see in just a minute why):

data Object\_ a = Object Jobject type Object a = Object\_ a

Type Object\_ is called a *phantom* type because its type parameter a is not used in its right hand side.

For every class *<class>* that extends its superclass *<super>* , we define a new (empty) phantom type *<class>* and a type synonym *<class>* that expresses that *<class>* extends *<super>* :

data *<class>* a

type *<class>* a = *<super>* (*<class>* a)

The definition for *<class>* expands to Object (... (*<super>* (*<class>* a)) ...), which precisely expresses the inheritance relationship between Object and *<super>* and *<class>* .

An instance of the Java class *<class>* is represented by a value of type

*<class>* (), which is still the same old Jobject pointer, but now guarded by a little type harness. The nice thing about this is that the Haskell type checker prevents us from using a method of the *<class>* class at an instance for which *<class>* is not a superclass.

Of course, when actually calling a method we use the Jobject pointer that is hidden inside the Object wrapper:

callMethod ::

(Descriptor a, GoesToJava a , Descriptor b, ComesFromJava b)

=> String -> -> a -> Object o -> IO b callMethod = \name -> \a -> \(Object this) ->

Prim.callMethod name a this

* + 1. *Class descriptors*

Using the above encoding of Java classes into Haskell types, we can generate type descriptor strings for Java classes by defining a function descriptor*<class>* for each Java class *<class>* :

descriptor*<class>* :: *<class>* () -> String descriptor*<class>* = \\_ -> "L*<classdescriptor>* ;"

instance Descriptor (*<class>* ()) where { descriptor = descriptor*<class>* ;

}

A *class descriptor* is the fully qualified Java class with all .-s replaced by /-s; for instance the class descriptor for the Java class Object is java/lang/Object.

For methods that take objects as arguments, JNI expects the class descrip- tor for the formal parameter and not of the actual argument. This means for example that if we call the method Component add(Component comp) of the Java class Container with an argument whose run-time class is Frame (a subclass of Container), we still must pass the class descriptor Ljava/awt/Component; instead of Ljava/awt/Frame;. The correct version of add silently casts its argument to a Component () just before making the actual call to function add:

add :: Component a -> Container b -> IO (Component a) add = \comp -> \this -> do{

this # callMethod "add" (cast comp :: Component ());

}

Inside function add we use the function cast :: a -> b to change the first argument of type Component a (which could be any subclass of Component) to exactly Component () so that we get the correct method descriptor. Function add itself remains completely type safe.

The above situation is the only case where we have to explicitly up an upcast in Haskell; the question remains how to deal with the occasional need for downcasting. In Java, it is possible to unsafely cast between arbitrary classes, which might raise a dynamic ClassCastException. Using the Haskell cast to do an illegal arbitrary cast will either result in a ClassCastException or other exception being thrown in Java (which is checked using the JNI entry ExceptionOccured), or a call to one of underlying the JNI entry calls will signal an error (usually by returning NULL). The Haskell wrapper will catch these and propagate the errors to the Haskell level by raising a userError when returning. In other words, using cast to simulate arbitrary Java casting in Haskell is no more safe or unsafe than arbitrary casting in Java.

* 1. *Dealing with interfaces*

Besides classes, Java also supports *interfaces*. The role of interfaces in Java is quite similar to that of type classes in Haskell. Saying that a Java class implements an interface is like making an instance declaration in Haskell.

To deal with interfaces in Lambada we introduce an empty Haskell type class for each Java interface, possibly by extending some one or more base interfaces, for example:

class ImageObserver a class MenuContainer a class Serializable a class EventListener a

class EventListener a => WindowListener a

For each method in an interface, we generate a function that requires its argument to satisfy the class constraint, eg for the MenuContainer interface we define:

getFont :: MenuContainer a => () -> a -> IO (Font ()) getFont = \() -> \this -> do{

this # callMethod "getFont" ();

}

For each Java class *<class>* that implements an interface *<interface>* , we make the corresponding Haskell type *<class>* an instance of the Haskell type class *<interface>* that corresponds to the Java interface *<interface>* . For instance, the Java class Component implements the interfaces ImageObserver, MenuContainer, and Serializable, so in Haskell we write:

instance ImageObserver (Component a) instance MenuContainer (Component a) instance Serializable (Component a)

Note that we have to be careful *not* to write Component (), since every sub- class of Component implements the interfaces that Component implements.

