Yhc.Core – from Haskell to Core

by Dimitry Golubovsky ⟨golubovsky@gmail.com⟩
and Neil Mitchell ⟨ndm@cs.york.ac.uk⟩
and Matthew Naylor ⟨mfn@cs.york.ac.uk⟩

The Yhc compiler is a hot-bed of new and interesting ideas. We present Yhc.Core – one of the most popular libraries from Yhc. We describe what we think makes Yhc.Core special, and how people have used it in various projects including an evaluator, and a Javascript code generator.

What is Yhc Core?

The York Haskell Compiler (Yhc) [1] is a fork of the nhc98 compiler [2], started by Tom Shackell. The initial goals included increased portability, a platform independent bytecode, integrated Hat [3] support and generally being a cleaner code base to work with. Yhc has been going for a number of years, and now compiles and runs almost all Haskell 98 programs and has basic FFI support – the main thing missing is the Haskell base library.

Yhc.Core is one of our most successful libraries to date. The original nhc compiler used an intermediate core language called PosLambda – a basic lambda calculus extended with positional information. The language was neither a subset nor a superset of Haskell. In particular there were unusual constructs and all names were stored in a symbol table. There was also no defined external representation.

When one of the authors required a core Haskell language, after evaluating GHC Core [4], it was decided that PosLambda was closest to what was desired but required substantial clean up. Rather than attempt to change the PosLambda language, a task that would have been decidedly painful, we chose instead to write a Core language from scratch. When designing our Core language, we took ideas from both PosLambda and GHC Core, aiming for something as simple as possible. Due to the similarities to PosLambda we have written a translator from our Core language to PosLambda, which is part of the Yhc compiler.

Our idealised Core language differs from GHC Core in a number of ways:

- Untyped – originally this was a restriction of PosLambda, but now we see this as a feature, although not everyone agrees.
- Syntactically a subset of Haskell.
- Minimal name mangling.

All these features combine to create a Core language which resembles Haskell much more than Core languages in other Haskell compilers. As a result, most Haskell programmers can feel at home with relatively little effort.

By keeping a much simpler Core language, it is less effort to learn, and the number of projects depending on it has grown rapidly. We have tried to add facilities to the libraries for common tasks, rather than duplicating them separately in projects. As a result the Core library now has facilities for dealing with primitives, removing recursive lets, reachability analysis, strictness analysis, simplification, inlining and more.

One of the first features we added to Core was *whole program linking* – any Haskell program, regardless of the number of modules, can be collapsed into one single Yhc.Core module. While this breaks separate compilation, it simplifies many types of analysis and transformation. If a such an analysis turns out to be successful then breaking the dependence on whole program compilation is a worthy goal – but this approach allows developers to pay that cost only when it is needed.

A Small Example

To give a flavour of what Core looks like, it is easiest to start with a small program:

```
head2 (x:xs) = x

map2 f [] = []
map2 f (x:xs) = f x : map2 f xs

test x = map2 head2 x
```

Compiling with `yhc -showcore Sample.hs` generates:

```
Sample.head2 v220 =
    case v220 of
        (:) v221 v222 -> v221
        _ -> Prelude.error Sample._LAMBDA228
```

```
Sample._LAMBDA228 =
    "Sample: Pattern match failure in function at 9:1-9:15."

Sample.map2 v223 v224 =
    case v224 of
        [] -> []
        (:) v225 v226 -> (:) (v223 v225) (Sample.map2 v223 v226)

Sample.test v227 = Sample.map2 Sample.head2 v227
```

The generated Core can be treated as a subset of Haskell, with many restrictions:

- Case statements only examine their outermost constructor
- No type classes
- No `where` statements
- Only top-level functions.
- All names are fully qualified
- All constructors and primitives are fully applied

Yhc.Core.Overlay

We provide many library functions to operate on Core, but one of our most unusual features is the *overlay* concept. Overlays specify modifications to be made to a piece of code – which functions should be replaced, which ones inserted, which data structures modified. By combining a Core file with an overlay, modifications can be made after translation from Haskell to Core. This idea originated in the Mozilla project [5], and is used successfully to enable extensions in Firefox, and elsewhere throughout their platform.

