Qt-style C++ in Haskell

Wolfgang Jeltsch

October 7, 2008

The programming paradigms behind Haskell and C++ are very different. Therefore, accessing a C++ library from Haskell is not straightforward. One has to deal with such C++ features as static and non-static method membership, function overloading and inheritance. In addition, the Qt library uses signals and slots which are not part of standard C++. We present an approach for simulating all these features in Haskell. Our key idea is to represent C++ entities like classes and methods by single-value Haskell types. This allows us to express relationships between these entities using multi-parameter type classes and type families. We are thus able to program in Haskell similarly to how we would program in C++ and enjoy the same compile-time checks. In the case of connecting signals to slots we can even offer checks at compile time where Qt only allows runtime checks.

1 Our goal

Our goal is to provide convenient access to C++ libraries from Haskell. We want to support signals and slots as introduced by Qt. Our goal is not to provide means for full object-oriented programming. All we want is to access a library which is already there and complete. This has the consequence that we do not offer means for forming new subclasses and that we allow access only to public class members. We do not make it possible to emit signals or to connect signals to signals since emitting signals can only be done from inside a class.[1]

2 Basics

Haskell has a class system which is very different from the OO-style class system of C++. Therefore we do not try to map C++ classes to Haskell classes and C++ methods to Haskell methods. Instead, we try to describe the different entities of the C++ library (classes, methods, and so on) by single-value types. Using multi-parameter type classes and type families, we can then express the relationships of these entities with a pure

[1]Actually, emitting a signal means calling a protected method internally.

Haskell library. For example, we can assign a return type and an implementation to every triple of a class, a method name and a list of argument types. This means that we use Haskell as a kind of meta language in which we describe the C++ library in question.

3 The solution in detail

In the following subsections, we present the implementation of a small Haskell library which allows access to object-oriented libraries according to the strategies sketched above. We also mention what kind of code a binding to a concrete C++ library would have to add.

3.1 Classes

As told above, we represent classes as types with a single (non-bottom) value. For each class, an algebraic data type with one nullary data constructor has to be introduced. The name of the data type as well as the name of the data constructor should match the class name used in the C++ library. We call the type as well as its sole value a class ID.

To represent classes in static method calls, we introduce the type *Class*:

```
newtype Class classID = Class classID
```

3.2 References

We access objects always through pointers. This is necessary because we want to bind to C++ in a portable way. Since the Haskell FFI specifies only access to C and PASCAL libraries, we bind to wrappers around the C++ libraries. These wrappers are written in C++ but have a C interface (using **export** "C"). This has the consequence that only values of valid C types can be transfered from C++ to Haskell and back.

We use values of type **char** $*$ for referring to C++ objects in the C interface. However, on the Haskell side, we want to restore type safety. Therefore, we introduce a type *Ref* which is parameterized by a class ID:

```
newtype Ref classID = Ref (Ptr CChar)
```

In the remainder of our library, we will have to calculate the class ID and the C pointer from a value of *Ref*. So we provide the following two helper functions:

```
classID :: Ref classID → classID
classID = error "Calculated class ID cannot be evaluated."
ptr :: Ref classID → Ptr CChar
ptr (Ref cCharPtr) = cCharPtr
```

Note that *classID* always returns $\perp$. This should be no problem since class IDs are not expected to be evaluated. They fulfill their purpose solely at the type level.

3.3 Constructors and destructors

We do not allow to call C++ constructors and destructors directly. Instead, we let programmers call the C++ **new** and **delete** operators from Haskell. For each constructor and each destructor, we need a separate C wrapper function for the respective **new** or **delete** call. To call these wrapper functions consistently, we introduce two type classes:

```
class Constructor classID args where
  new :: classID → args → IO (Ref classID)
class Destructor classID where
  delete :: Ref classID → IO ()
```

The type variable *args* stands for the type of the complete argument list. Empty argument lists are denoted by the unit value, argument lists with a single value by this very value, and argument lists with more than one value by tuples. Since one can put parantheses around a single argument value without changing the meaning of the program, it is possible to write argument lists exactly as one would do in C++

3.4 Methods

There are static methods and instance methods. We want to use a single type class for both kinds of methods. We distinguish between them using single-value data types:

```
data Static = Static
data Instance = Instance
```