Since Component is an instance of MenuContainer, a call c # getFont () is well-typed whenever c has type Component (), Container (), Window (), Frame (), or any subclass of these.

Some methods expect interface ‘instances’ as arguments:

void addWindowListener(WindowListener)

To deal with this situation, we have to be able to generate descriptor strings for interfaces. For each interface *<interface>* we define a new type

*<interface>* that uniquely identifies that interface and make *<interface>* an instance of Descriptor. Of course an interface satisfies its own interface constraint (in reality, we have to do name mangling because in Haskell data types and type classes are in the same name spaces but here no confusion can arise):

data *<interface>*

descriptor*<interface>* :: *<interface>* -> String descriptor*<interface>* = \\_ -> "L*<interfacedescriptor>* ;"

instance Descriptor *<interface>* where { descriptor = descriptor*<interface>* ;

}

instance *interface <interface>*

Now we can (safely) coerce any object reference that implements a certain interface to that interface and obtain the correct descriptor.

* 1. *Static methods and class objects*

Up to now we have ignored static methods. A static or class method is a method that does not act on a *class* instance, but on the *class object* instance.

Calling a static method, say void Foo(), on a class object cls and given a JNIEnv pointer env is quite similar to calling an instance method, except that the method is not invoked on the object instance but on its class object:

1. Look up the methodID for the static method via its name and type description using the JNI method GetStaticMethodID:

mid <- env # getStaticMethodID cls "Foo" "()V"

1. Call the static method using the JNI method CallStatic*<t>* Method where *<t>* is the result type of the method, passing the *class object*, the methodID and the actual arguments:

env # callStaticVoidMethod cls mid nullAddr

If we continue and mirror the overloading have done for instance methods, we will end up with a function callStaticMethod for invoking static methods on a class object:

callStaticMethod ::

(Descriptor a, GoesToJava a , Descriptor b, ComesFromJava b)

=> String -> a -> Jclass -> IO b

Although in Java we can call a static method on an object (which the com- piler translates to the right class), the recommended style is to call it on the class directly. By *requiring* the latter style, we can distinguish the types of callStaticMethod and callMethod. This enables us to introduce overloading to hide the distinction between calling static and instance methods:

class JNI j where { call

:: ( Descriptor a, GoesToJava a

, Descriptor b, ComesFromJava b

) => String -> a -> j -> IO b;

}

For instance method calls we can immediately call callMethod: instance JNI (Object a) where { call = callMethod; }

For static methods we define a new phantom type ClassObject such that

ClassObject *<class>* uniquely encodes the class object of a Java class

*<class>* . Type ClassObject just wraps a type harness around an untyped

Jclass pointer:

data ClassObject a = ClassObject Jclass

The JNI instance declaration for static methods peels off the type harness and calls callStaticMethod:

instance JNI (ClassObject a) where {

call = \n -> \a -> (ClassObject clazz) -> clazz # callStaticMethod n a;

}

So now we can call static methods on class objects, but how do we obtain class objects in the first place? The JNIEnv entry findClass takes a descriptor for the Java class *<class>* and returns the class object for *<class>* :

getClassObject

:: Descriptor (Object o) => (Object o) -> ClassObject (Object o) getClassObject = \this ->

unsafePerformIO $ do{ env <- getEnv;

clazz <- env # findClass (descriptor this); return (ClassObject clazz);

}

Since function getClassObject does not use its argument to compute the class object, we can define an overload class object ‘constant’ with the help of function stripObject :: Object o -> o:

classObject :: Descriptor (Object o) => ClassObject (Object o) classObject = let{ c = getClassObject (stripObject c) } in c

* 1. *Constructing object instances*

The distinction between instances (subclasses of java.lang.Object) and class objects (subclasses of java.lang.Class) can be confusing at first, especially because we can also call non-static methods on class references. One such example are constructor functions to create class instances.

