FFICXX

Automatic Haskell-C++ FFI Generator

Ian-Woo Kim

up’here

Bay Area Haskell Users Group
Feb 2, 2016

https://github.com/wavewave/fficxx
https://github.com/wavewave/HROOT-generate

Foreign Function Interface

* Haskell is super-great, but…

* To rule the world, interface with practical foreign library is very important.

* Haskell-C FFI is clearly defined and well-implemented as language standard.

* Haskell is a compiled language to native binary with relatively small size of RTS, which is great for FFI.

* But Haskell is very different. Most of useful libraries are written in imperative OOP framework. This makes FFI difficult.…

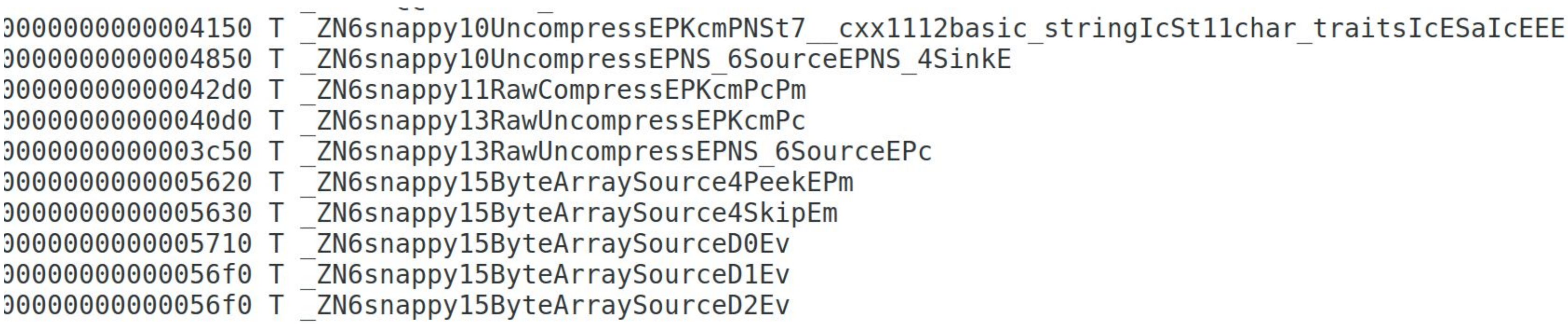
Foreign Function Interface with C++

* My focus is C++. Many high performance computing library written in C++ for numerical analysis.

* C++ complex language with huge syntax.

* C FFI is the only way to make FFI in Haskell. Though good, C++ FFI via C has issues.

    - Name mangling: undefined spec. Cannot rely on mangled name (and it’s ugly)



```
0000000000004150 T _ZN6snappy10UncompressEPKcmPNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
0000000000004850 T _ZN6snappy10UncompressEPNS_6SourceEPNS_4SinkE
00000000000042d0 T _ZN6snappy11RawCompressEPKcmPcPm
00000000000040d0 T _ZN6snappy13RawUncompressEPKcmPc
0000000000003c50 T _ZN6snappy13RawUncompressEPNS_6SourceEPc
0000000000005620 T _ZN6snappy15ByteArraySource4PeekEPm
0000000000005630 T _ZN6snappy15ByteArraySource4SkipEm
0000000000005710 T _ZN6snappy15ByteArraySourceD0Ev
00000000000056f0 T _ZN6snappy15ByteArraySourceD1Ev
00000000000056f0 T _ZN6snappy15ByteArraySourceD2Ev
```

    - OOP introduces lots of boilerplate to interface with C wrapper

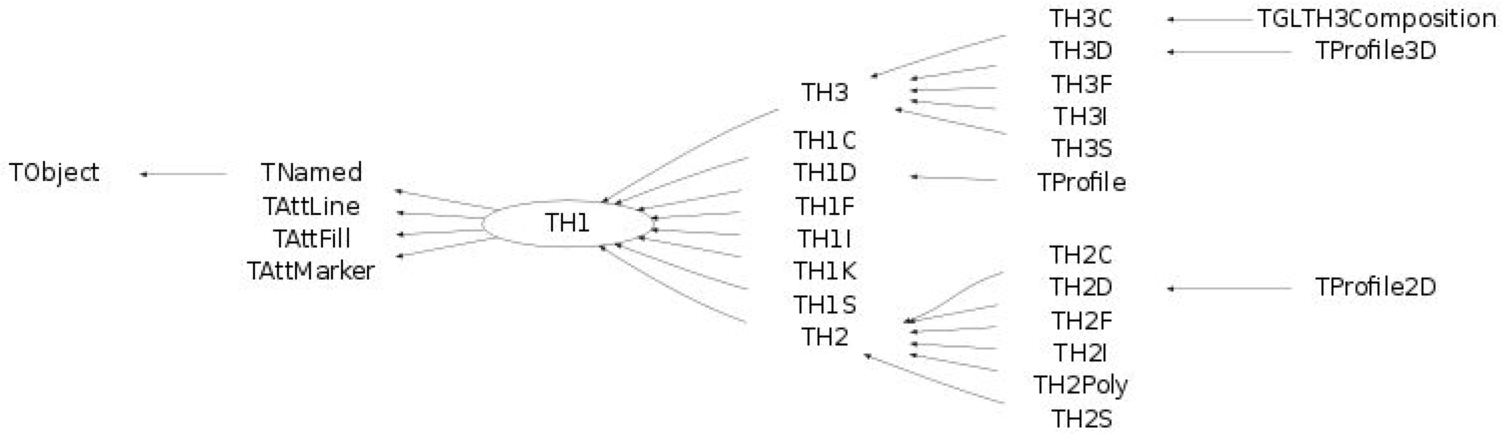
The FFICXX project for me grew out of need to make a good FFI to ROOT

ROOT

* ROOT is Data Analysis Framework developed at CERN, especially suited for statistical analysis in High Energy Physics and Astrophysics. (but it’s general)

* Huge project and has a long history

* Infamously deeply nested OO Class Hierarchy



* Although modern C++ design tends to simplify deep nested class hierarchy, situations in other big frameworks (especially GUI frameworks like Qt) are not very different. ROOT is a typical example.

Maxim of Good FFI

* **Faithful representation:**

- FFI library author should faithfully implement interface without modifying underlying concepts. This is hard between FP and OOP

* **Noninvasive:**

- Syntactic, notational, conceptual or other noise should not be big. It should be well-integrated with target language. -> compatible with other Haskell part well.

ex) direct foreign function is better than quasi-quotation.