To take a simple example, in Haskell there are two common definitions for `reverse`:

```
reverse = foldl (flip (:)) []

reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

The first definition uses an accumulator, and takes $O(n)$. The second definition requires $O(n^2)$, as the tail element is appended onto the whole list. Clearly a Haskell compiler should pick the first variant. However, a program analysis tool may wish to use the second variant as it may present fewer analysis challenges. The overlay mechanism allows this to be done easily.

The first step is to write an overlay file:

```
global_Prelude'_reverse []     = []
global_Prelude'_reverse (x:xs) = global_Prelude'_reverse xs ++ [x]
```

This Overlay file contains a list of functions whose definitions we would like to replace. Any function that previously called `Prelude.reverse` will now invoke this new copy. For a program to insert an overlay, both Haskell files need to be compiled to Core, then the `overlay` function is called.

But we need not stop at simply replacing the `reverse` function. Yhc defines an IO type as a function over the `World` type, but for some applications this may not be appropriate. We can redefine IO as:

```
data IO a = IO a

global_Monad'_IO'_return a = IO a
global_Monad'_IO'_'gt'gt (IO a) b = b
global_Monad'_IO'_'gt'gt'eq (IO a) f = f a
global_YHC'_Internal'_unsafePerformIO (IO a) = a
```

The Overlay mechanism supports escape characters – `'gt` is the `>` character – allowing us to replace the bind and return methods.

We have found that with Overlays a compiler can be customized for many different tasks, without causing conflicts. With one code base, we can allow different programs to modify the libraries to suit their needs. Taking the example of `Int` addition, there are at least three different implementations in use: Javascript native numbers, binary arithmetic on a Haskell data type and abstract interpretation.

Semantics of Yhc Core

In this section an evaluator for Yhc Core programs is presented in the form of a literate Haskell program. The aim is to define the informal semantics of Core programs while demonstrating a full, albeit simple, application of the `Yhc.Core` library.

```
module Main where

import Yhc.Core
import System
import Monad
```

Our evaluator is based around the function `whnf` that takes a Core program (of type `Core`) along with a Core expression (of type `CoreExpr`) and reduces that expression until it has the form of:

- a data constructor with unevaluated arguments, or
- an unapplied lambda expression.

In general, data values in Haskell are tree-shaped. The function `whnf` is often said to "reduce an expression to head normal form" because it reveals the head (or root) of a value's tree and no more. Stricly speaking, when the result of reduction could be a functional value (i.e. a lambda expression), and the body of that lambda is left unevaluated, then the result is said to be in "weak head normal form" – this explains the strange acronym.

The type of `whnf` is:

```
whnf :: Core -> CoreExpr -> CoreExpr
```

Defining it is a process of taking each kind of Core expression in turn, and asking "how do I reduce this to weak head normal form?" As usual, it makes sense to define the base cases first, namely constructors and lambda expressions:

```
whnf p (CoreCon c)              =  CoreCon c
whnf p (CoreApp (CoreCon c) as) =  CoreApp (CoreCon c) as
whnf p (CoreLam (v:vs) e)       =  CoreLam (v:vs) e
```

Notice that a constructor may take one of two forms: stand-alone with no arguments, or as function application to a list of arguments. Also, because of the way our evaluator is designed, we may encounter lambda expressions with no arguments. Hence, only lambdas with arguments represent a base-case. For the no-arguments case, we just shift the focus of reduction to the body:

```
whnf p (CoreLam [] e) =  whnf p e
```

Currently, lambda expressions do not occur in the Core output of Yhc. They are part of the Core syntax because they are useful conceptually, particularly when maniplating (and evaluating) higher-order functions.

Moving on to case-expressions, we first reduce the case subject, then match it against each pattern in turn, and finally reduce the body of the chosen alternative. In Core, we can safely assume that patterns are at most one constructor deep, so reduction of the subject to WHNF is sufficient.