Without using overloading to generate method signatures, we can combine the steps to create a new object into a single function as follows:

new :: (GoesToJava a) => String -> Jclass -> a -> IO (Object o) new = \sig -> \clazz -> \a -> do{

cid <- env # getMethodId clazz "<init>" sig; p <- marshall a;

r <- newForeignObject (free a) p; o <- env # newObject clazz cid r; return (Object o);

}

As we did before, we want to use function methodDescriptor to generate the method descriptor, and furthermore we want to avoid passing the class object as an argument to new since we can get that using function getClassObject (again we qualify the previous version of new with Prim):

new :: (GoesToJava a , Descriptor (Object o) )

=> a -> IO (Object o) new = \a -> let{

o = unsafePerformIO mo; clazz = getClassObject o; sig = methodDescriptor a o; mo = Prim.new sig clazz a;

} in mo

In practice overloading new on its return type often requires manual resolution using explicit type signatures. For convenience we will therefor define an additional un-overloaded constructor function new\_*<class>* for each <class> that is an instance of Descriptor by restricting the type of new:

new\_*<class>* :: (GoesToJava a) => a -> IO *<class>*

new\_*<class>* = new

# Simulating anonymous inner classes

Haskell functions/actions are represented in Java as objects of class Function, which store the actual exported Haskell function pointer as an int in a private field f and implement a native method call that takes an Object array as argument, and an Object representing the this pointer (or class object) and returns an Object as result:

class Function { private int f;

Function(int f){ this.f = f; } public native Object call

(Object[] args, Object this);

...

}

A typical situation is that we have to create an event-handler that implements the WindowListener interface. To facilitate the construction of such a handler in Haskell, we define a Java class WindowHandler that has a private Function object field for each of its methods which are initialized by the constructor function. Each method in the WindowHandler class simply delegates its work to the appropriate Function object:

class WindowHandler implements WindowListener { private Function activated;

...;

WindowHandler

(Function activated, ...) { this.activated = activated;

...

}

void windowActivated(WindowEvent e){ this.activated.call (new Object [] {e}, this);

}

...

}

The JNI framework dictates that the native implementation of the call method is a function that takes the JNIEnv and the Jobject this-pointer as all native methods do, a Jobject argument that represents the argument array, the explicit this argument of the instance in which the call was made, and that returns a Jobject:

call :: JNIEnv -> Jobject -> (Jobject -> Jobject -> IO Jobject)

Function call uses the this pointer of the Function instance to retrieve the function pointer from the private field f and then calls that as a function of Haskell signature (Jobject -> Jobject -> IO Jobject):

foreign import dynamic importFunction

:: FARPROC

-> IO (Jobject -> Jobject -> IO Jobject)

call = \env -> \this -> \args -> \t -> do{ f <- env # getField this "f"; importFunction f args t;

}

We create Function instances in Haskell by creating a C function pointer for the Haskell function we want to export using exportFunction and then use the Haskell JNI library to call the new\_Function constructor:

foreign export dynamic exportFunction

:: (Jobject -> Jobject -> IO Jobject) -> FARPROC new\_Function :: Int -> IO (Function ())

All this quite low-level and error prone, so let’s use a type class to tidy up and do the grungy work for us.

First of all, we extend the ComesFromJava class with a new member func- tion unmarshallArray that unmarshalls zero or more values from a Jobject pointer representing an Object array to a tuple of corresponding Haskell val- ues, and a new member function unmarshall that marshalls a single Jobject pointer into its corresponding Haskell value:

class ComesFromJava b where { callMethod :: GoesToJava a =>

String -> String -> a -> Object o -> IO b; unmarshall :: Jobject -> IO b; unmarshallArray :: Jarray -> IO b

}

For basic types *<t>* we simply use the unmarshaller for that type, or first get the single element from the array and unmarshall that:

instance ComesFromJava *<t>* where { unmarshall = \o -> do{ unmarshall*<t>* o;

unmarshallArray = \args

args # (getArrayElement 0 ## unmarshall);

}

For pairs, triples, etc, we recursively unmarshall the elements from the array one at a time.

instance

(ComesFromJava a, ComesFromJava b)

=> ComesFromJava (a,b) where { unmarshall = unmarshallArray; unmarshallArray = \args -> do{

1. <- args # (getArrayElement 0 ## unmarshall);
2. <- args # (getArrayElement 1 ## unmarshall); return (a,b);

};

}

Function marshalFun takes a high-level Haskell function of type (ComesFromJava a, GoesToJava b) => (a -> Object o -> IO b), turns it into a low-level function of type Jobject -> Jobject -> IO Jobject and then dynamically exports it using exportFunction:

marshallFun

:: (ComesFromJava a, GoesToJava b)

=> (a -> Object o -> IO b) -> IO FARPROC

marshallFun = \f -> exportFunction $ \args -> \this -> do{ a <- unmarshall args;

t <- unmarshallObject this b <- f a t; marshall b;