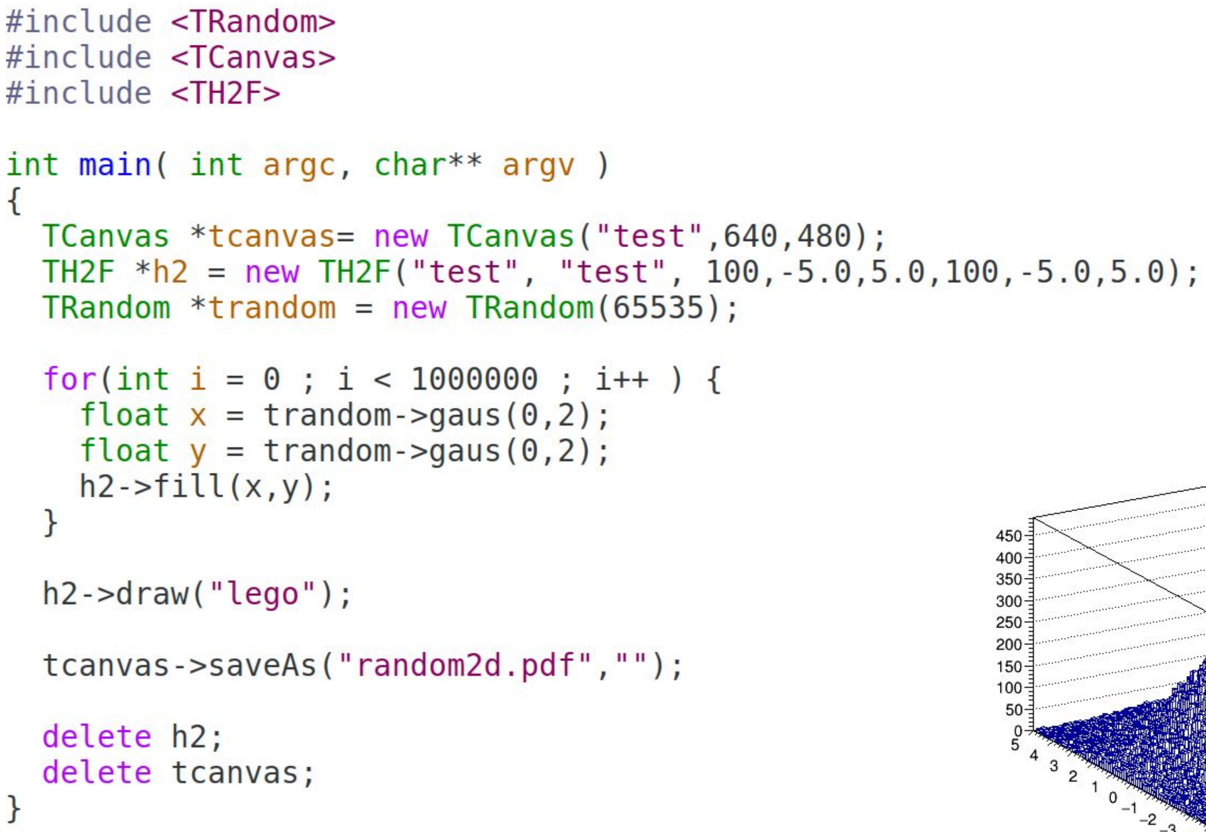
* **Good coverage of syntax:**

- C++ has huge syntax rules. How far can we cover?

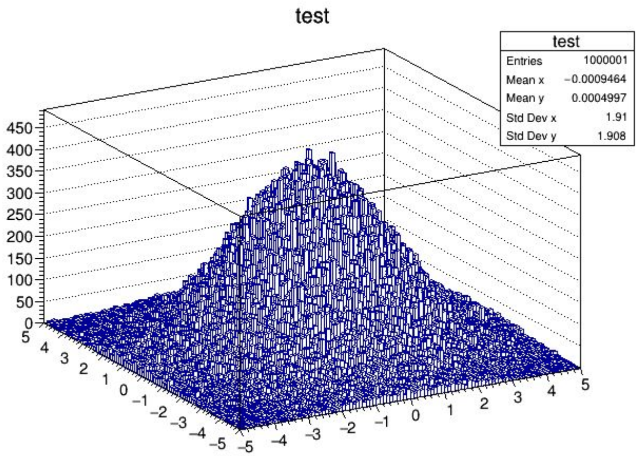
* **Automated!**

- Nobody like to write boilerplate code!

What library user wants:



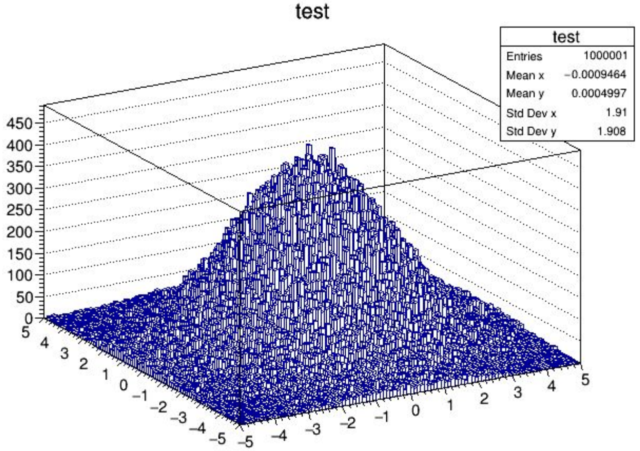
```
#include <TRandom>
#include <TCanvas>
#include <TH2F>

int main( int argc, char** argv )
{
  TCanvas *tcanvas= new TCanvas("test",640,480);
  TH2F *h2 = new TH2F("test", "test", 100,-5.0,5.0,100,-5.0,5.0);
  TRandom *trandom = new TRandom(65535);

  for(int i = 0 ; i < 1000000 ; i++ ) {
    float x = trandom->gaus(0,2);
    float y = trandom->gaus(0,2);
    h2->fill(x,y);
  }

  h2->draw("lego");

  tcanvas->saveAs("random2d.pdf","");

  delete h2;
  delete tcanvas;
}
```



As a ROOT developer, I used to write a program like this. How can I program this similarly in Haskell?

