XHaskell

Kenny Zhuo Ming Lu

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
luzm@comp.nus.edu.sg

We present an extension of Haskell, baptized XHaskell, which combines parametric polymorphism, algebraic data types and type classes found in Haskell with regular expression types, subtyping and regular expression pattern matching found in XDuce. Such an extension proves in particular useful for the type-safe processing of XML data. For example, we can express XQuery and XPath style features via XHaskell combinators.

Regular Expression and Data Types In XHaskell we can mix algebraic datatypes and regular expression types. We can give a recast of the classic XDuce example found in [1]. We first consider some datatype definitions as follows,

```
data Person    = Person Name (Tel?) (Email*)
data Name      = Name String
data Tel       = Tel String
data Email     = Email String
data Entry     = Entry (Name,Tel)
```

The above extends data type definitions as found in Haskell. The novelty is the use of regular expression notation on the right-hand sides. Then in the following we can define a transformation from `Person` type to `Entry` type.

```
pToE :: Person -> Entry?
pToE (Person (n::Name) (t::Tel) (es::Email*)) = Entry (n,t)
pToE (Person (n::Name) (t::()) (es::Email*)) = ()
```

In the first clause we use the combination of Haskell style patterns and XDuce style type-based regular expression patterns to check whether a person has a telephone number. In the body of the second clause, we use semantic subtyping. The empty sequence value `()` of type `()` is a subtype of `(Entry?)` because the language denoted by `()` is a subset of the language denoted by `(Entry?)`. Hence, we can conclude that the above program is type correct.

Regular Expression Types and Parametric Polymorphism XHaskell supports the mixing of regular expression type and parametric polymorphism. We are allowed to write a polymorphic traversal function for sequences similar to the `map` function in Haskell.

```
mapStar :: (a -> b) -> a* -> b*
mapStar f (x :: ()) = x
mapStar f (x :: a, xs :: a*) =  (f x, mapStar f xs)
```

In the above, we assume that type annotations are lexically scoped. For example, variable `a` in the pattern `x::a` refers to `mapStar`'s annotation.

We can now straightforwardly specify a function which turns an address into a phone book by mapping function `pToE` over the sequence of `Persons`.

```
data Book a    = Book a*
type Addrbook  = Book Person
type Phonebook = Book Entry

addrbook :: Addrbook -> Phonebook
addrbook (Book (x :: Person*)) = Book (mapStar pToE x)
```

Like `mapStar`, we can easily define `filterStar` which is similar to the `filter` function in Haskell. As a result, we can express star-comprehension in XHaskell similar to the way list- comprehension is expressed via `map` and `filter` in Haskell. With star-comprehension, we can naturally express XQuery style program in XHaskell. For example, the following function `singapore_phonebook` uses star-comprehension to filter out the phone entries which are from Singapore, i.e. having "65" as the country code.

```
singapore_phonebook :: Phonebook -> Phonebook
singapore_phonebook (Book (es :: Entry*)) =
  Book  [ e | e@(Entry (_,t::Tel)) <- es
          , country_code t == "65" ]

country_code :: String -> String
country_code (d1:(d2:_)) = [d1,d2]
country_code _ = ""
```

Like list-comprehension, a star-comprehension consists of a sequence of statements. Concretely, the above star-comprehension has two statements. The first statement `e@(Entry (_,t::Tel)) <- es` is a generator for each entry elements in `es`, we extract the telephone number and bind it to `t`. Via the next statement, we check whether `t` is having a country code `"65"`. If this is the case, we return `e`. In XQuery, the above could be written as follows,

```
declare function singapore_phonebook ($phonebook) {
  for $e in $phonebook/Entry
  where country_code($e/Tel) = "65"
  return $e
}
```

where the for-clause iterates through a sequence of entries, and the where-clause filters out those entries from Singapore. For simplicity, we omit the definition of the `country_code` function in XQuery.