}

With the help of marshallFun we can finally make functions an instance of GoesToJava. To marshall a function from Haskell to Java, we first use marshallFun to get its function pointer representation, then we construct a new Java Function object for the pointer and marshall the resulting Java object to obtain a pointer we can throw over to Java:

instance

(ComesFromJava a, GoesToJava b)

=> GoesToJava (a -> Object o -> IO b) where {

marshall = \f -> do{ a <- marshallFun f;

func <- new\_Function (toInt a); marshall func

}

}

Since functions are an instance of GoesToJava, writing event handlers in Java has now become completely painless. We can call the constructor for our WindowHandler example, passing it seven Haskell functions of type Event e -> WindowHandler w -> IO () that handle each of the seven pos- sible Window events:

new\_WindowHandler

( \event -> \this -> do{ this # activated event;

} , ... )

This nearly exactly matches the corresponding code we would write in Java

new WindowHandler () { void activated (Event e) { ...; } ... }

# Tool support

To reiterate the introduction of this paper, Java is an interesting partner for a foreign function interface because there are a lot of Java classes out there. However, even with the layers of abstractions we have introduced on top of

JNI, manually coding the Haskell callable stubs for the Java APIs we are interested in is not always practical.

Lambada provides tool support in concert with HDirect[3]. HDirect is IDL based, so provided we can derive an IDL specification from a Java class or interface, HDirect will take care of generating valid Haskell stubs. We do this by providing a tool that uses the Java Reflection API to generate IDL interfaces (annotated with a couple of custom attributes) corresponding to Java classes or interfaces. Provided with that information, HDirect is then able to automatically generate the Haskell callable stubs we have described in this paper. HDirect is also capable of generating Java callable stubs to Haskell implemented classes/methods.

At the Lambada implementation level, HDirect was also used to generate the low-level JNI stubs outlined in Section 3.

# Related work

Claus Reinke was a remarkably early adaptor of JNI, as early as 1997 he proposed to use JNI to interoperate between Haskell and Java [10]. Reinke’s system provides only one-way interop between Haskell and Java using the basic low-level JNI methods, ie he does not support calling back from Java to Haskell. The most important contribution of our work over his is our sophisticated use of overloading and other advanced language features to hide the idiosyncrasies of JNI.The integration that Lambada offers between Haskell and Java is very close to the tight integration offered by MLj [1], with the important exception that Lambada is implemented completely *within* Haskell, whereas MLj needs many language changes.

# Acknowledgements

We would like to thank Sheng Liang for inviting us explain Lambada inside the lion’s cave at JavaSoft, Martijn Schrage for being a model Lambada beta- tester, and Conal Elliot and Simon Peyton Jones for proofreading the paper during pleasant visits of the first author to Microsoft. S´ergio de Mello Schnei- der suggested the name Lambada.

# References

1. Nick Benton and Andrew Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *Proceedings of ICFP’99*.
2. Sigbjorn Finne, Erik Meijer, Daan Leijen, and Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of ICFP’99*.
3. Sigbjorn Finne, Erik Meijer, Daan Leijen, and Simon Peyton Jones. HDirect: A Binary Foreign Function Iinterface for Haskell. In *Proceedings of ICFP’98*.
4. Rob Gordon. *Essential JNI : Java Native Interface*. Prentice Hall, 1999.
5. Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *Proceedings of IFL’99*, LNCS.
6. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of ICSR5*, 1998.
7. Daan Leijen, Erik Meijer, and Jim Hook. *Haskell as an Automation Controller*. LNCS 1608. 1999.
8. Sheng Liang. *The Java Native Interface*. Addison-Wesley, 1999.
9. Alastair Reid. Malloc Pointers and Stable Pointers: Improving Haskell’s Foreign Language Interface. [http://www.cs.utah.edu/~reid/writing.html,](http://www.cs.utah.edu/~reid/writing.html) September 1994.
10. Claus Reinke. Towards a Haskell/Java Connection. In *Proceedings of IFL’98*, LNCS 1595.
11. Julian Seward, Simon Marlow, Andy Gill, Sigbjorn Finne, and Simon Peyton Jones. Architecture of the Haskell Execution Platform (HEP) Version 6. [http://www.haskell.org/ghc/docs/papers/hep.ps.gz,](http://www.haskell.org/ghc/docs/papers/hep.ps.gz) July 1999.