What library user wants - Haskell

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Control.Monad
import Data.ByteString (ByteString)
import Foreign.C.Types
import Foreign.Storable

import HROOT

main :: IO ()
main = do
  tcanvas <- newTCanvas ("Test" :: ByteString) ("Test" :: ByteString) 640 480
  h2 <- newTH2F ("test" :: ByteString) ("test" :: ByteString) 100 (-5.0) 5.0 100 (-5.0) 5.0
  tRandom <- newTRandom 65535

  let gen = gaus tRandom 0 2
      go n | n < 0 = return ()
           | otherwise = do
               (x,y) <- (,) <$> gen <*> gen
               fill2 h2 x y
               go (n-1)
  go 1000000

  draw h2 ("lego" :: ByteString)
  saveAs tcanvas ("test.png" :: ByteString) ("" :: Byt
  delete h2
  delete tcanvas
```

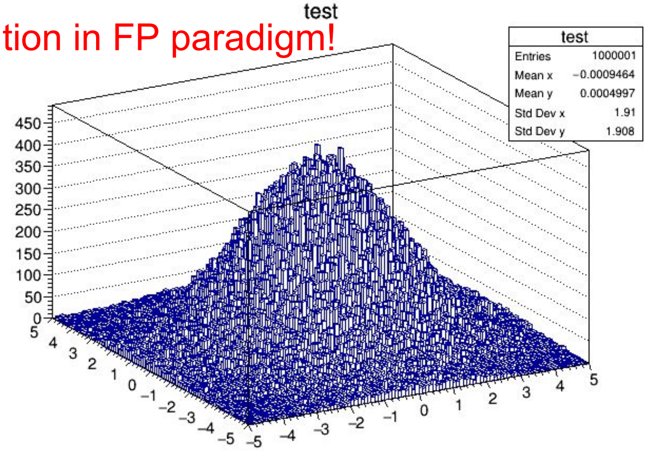


C++ program can be literally translated into Haskell!

What library user wants - Haskell

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Control.Monad
import Data.ByteString (ByteString)
import Foreign.C.Types
import Foreign.Storable

import HROOT

main :: IO ()
main = do
  tcanvas <- newTCanvas ("Test" :: ByteString) ("Test" :: ByteString) 640 480
  h2 <- newTH2F ("test" :: ByteString) ("test" :: ByteString) 100 (-5.0) 5.0 100 (-5.0) 5.0
  tRandom <- newTRandom 65535

  let gen = gaus tRandom 0 2
      go n | n < 0 = return ()
           | otherwise = do
               (x,y) <- (,) <$> gen <*> gen
               fill2 h2 x y
               go (n-1)
  go 1000000

  draw h2 ("lego" :: ByteString)
  saveAs tcanvas ("test.png" :: ByteString) ("" :: Byt
  delete h2
  delete tcanvas
```

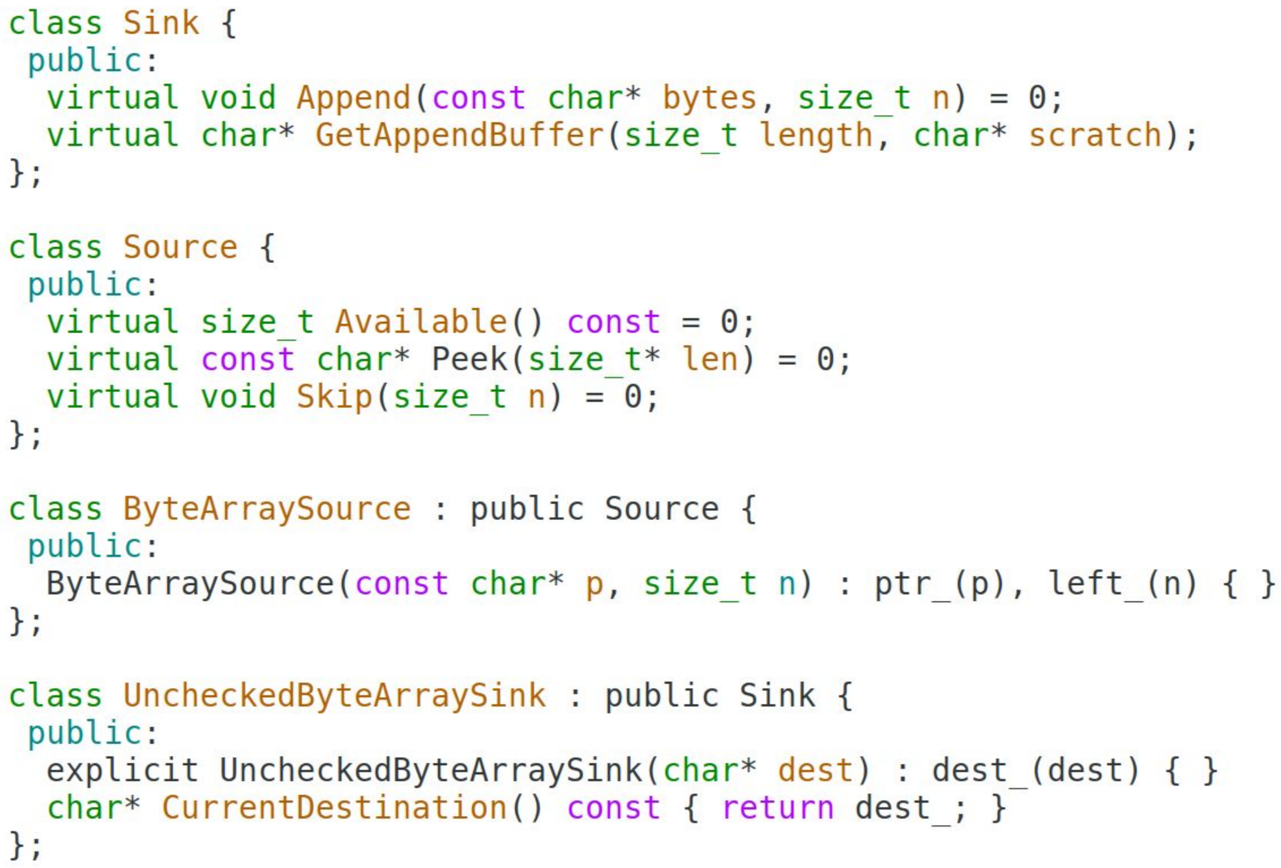
Even better with composition in FP paradigm!



C++ program can be literally translated into Haskell!