Regular Expression Types and Type Classes XHaskell inherits type classes from Haskell. Therefore, we are allowed to define overloaded function in XHaskell. In our third example, we show how to express a generic set of XPath operations in XHaskell.

```
class XPath a b where
  (//) :: a -> b -> b*

instance XPath a () where
  (//) _ _ = ()

instance XPath a t => XPath a* t where
  (//) xs t = mapStar (\x -> x // t) xs

instance (XPath a t, XPath b t) => XPath (a|b) t where
  (//) (x::a) t = x // t
  (//) (x::b) t = x // t
```

The operation `e1 // e2` extracts all "descendants" of `e1` whose type is equivalent to `e2`'s type.

Integration with GHC and HaXml XHaskell supports a module system and makes use of GHC-as-a-library to process Haskell modules which are imported by a XHaskell program. Thus, XHaskell programmer enjoys the availability of Haskell library support for free.

```
module Main where

import IO           -- Haskell IO module
import Addrbook     -- XHaskell Addrbook module
import XConversion -- XHaskell module defining
                    -- parseXml and writeXml etc

filepath = ``addrbook.xml''

main :: IO ()
main =  do (a :: Addrbook) <- parseXml filepath
           let p = addrbook a
               sp = singapore_phonebook p
           writeXml "myphonebook.xml" sp
```

Our implementation comes with a tool called `dtdToxhs` which we use here to automatically generate XHaskell datatypes from the address book and phone book DTD specifications, We can then import these data types found in module `Addrbook` in the `Main` module. Another XHaskell module `XConversion` provides two functions `parseXml :: String -> IO Addrbook` to read and validate the address book (XML) document and `writeXml :: Phonebook -> IO ()` to store the phone book values into a (XML) file. We read from and print to standard I/O. Therefore, we import the Haskell module `IO`. We make use of GHC-as-a-library to extract type information out of the imported Haskell module `IO`. We use this information to type check and translate the XHaskell program parts.

In the `main` function, we make use of the `addrbook` and `singapore_phonebook` functions defined earlier to transform an address book into a phone book. For instance, given the input file `addrbook.xml` as follows,

```
<addrbook>
 <person>
   <name>Kenny Z. M. Lu</name>
   <tel>65-12345</tel>
   <email>luzm@comp.nus.edu.sg</email>
 </person>
 <person>
   <name>Martin Sulzmann</name>
   <tel>62-98765</tel>
 </person>
</addrbook>
```

executing the program Main yields the following phone book document,

```
<phonebook>
 <entry>
  <name>Kenny Z. M. Lu</name>
  <tel>65-12345</tel>
 </entry>
</phonebook>
```

In pratice, the programmer may want to reuse some of the existing libraries written in HaXml [4]. To allow for easier integration of XHaskell with HaXML legacy code, we provide two XHaskell library functions `toHaXml` and `fromHaXml` to convert data from its XHaskell type representation to HaXml type representation and vice versa. Suppose that `haxml_pToE` is HaXml legacy function which generates a `Entry` element out of a `Person` element. Then we can redefine the function `pToE` from above as follows.

```
import MyHaXmlLib (haxml_pToE)
pToE' :: Person -> Entry?
pToE' x = fromHaXml (haxml_pToE (toHaXml x))
```

In summary, XHaskell is an extension of Haskell which combines algebraic datatypes, parametric polymorphism, type classes found in Haskell and regular expression types, regular expression pattern matching and semantic subtyping found in XDuce. We have implemented XHaskell as a source-to-source translator which can be used in combination with GHC. The interested reader can refer to [5, 3, 2] for more supporting materials.

References

[1] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL'01*, pages 67–80. ACM Press, 2001.

[2] M. Sulzmann and K.Z.M. Lu. A faithful semantics for Hindley/Milner with regular expression types. Manuscript, July 2007.

[3] M. Sulzmann and K.Z.M. Lu. XHaskell – adding regular expression types to Haskell. Manuscript, June 2007.

[4] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP '99*, pages 148–159. ACM Press, 1999.

[5] Xhaskell. http://www.comp.nus.edu.sg/~luzm/xhaskell.