```
whnf p (CoreCase e as) =  whnf p (match (whnf p e) as)
```

We defer the definition of `match` for the moment.

To reduce a let-expression, we substitute the let-bindings in the body of the let. This is easily done using the Core function `replaceFreeVars`. Like in Haskell, let-expressions in Core are recursive, but before evaluating a Core program we transform them all to non-recursive lets (see below). Notice that we are in no way trying to preserve the sharing implied by let-expressions, although we have done so in more complex variants of the evaluator. Strictly speaking, Haskell evaluators are not obliged to implement sharing – this is why it is more correct to term Haskell non-strict than lazy.

```
whnf p (CoreLet bs e) =  whnf p (replaceFreeVars bs e)
```

When we ecounter an unapplied function we call `coreFunc` to lookup its definition (i.e. its arguments and its right-hand-side), and construct a corresponding lambda expression:

```
whnf p (CoreFun f)     =  whnf p (CoreLam bs body)
  where
    CoreFunc _ bs body =  coreFunc p f
```

This means that when reducing function applications, we know that reduction of the function part will yield a lambda:

```
whnf p (CoreApp f [])     = whnf p f
whnf p (CoreApp f (a:as)) = whnf p (CoreLet [(b,a)]
                                    (CoreApp (CoreLam bs e) as))
  where
    CoreLam (b:bs) e      = whnf p f
```

Core programs may contain information about where definitions originally occurred in the Haskell source. We just ignore these:

```
whnf p (CorePos _ e) =  whnf p e
```

And the final, fall-through case covers primitive literals and functions which we are not concerned with here:

```
whnf p e =  e
```

Now, for the sake of completeness, we return to our `match` function. It takes the evaluated case subject and tries to match it against each case-alternative (a pattern-expression pair) in order of appearance. We use the "failure by a list of successes" technique [6] to model the fact that matching may fail.

```
type Alt    =  (CoreExpr, CoreExpr)

match       :: CoreExpr -> [Alt] -> CoreExpr
match e as  =  head (concatMap (try e) as)
```

Before defining `try`, it is useful to have a function that turns the two possible constructor forms into a single normal form. This greatly reduces the number of cases we need to consider in the definition of `try`.

```
norm             :: CoreExpr -> CoreExpr
norm (CoreCon c) =  CoreApp (CoreCon c) []
norm x           =  x
```

Hopefully, by now the definition of `try` will be self-explanatory:

```
try                             :: CoreExpr -> Alt -> [CoreExpr]
try e (pat, rhs)                =
  case (norm pat, norm e) of
    (CoreApp (CoreCon f) as, CoreApp (CoreCon g) bs)
       | f == g                 -> [CoreLet (zip (vars as) bs) rhs]
    (CoreVar v, e)              -> [CoreLet [(v, e)] rhs]
    _                           -> []
  where
    vars                        =  map fromCoreVar
```

This completes the definition of `whnf`. However, we would like to be able to fully evaluate expressions – to what we simply call "normal form" – so that the resulting value's tree is computed in its entirety. Our `nf` function repeatedly applies `whnf` at progressively deeper nodes in the growing tree:

```
nf                          :: Core -> CoreExpr -> CoreExpr
nf p e                      =
  case whnf p e of
    CoreCon c               -> CoreCon c
    CoreApp (CoreCon c) es -> CoreApp (CoreCon c) (map (nf p) es)
    e                       -> e
```

All that remains is to turn our evaluator into a program by giving it a sensible `main` function. We first load the Core file using `loadCore` and then apply `removeRecursiveLet`, as discussed ealier, before evaluating the expression `CoreFun "main"` to normal form and printing it.