For library authors - ex) Snappy library

Classes

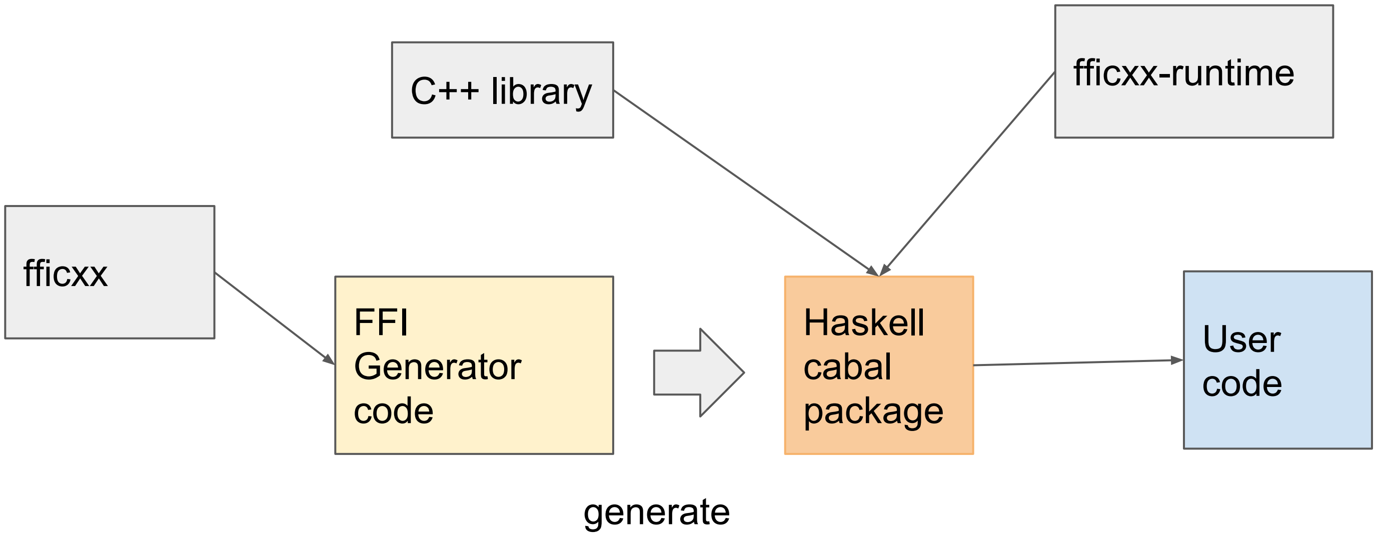


```
class Sink {
 public:
  virtual void Append(const char* bytes, size_t n) = 0;
  virtual char* GetAppendBuffer(size_t length, char* scratch);
};

class Source {
 public:
  virtual size_t Available() const = 0;
  virtual const char* Peek(size_t* len) = 0;
  virtual void Skip(size_t n) = 0;
};

class ByteArraySource : public Source {
 public:
  ByteArraySource(const char* p, size_t n) : ptr_(p), left_(n) { }
};

class UncheckedByteArraySink : public Sink {
 public:
  explicit UncheckedByteArraySink(char* dest) : dest_(dest) { }
  char* CurrentDestination() const { return dest_; }
};
```

Library author - Snappy library (Demo)

HROOT (Demo)

FFICXX pipeline (current stage)



C++ library

fficxx-runtime

fficxx

FFI Generator code

Haskell cabal package

User code

generate

FFI library author implements generator program by making data structure from C++ interface

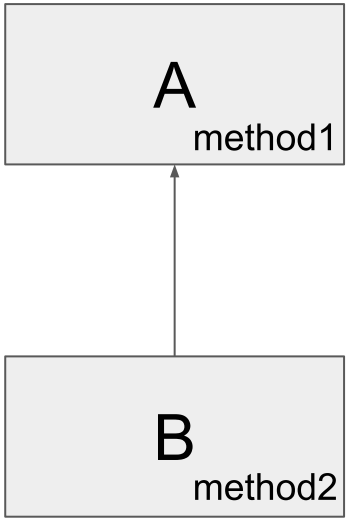
User uses generated library as normal cabal package

Code Generation

Each class is organized into separated modules

- **C shim:** wrapper for C++ methods
- **FFI code:** direct foreign import to C wrapper functions
- **Raw level type:** Haskell newtype wrapper
- **High-level Haskell interface:** type class interface definition
- **Implementation:** type class instance definition
- **Re-exporting module**

MySample example



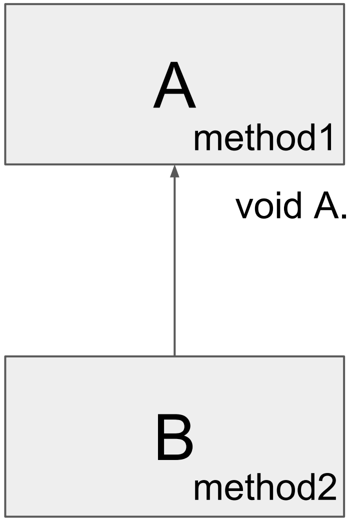
Raw-level newtype wrapper of C pointer

```
data RawA

newtype A = A (Ptr RawA)
          deriving (Eq, Ord, Show)

instance FPtr A where
        type Raw A = RawA
        get_fptr (A ptr) = ptr
        cast_fptr_to_obj = A
```

```
data RawB

newtype B = B (Ptr RawB)
          deriving (Eq, Ord, Show)

instance FPtr B where
        type Raw B = RawB
        get_fptr (B ptr) = ptr
        cast_fptr_to_obj = B
```

MySample example

High-level type class



void A.method1(void)

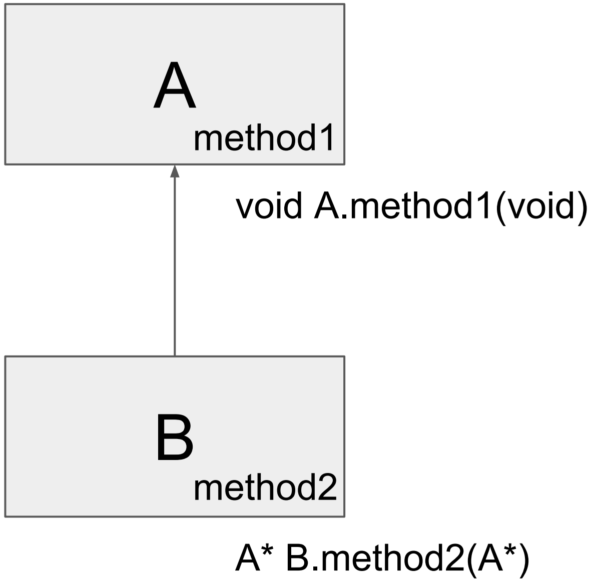
```
class IA a where
        method1 :: a -> IO ()
```

```
class IA a => IB a where
        method2 :: (IA c0, FPtr c0) => a -> c0 -> IO A
```

A* B.method2(A*)

MySample example

C++ wrapper

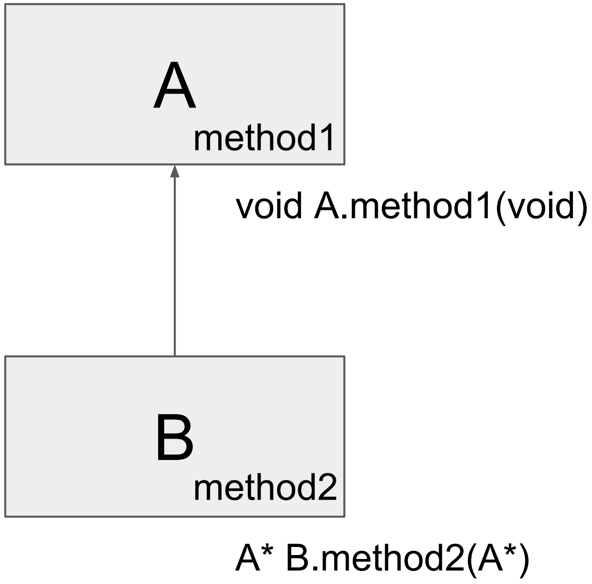


```
typedef A* A_p;

typedef B* B_p;

void A_method1( A_p p ) {
  p->method1();
}

void B_method1( B_p p ) {
  p->method1();
}

A_p B_method2( B_p p, A_p x ) {
  return p->method2(x);
}
```

Actually, fficxx generates C macro functions, but eventually the generated macro will generate the above C wrapper functions.