Depending on its kind, a method works on different kinds of "entities". Static methods work on classes while instance methods work on class instances. We introduce a class *MethodKind* with an associated type *Entity* so that applying *Entity* to a method kind and a class ID yields the corresponding entity type:

```
class MethodKind methodKind where
  type Entity methodKind :: * → *
instance MethodKind Static where
  type Entity Static = Class
instance MethodKind Instance where
  type Entity Instance = Ref
```

In C++, a method is identified by a class, the method name and the list of argument types. The result type is not used for identifying the method. We introduce a class *Method* whose instances denote concrete C++ methods:

```
class (MethodKind (Kind classID methodID args)) ⇒
        Method classID methodID args where
```

```
type Kind classID methodID args :: ∗
type Result classID methodID args :: ∗
invoke :: Entity (Kind classID methodID args) classID → methodID → args →
          IO (Result classID methodID args)
```

Kind gives the kind of the method (*Static* or *Instance*) and *Result* gives its return type. The *invoke* method calls the C wrapper around the respective C++ method call.

Calling *invoke* directly does not really mimic the C++ syntax of method calls. In C++, a method call consist of a pointer, an arrow and the actual method call when using pointers to denote objects and classes.

The actual method call consists of the method name (usually starting with a lowercase letter) and the argument list without any additional characters inbetween. This looks like a Haskell function application. So we make it possible to use function application on the Haskell side. We introduce a type *Application* which denotes the actual method call by just listing the method ID and the list of arguments:

```
data Application id args = Application id args
```

For every method ID, a function should be introduced which is equivalent to the partial application of the *Application* data constructor to the method ID in question. This function should be named like the methodID, except that the first letter should be lowercase.

For mimicing the method call syntax of C++, we introduce an operator ⇝:

```
infix 0 ⇝
(⇝) :: (Method classID methodID args) ⇒
        Entity (Kind classID methodID args) classID →
        Application methodID args →
        IO (Result classID methodID args)
entity ⇝ Application methodID args = invoke entity methodID args
```

3.5 Slots

Like methods, a slot is identified by its class, its name and its list of argument types. We introduce a class *Slot*:

```
class (Method classID slotID args) ⇒ Slot classID slotID args where
  slotConnectString :: classID → slotID → args → String
```

Each slot can also be used as a method, so *Slot* is a subclass of *Method*. The class method *slotConnectString* yields the result of applying Qt's *SLOT* macro to the string representing the slot. The C wrapper should export the string as a C variable[2] and *slotConnectString* should just access this variable. The value of the *args* argument must not be evaluated. It is just there to specify the types of the arguments.

[2]This is possible since the string is created by macro expansion and therefore known at compile time.

3.6 Signals

Signals are represented by instances of the *Signal* class which is defined as follows:

$$
\begin{aligned}
&\textbf{class } \mathit{Signal\ classID\ signalID\ args} \textbf{ where}\\
&\quad \mathit{signalConnectString} :: \mathit{classID} \rightarrow \mathit{signalID} \rightarrow \mathit{args} \rightarrow \mathit{String}
\end{aligned}
$$

signalConnectString works analogous to *slotConnectString*.

Although signals cannot be called like methods (and therefore slots), a function equivalent to a partial application of *Application* should also be defined for every signal. This will be used when connecting signals to slots as described in subsection 3.8.

3.7 Dropping arguments

When connecting a signal to a slot, it is possible for the signal to have more arguments than the slot. The only precondition for a connection is that the list of slot argument types matches a prefix of the list of signal argument types. We develop a class whose instances show which argument type lists are prefixes of what other argument type lists.

For implementing our type class with little effort, we first introduce a type family *Init*:

$$\textbf{type family } \mathit{Init\ args} :: *$$

Applying *Init* to a list of argument types drops the last argument type.