```
main :: IO ()
main =   liftM head getArgs
     >>= liftM removeRecursiveLet . loadCore
     >>= print . flip nf (CoreFun "main")
```

In future we hope to use a variant of this evaluator (with sharing) in a property-based testing framework. This will let us check that various program analyses and transformations that we have developed are semantics-preserving. As part of another project, we have sucessfully extended the evaluator to support various functional-logic evaluation strategies.

Javascript backend

The Javascript backend is a unique feature of Yhc. The idea to write a converter from Haskell to Javascript, enabling the execution of Haskell programs in a web browser, has been floating around for some time [7, 8, 9]. Many people expressed interest in such feature, but no practical implementation has emerged.

Initial goals of this subproject were:

- To develop a program that converts the Yhc Core to Javascript, thus making it possible to execute arbitrary Haskell code within a web browser.
- To develop an unsafe interface layer for quick access to Javascript objects with ability to wrap arbitrary Javascript code into a Haskell-callable function.
- To develop a typesafe interface layer on top of the unsafe interface layer for access to the Document Object Model (DOM) available to Javascript executed in a web browser.
- To develop or adopt an existing GUI library or toolkit working on top of the typesafe DOM layer for actual development of client-side Web applications.

General concepts

The Javascript backend converts a linked and optimized Yhc Core file into a piece of Javascript code to be embedded in a XHTML document. The Javascript code generator tries to translate Core expressions to Javascript expressions one-to-one with minor optimizations of its own, taking advantage of the Javascript capability to pass functions around as values.

Three kinds of functions are present in the Javascript backend:

- Unsafe functions that embed pieces of Javascript directly into the generated code: these functions pay no respect to types of arguments passed, and may force evaluation of their arguments if needed.
- Typesafe wrappers that provide type signatures for unsafe functions. Such wrappers are either handwritten, or automatically generated from external interface specifications (such as the DOM interface).
- Regular library functions. These either come unmodified from the standard Yhc packages, or are substituted by the Javascript backend using the Core overlay technique. An example of such a function is the `toUpper` function which is hooked up to the Javascript implementation supporting Unicode (the original library function currently works correctly only for the Latin1 range of characters).

Unsafe interfaces

The core part of unsafe interface to Javascript (or, in other words, Javascript FFI) is a pseudo-function `unsafeJS`. The function has a type signature:

```
foreign import primitive unsafeJS :: String -> a
```

The input is a `String`, but the type of the return value does not matter: the function itself is never executed. Its applications are detected by the Yhc Core to Javascript conversion program and dealt with at the time of Javascript generation.

The `unsafeJS` function should be called with a string literal. Both explicitly coded (with (:)) lists of characters and the concatenation of two or more strings will cause the converter to report an error.

A valid example of using `unsafeJS` is shown below:

```
global_YHC'_Primitive'_primIntSignum :: Int -> Int
global_YHC'_Primitive'_primIntSignum a = unsafeJS
  "var ea = exprEval(a); if (ea>0) return 1; else if (ea<0)
   return -1; else return 0;"
```

This is a Javascript overlay (in the sense that it overlays the default Prelude definition of the `signum` function) of a function that returns sign of an `Int` value. The string literal given to `unsafeJS` is the Javascript code to be wrapped. Below is the Javascript representation of this function found in generated code.

```
strIdx["F_hy"] = "YHC.Primitive.primIntSignum";
...
var F_hy=new HSFun("F_hy", 1, function(a){
    var ea = exprEval(a); if (ea>0) return 1;
    else if (ea<0) return -1; else return 0;});
```

Typesafe wrappers

These functions add type safety on top of unsafe interface to Javascript. Sometimes they are defined within the same module as unsafe interfaces themselves, thus avoiding the exposure of unsafe interfaces to programmers.

An example of a handwritten wrapper is a function to create a new `JSRef`: a mechanism similar to Haskell's `IORef`, but specific to Javascript.