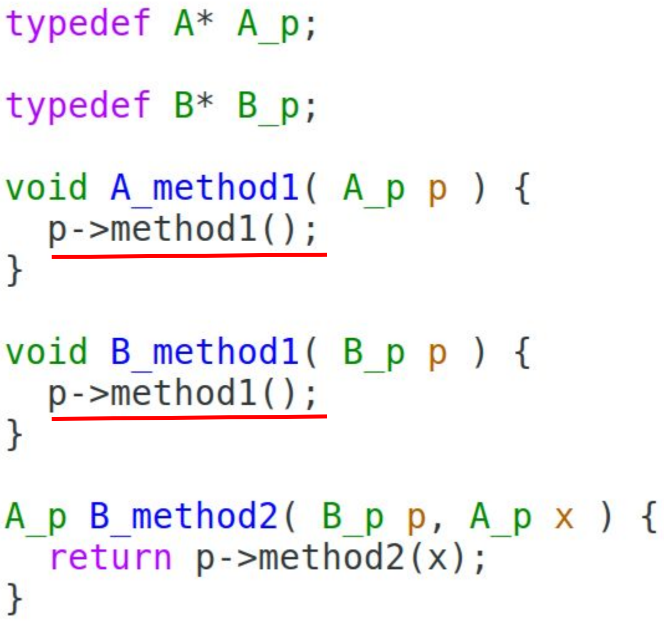
MySample example

C++ wrapper



void A.method1(void)

A* B.method2(A*)



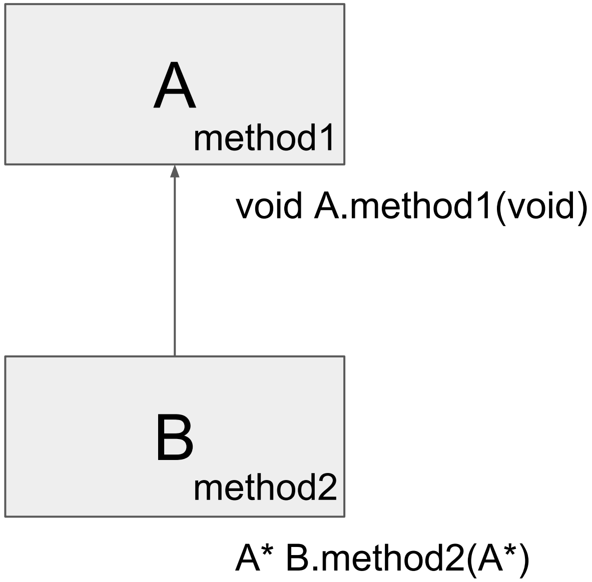
```
typedef A* A_p;

typedef B* B_p;

void A_method1( A_p p ) {
  p->method1();
}

void B_method1( B_p p ) {
  p->method1();
}

A_p B_method2( B_p p, A_p x ) {
  return p->method2(x);
}
```

Due to virtuality, these two methods are different. Therefore, we need to have separate wrappers for each class, and use type class interface if we want to use the same name method1 for A and B

Actually, fficxx generates C macro functions, but eventually the generated macro will generate the above C wrapper functions.

MySample example

FFI and instance definition



```
foreign import ccall safe "MySampleA.h A_newA"
  c_a_newa :: IO (Ptr RawA)

foreign import ccall safe "MySampleA.h A_method1"
  c_a_method1 :: Ptr RawA -> IO ()


foreign import ccall safe "MySampleB.h B_method1"
  c_b_method1 :: Ptr RawB -> IO ()

foreign import ccall safe "MySampleB.h B_newB"
  c_b_newb :: IO (Ptr RawB)

foreign import ccall safe "MySampleB.h B_method2"
  c_b_method2 :: Ptr RawB -> Ptr RawA -> IO (Ptr RawA)
```

```
instance IA A where
        method1 = xform0 c_a_method1

newA :: IO A
newA = xformnull c_a_newa
```

```
instance IA B where
        method1 = xform0 c_b_method1

instance IB B where
        method2 = xform1 c_b_method2

newB :: IO B
newB = xformnull c_b_newb
```

What’s been supported now?

- Nice integration with Haskell language system. Especially, Traditional C FFI well integrated

- High-level C++ OO Class hierarchy : interfacing existing Inheritance and multiple inheritance with typeclasses (but not creating new one)

- virtual / non-virtual method separation, static function, constructor, destructor, top-level functions,

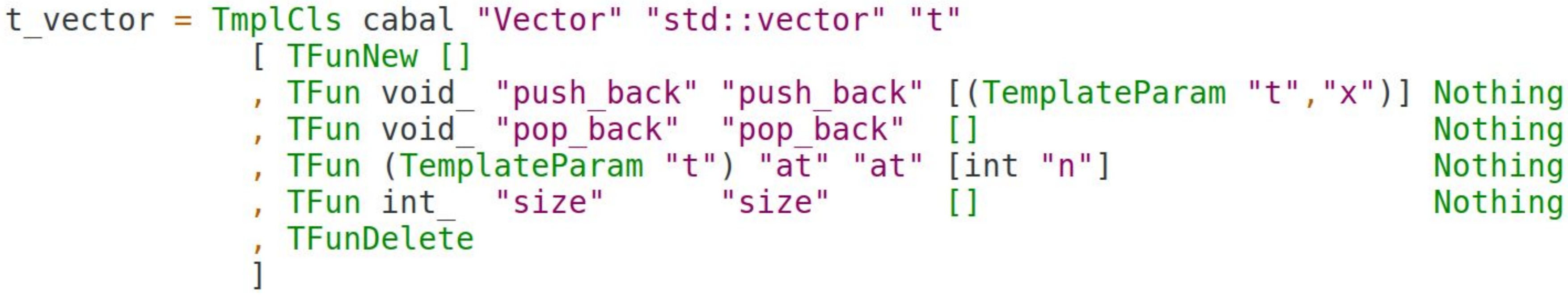
- type safety guaranteed by Haskell type system

- multiple cabal package generation.