We first deal with the case of a single argument. This is problematic because we do not use a special type (like a tuple type) for argument lists of length one but the type of the argument directly. The straightforward instance

$$\textbf{type instance } \mathit{Init\ soleArg} = ()$$

does not work because this would also introduce () as the result for all tuple types.[3] Therefore, we explicitly list all single argument cases which may occur. Since we only use reference types and some types corresponding to simple C++ types as argument types, we can use the following declarations:

$$
\begin{aligned}
&\textbf{type instance } \mathit{Init\ Bool} &&= ()\\
&\textbf{type instance } \mathit{Init\ Char} &&= ()\\
&\textbf{type instance } \mathit{Init\ Int} &&= ()\\
&\textbf{type instance } \mathit{Init\ Double} &&= ()\\
&\textbf{type instance } \mathit{Init}\ (\mathit{Ref\ classID}) &&= ()
\end{aligned}
$$

Now we would have to give an instance declaration for every possible tuple type. Since there are infinitely many of them, this is not possible. However, a method has not more

[3]Note that at the moment of writing, overlapping is not allowed for type families. If closed type families become available at some point in the future, *Init* could be a closed type family with a default case for single arguments. Using closed type families would be advantageous anyway because with the current solution, a library user can easily extend the *Init* family and thereby allow signal-slot connections which are not type-correct.

than a small number of arguments typically, so we can limit our list to a reasonable finite size. We give the instance declarations

$$\begin{aligned}
&\textbf{type instance}\ \mathit{Init}\ (\mathit{arg}_1, \mathit{arg}_2) &&= \mathit{arg}_1\\
&\textbf{type instance}\ \mathit{Init}\ (\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3) &&= (\mathit{arg}_1, \mathit{arg}_2)\\
&\textbf{type instance}\ \mathit{Init}\ (\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3, \mathit{arg}_4) &&= (\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3)\\
&\textbf{type instance}\ \mathit{Init}\ (\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3, \mathit{arg}_4, \mathit{arg}_5) &&= (\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3, \mathit{arg}_4)
\end{aligned}$$

and so on.

Now we declare the *Prefix* class and give two instances forming an inductive definition of the prefix property:

$$\begin{aligned}
&\textbf{class}\ \mathit{Prefix}\ \mathit{prefix}\ \mathit{args}\\
&\textbf{instance}\ \mathit{Prefix}\ \mathit{args}\ \mathit{args}\\
&\textbf{instance}\ (\mathit{Prefix}\ \mathit{prefix}\ (\mathit{Init}\ \mathit{args})) \Rightarrow \mathit{Prefix}\ \mathit{prefix}\ \mathit{args}
\end{aligned}$$

Note that we need to allow overlapping class instances for this to work.

3.8 Connections

We will provide a *connect* function which corresonds to Qt's function *Qt*::*connect*. In contrast to this function, our function will allow for a nicer syntax. Qt's *connect* function takes two argument for each the signal and the slot. One denotes the respective object and one the signal or slot, including the list of argument types. Our *connect* function shall take two arguments, each denoting a so-called port. A port is a pair of a reference and a signal or slot. We define a *Port* type:

$$\textbf{data}\ \mathit{Port}\ \mathit{classID}\ \mathit{portID}\ \mathit{args} = \mathit{Port}\ (\mathit{Ref}\ \mathit{classID})\ \mathit{portID}$$

We want to specify ports using a syntax similar to method calls. The $\rightsquigarrow$ operator serves this purpose:[4]

$$\begin{aligned}
&\textbf{infix}\ 0 \rightsquigarrow\\
&(\rightsquigarrow) :: \mathit{Ref}\ \mathit{classID} \to \mathit{Application}\ \mathit{portID}\ \mathit{args} \to \mathit{Port}\ \mathit{classID}\ \mathit{portID}\ \mathit{args}\\
&\mathit{ref} \rightsquigarrow \mathit{Application}\ \mathit{portID}\ \mathit{args} = \mathit{Port}\ \mathit{ref}\ \mathit{portID}
\end{aligned}$$

The second argument denotes the signal or slot which is usually given as a function application. The argument of this application is only needed to give the signal's or slot's argument types. Its value is irrelevant. We usually write it as $(\mathit{args} :: (\ldots))$ which works since there is a variable *args* defined as follows:

$$\begin{aligned}
&\mathit{args} :: \mathit{args}\\
&\mathit{args} = \mathit{error}\ \texttt{"Argument list placeholder cannot be evaluated."}
\end{aligned}$$

We are now ready to give the implementation of *connect*:

[4]Note that $\rightsquigarrow$ is different from $\leadsto$.

```
connect :: (Signal signalClassID signalID signalArgs,
            Slot slotClassID slotID slotArgs,
            Prefix slotArgs signalArgs) ⇒
           Port signalClassID signalID signalArgs →
           Port slotClassID slotID slotArgs →
           IO ()
connect signalPort slotPort = nativeConnect (ptr signalRef)
                                            signalConnectStr
                                            (ptr slotRef)
                                            slotConnectStr where
  Port signalRef signalID = signalPort
  Port slotRef slotID     = slotPort
  signalConnectStr        = signalConnectString (classID signalRef)
                                                signalID
                                                (fakeArgs signalPort)
  slotConnectStr          = slotConnectString (classID slotRef)
                                              slotID
                                              (fakeArgs slotPort)
```

This implementation uses two helper functions. First, there is

```
nativeConnect :: Ptr CChar → String → Ptr CChar → String → IO ()
```

which is a binding to *QObject::connect*. Second, there is *fakeArgs* which restores the argument list placeholder for a given port and is defined as follows:

```
fakeArgs :: Port classID portID args → args
fakeArgs = const args
```

Note that our *connect* function should never fail since we can assure that the specified ports exist and are compatible. On the other hand, Qt's *connect* method works with string arguments. This makes it impossible to check for compatibility of signals and slots and even for existence of signals and slots at compile time, making runtime checks necessary.

3.9 Subclassing

As mentioned in section 1, we do not want to provide means for creating new subclasses. However, we want to mirror the existing inheritance structure of the respective C++ library and provide a casting function on top of this.

We declare a class whose instances denote subclass-superclass relationships:

```
class Subclass subclassID superclassID where
  ptrCast :: subclassID → superclassID → Ptr CChar → Ptr CChar
```

The *ptrCast* method calls a respective C wrapper function which converts a subclass pointer to a superclass pointer. Note that it is not possible in general to just treat the subclass pointer as a superclass pointer without any actual conversion. Pointer conversion in C++ is likely to change internal representations as soon as virtual superclasses come into play. In order to stay portable, we propose to always use actual conversion functions.

Based on *ptrCast*, we implement a cast function intended for the user which works on *Ref* values:

```
cast :: (Subclass subclassID superclassID) => Ref subclassID -> Ref superclassID
cast subclassRef = superclassRef where
  superclassRef = Ref $ ptrCast (classID subclassRef)
                                (classID superclassRef)
                                (ptr subclassRef)
```

4 Example

We will show our approach in action by translating the example from the "Making connections" section of the first edition of "C++ GUI Programming with Qt 4".

First we show what a binding to Qt would have to contain in order for our translation to work. We start with the class IDs:

```
data QApplication = QApplication
data QPushButton = QPushButton
```

Now we introduce method, signal and slot IDs as well as their corresponding partial applications of the *Application* data constructor:

```
data Exec = Exec
exec = Application Exec
data Quit = Quit
quit = Application Quit
data Show = Show
show = Application Show
data Clicked = Clicked
clicked = Application Clicked
```

Finally, we need to provide the various class instances:

```
instance Constructor QApplication [String] where
  ...
instance Destructor QApplication where
```

```
  ...
instance Method QApplication Exec () where
  type Kind QApplication Exec () = Static
  type Result QApplication Exec () = ()
  ...
instance Method QApplication Quit () where
  type Kind QApplication Quit () = Static
  type Result QApplication Quit () = Int
  ...
instance Slot QApplication Quit () where
  ...
instance Constructor QPushButton (String) where
  ...
instance Method QPushButton Show () where
  type Kind QPushButton Show () = Instance
  type Result QPushButton Show () = ()
  ...
instance Signal QPushButton Clicked () where
  ...
```

And now the example code translated into Haskell:

```
main :: IO ()
main = do
         cmdLineArgs ← getArgs
         appRef ← new QApplication (cmdLineArgs)
         buttonRef ← new QPushButton ("Quit")
         connect (buttonRef ⤳ clicked (args :: ()))
                 (appRef    ⤳ quit    (args :: ()))
         buttonRef ⤳ show ()
         Class QApplication ⤳ exec ()
         delete appRef
         return ()
```