- NEW: template class support up to one parameter since 0.3

C++ Template Support

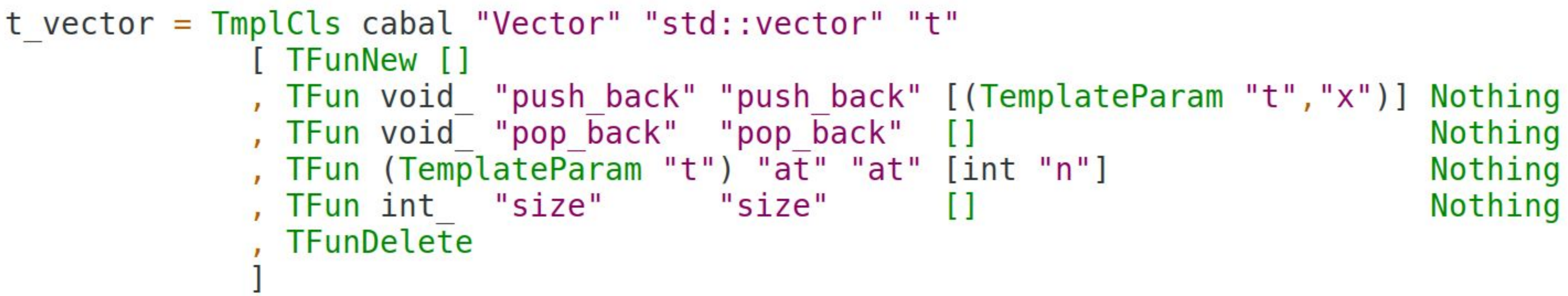
Example: STL vector



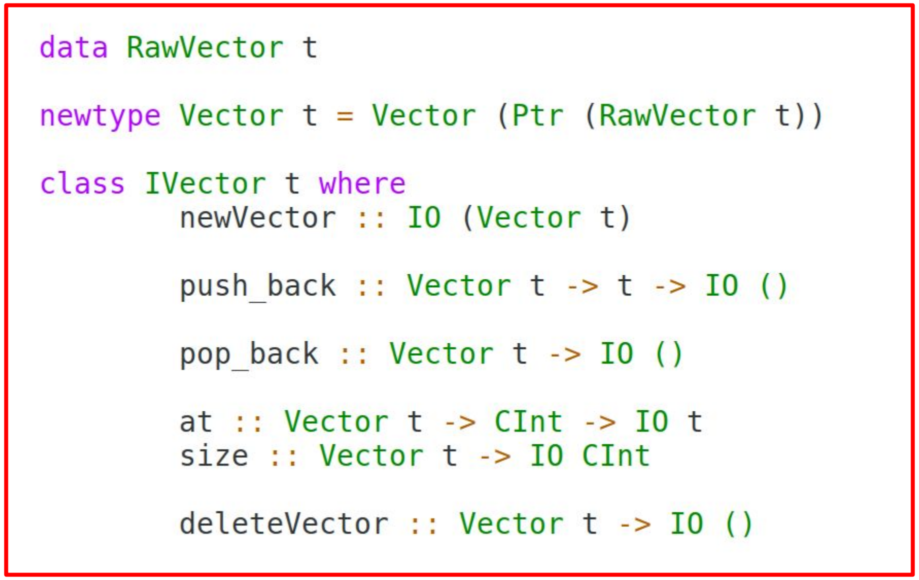
```
t_vector = TmplCls cabal "Vector" "std::vector" "t"
             [ TFunNew []
             , TFun void_ "push_back" "push_back" [(TemplateParam "t","x")] Nothing
             , TFun void_ "pop_back"   "pop_back"  []                        Nothing
             , TFun (TemplateParam "t") "at" "at" [int "n"]                 Nothing
             , TFun int_  "size"      "size"      []                        Nothing
             , TFunDelete
             ]
```

C++ Template Support

Example: STL vector



```
t_vector = TmplCls cabal "Vector" "std::vector" "t"
             [ TFunNew []
             , TFun void_ "push_back" "push_back" [(TemplateParam "t","x")] Nothing
             , TFun void_ "pop_back"  "pop_back"  []                       Nothing
             , TFun (TemplateParam "t") "at" "at" [int "n"]                Nothing
             , TFun int_  "size"      "size"      []                       Nothing
             , TFunDelete
             ]
```



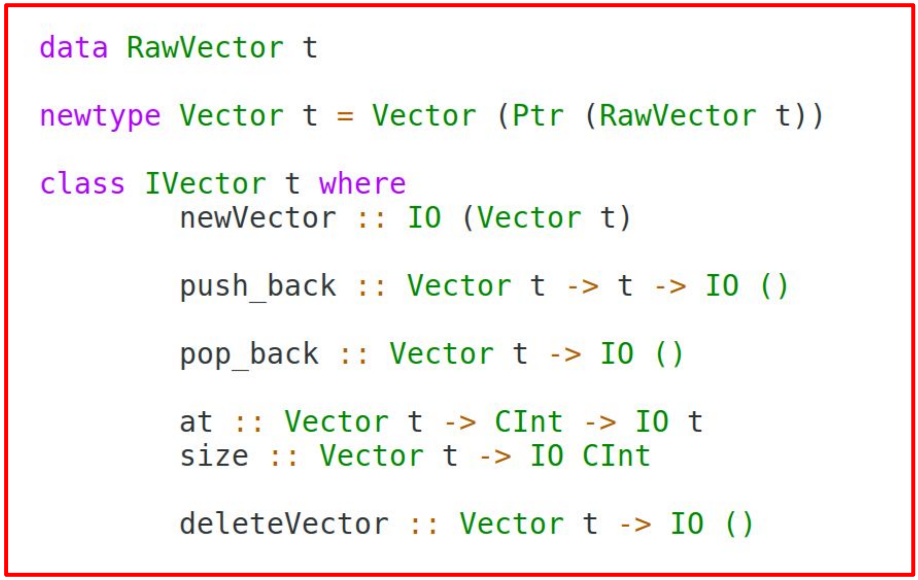
```
data RawVector t

newtype Vector t = Vector (Ptr (RawVector t))

class IVector t where
        newVector :: IO (Vector t)

        push_back :: Vector t -> t -> IO ()

        pop_back :: Vector t -> IO ()

        at :: Vector t -> CInt -> IO t
        size :: Vector t -> IO CInt

        deleteVector :: Vector t -> IO ()
```

Generates high-level interface with type parameter!

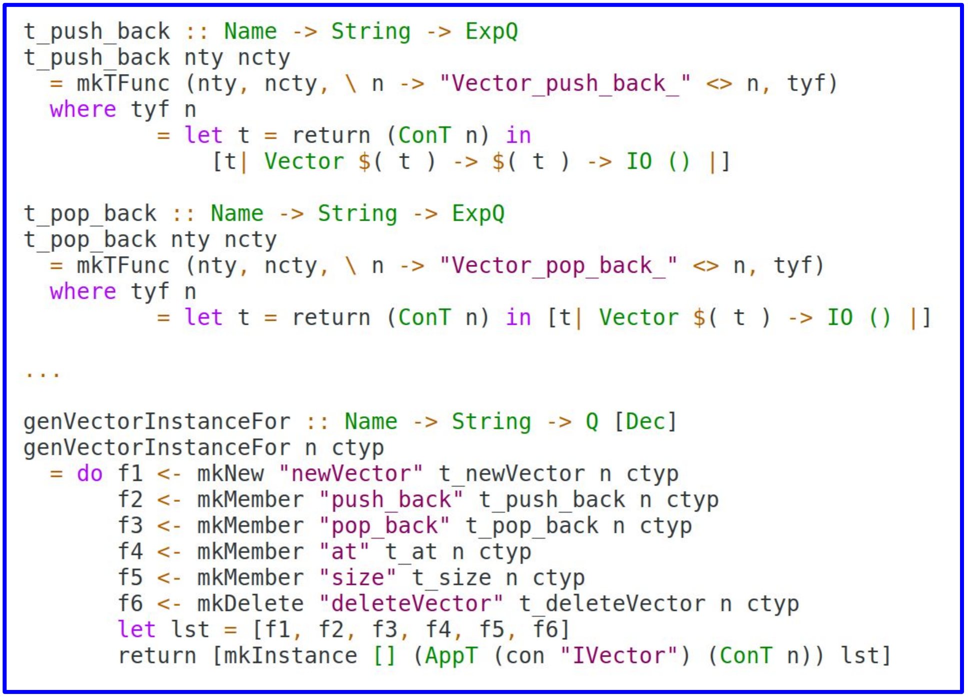
C++ Template Support

Example: STL vector

```
t_vector = TmplCls cabal "Vector" "std::vector" "t"
             [ TFunNew []
             , TFun void_ "push_back" "push_back" [(TemplateParam "t","x")] Nothing
             , TFun void_ "pop_back"  "pop_back"  []                        Nothing
             , TFun (TemplateParam "t") "at" "at" [int "n"]                Nothing
             , TFun int_  "size"      "size"      []                        Nothing
             , TFunDelete
             ]
```



```
data RawVector t

newtype Vector t = Vector (Ptr (RawVector t))

class IVector t where
        newVector :: IO (Vector t)

        push_back :: Vector t -> t -> IO ()

        pop_back :: Vector t -> IO ()

        at :: Vector t -> CInt -> IO t
        size :: Vector t -> IO CInt

        deleteVector :: Vector t -> IO ()
```



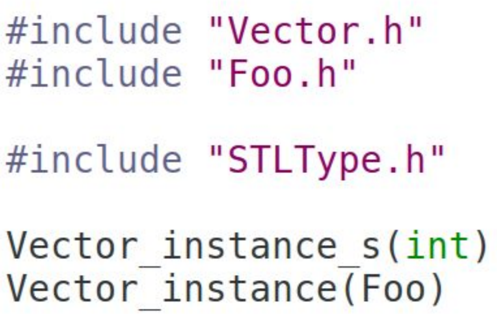
```
t_push_back :: Name -> String -> ExpQ
t_push_back nty ncty
  = mkTFunc (nty, ncty, \ n -> "Vector_push_back_" <> n, tyf)
  where tyf n
          = let t = return (ConT n) in
              [t| Vector $( t ) -> $( t ) -> IO () |]

t_pop_back :: Name -> String -> ExpQ
t_pop_back nty ncty
  = mkTFunc (nty, ncty, \ n -> "Vector_pop_back_" <> n, tyf)
  where tyf n
          = let t = return (ConT n) in [t| Vector $( t ) -> IO () |]


...

genVectorInstanceFor :: Name -> String -> Q [Dec]
genVectorInstanceFor n ctyp
  = do f1 <- mkNew "newVector" t_newVector n ctyp
       f2 <- mkMember "push_back" t_push_back n ctyp
       f3 <- mkMember "pop_back" t_pop_back n ctyp
       f4 <- mkMember "at" t_at n ctyp
       f5 <- mkMember "size" t_size n ctyp
       f6 <- mkDelete "deleteVector" t_deleteVector n ctyp
       let lst = [f1, f2, f3, f4, f5, f6]
       return [mkInstance [] (AppT (con "IVector") (ConT n)) lst]
```

Generates high-level interface with type parameter!

And template haskell implementation!!!

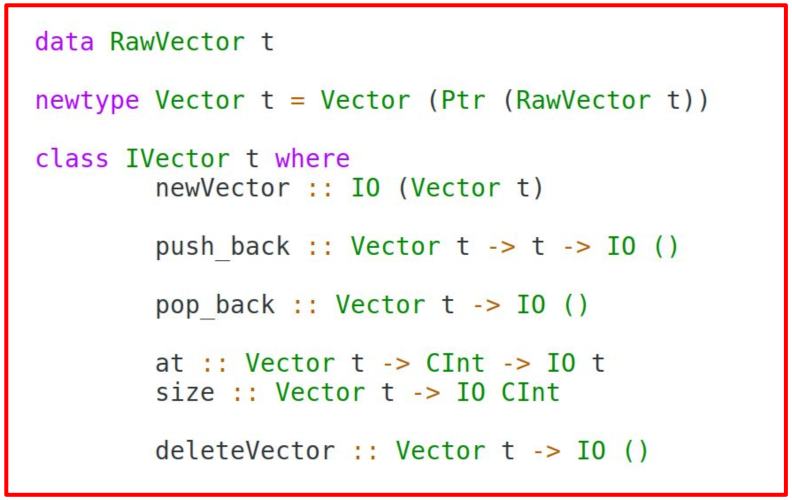
C++ Template Support

Example: STL vector instantiated to std::vector<foo>

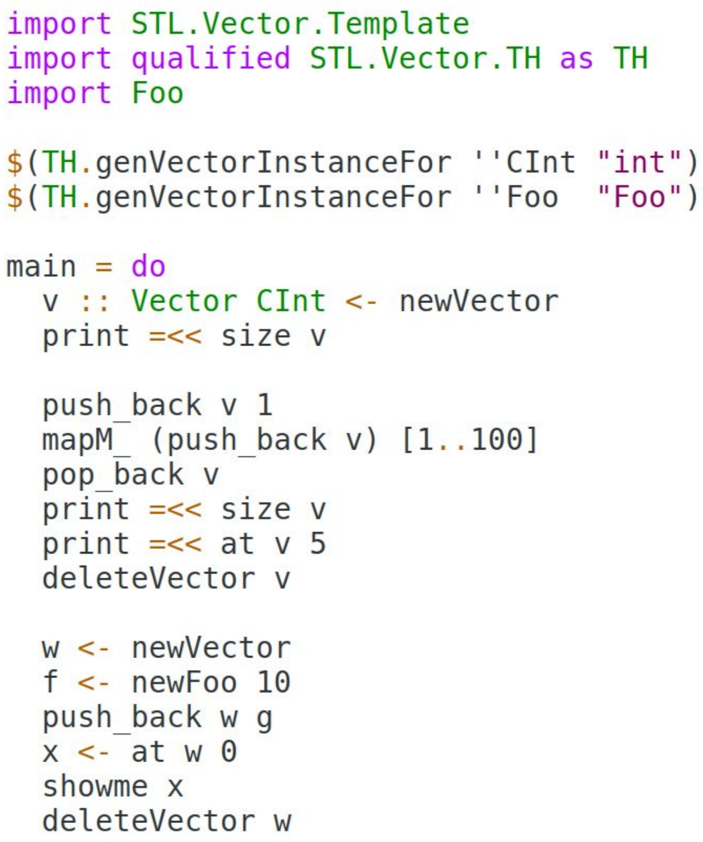


```
#include "Vector.h"
#include "Foo.h"

#include "STLType.h"

Vector_instance_s(int)
Vector_instance(Foo)
```

You need to add one macro function when instantiating the template (std::vector<Foo>)



```
data RawVector t

newtype Vector t = Vector (Ptr (RawVector t))

class IVector t where
        newVector :: IO (Vector t)

        push_back :: Vector t -> t -> IO ()

        pop_back :: Vector t -> IO ()

        at :: Vector t -> CInt -> IO t
        size :: Vector t -> IO CInt

        deleteVector :: Vector t -> IO ()
```



```
import STL.Vector.Template
import qualified STL.Vector.TH as TH
import Foo

$(TH.genVectorInstanceFor ''CInt "int")
$(TH.genVectorInstanceFor ''Foo  "Foo")

main = do
  v :: Vector CInt <- newVector
  print =<< size v

  push_back v 1
  mapM_ (push_back v) [1..100]
  pop_back v
  print =<< size v
  print =<< at v 5
  deleteVector v

  w <- newVector
  f <- newFoo 10
  push_back w g
  x <- at w 0
  showme x
  deleteVector w
```

Use C++ template as a normal higher-order type

Current Progress and Future Direction

* **Template class support:** more generic multiparameter

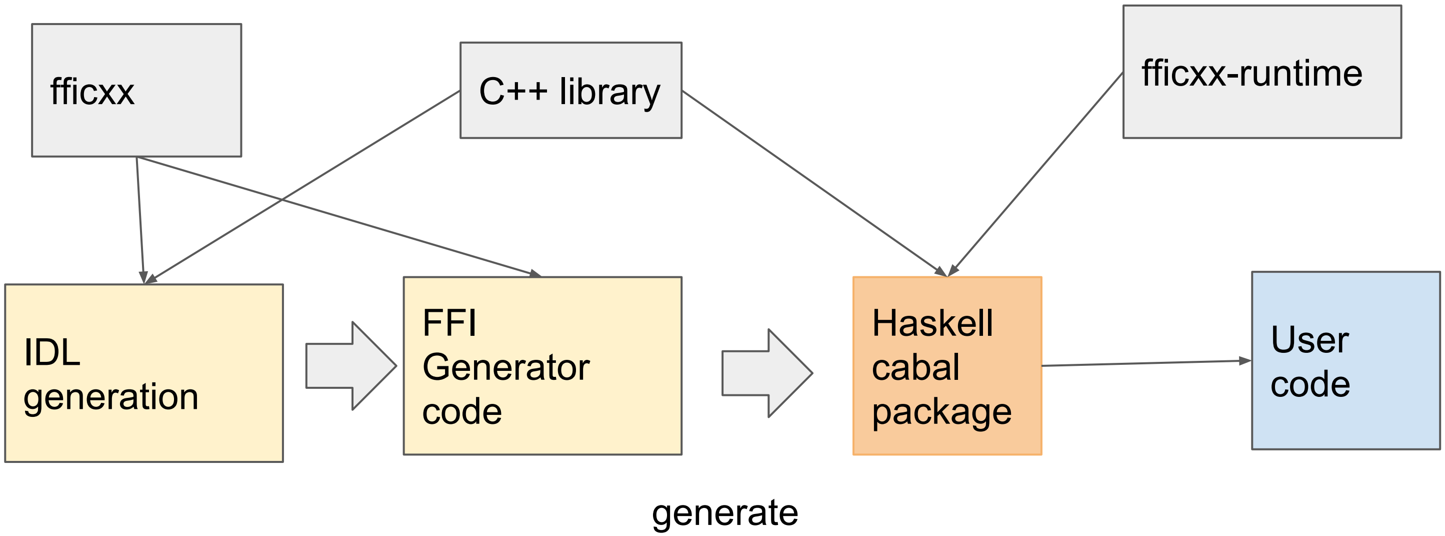
* **Code generation via Abstract Syntax Tree:** haskell-src-exts since 0.3. C code generation part still uses string template, but plan to move to AST.

* **Function pointer and object, lambda:** Function pointer integration has been tested.

* **Default parameters:** different direction of polymorphism in C++. Need some type level tricks to support it. It’s possible, but how to make a clean syntax is an issue

* **DSL for Interface Definition Language (IDL):** Currently, code generator uses internal haskell data type. I am planning to promote it to EDSL and later to DSL. It will be super-nice if we can automatically extract IDL automatically from C++ header file. (pros and cons)

FFICXX pipeline (Future?)



fficxx

C++ library

fficxx-runtime

IDL generation

FFI Generator code

Haskell cabal package

User code

generate

FFI library author just gives the high-level spec of C++ library to fficxx, and interface is automatically extracted and FFI is generated.

User uses generated library as normal cabal package

Thank you!

Please contribute to the fficxx project! :-)

Thank you!

And Announcement:

Bay Area Nix/NixOS User Group

https://www.meetup.com/Bay-Area-Nix-NixOS-User-Group/

Next meetup

@Takt
March 2, 2017