```
data JSRef a

newJSRef :: a -> CPS b (JSRef a)

newJSRef a = toCPE (newJSRef' a)
newJSRef' a = unsafeJS "return {_val:a};"
```

Technically, a `JSRef` is a Javascript object with a property named *_val* that holds a persistent reference to some value. On the unsafe side, invoking a constructor for such an object would be sufficient. It is however desired that:

- calls to functions creating such persistent references are properly sequenced with calls to functions using these references, and
- the type of values referred to are known to the Haskell compiler.

The unsafe part is implemented by the function `newJSRef'` which merely calls `unsafeJS` with a proper Javascript constructor. The wrapper part `newJSRef` wraps the unsafe function into a CPS-style function, and is given a proper type signature, so more errors can be caught at compile time.

In some cases, such typesafe wrappers may be generated automatically, using some external interface specifications provided by third parties for their APIs. The W3C DOM interface is one such API. For instance, this piece of OMG IDL:

```
interface Text : CharacterData {
  Text               splitText(in unsigned long offset)
                                      raises(DOMException);
};
```

is converted into:

```
data TText = TText
...
instance CText TText
instance CCharacterData TText
instance CNode TText
...
splitText :: (CText this, CText zz) => this -> Int -> CPS c zz
splitText a b = toCPE (splitText' a b)
splitText' a b
  = unsafeJS "return((exprEval(a)).splitText(exprEval(b)));"
```

These instances and signatures give the Haskell compiler better control over this function's (initially type-agnostic) arguments.

Usage of Continuation Passing Style

Initially we attempted to build a monadic framework. The `JS` monad was designed to play the same role as the `IO` monad plays in "regular" Haskell programming. There were, however, arguments in favor of using Continuation Passing Style (CPS) [10]:

- CPS involves less overhead as each expression passes its continuation itself, instead of `bind` which takes the expression and invokes the continuation
- CPS results in Javascript patterns that are easy to detect and optimize, although this is a not implemented yet.
- The Fudgets [11] GUI library internals are written in CPS, so taking CPS as general approach to programming is believed to make adoption of Fudgets easier.

Integration with DOM

The Web Consortium [12] provides OMG IDL [13] files to describe the API to use with the Document Object Model (DOM) [14]. A utility was designed, based on

HaskellDirect [15], to parse these files and convert them to set of Haskell modules. The way interface inheritance is reflected differs from HaskellDirect: in HaskellDirect this was achieved by declaration of "nested" algebraic data types. The Javascript backend takes advantage of Haskell typeclasses – representing DOM types with phantom types, and declaring them instances of appropriate classes.

Unicode support

Despite the fact that all modern Web browsers support Unicode, this is not the case with Javascript: no access to Unicode characters' properties is provided. At the same time it is desirable for a Haskell application running in a browser to have access to such information. The approach used is the same as in Hugs [16] and GHC [17]: the Unicode characters database file from the Unicode Consortium [18] was converted into a set of Javascript arrays, each array entry represents a range of character code values, or a case conversion rule for a range. For this implementation, Unicode support is limited to the character category, and simple case conversions. First, a range is found by looking up the character code; then the character category and case conversion distances, i.e. values to add to character code to convert between upper and lower cases, are retrieved from the range entry. The whole set of arrays adds about 70 kilobytes to the web page size, if embedded inside a `<script>` tag.

Using the Core overlay technique, Haskell character functions (like `toUpper`, `isAlpha`, etc.) were hooked up to the Javascript implementations supporting Unicode. This did not result in noticeable slowdowns; some browsers even showed a minor speedup in functions like `read::String -> Int` that perform large amounts of string manipulations.

Examples of code generation

The two examples below show conversion of real-life functions from Haskell, via Core, to Javascript. It is important to mention that as the Javascript code generator evolves, the resultant code may do so too.

Example 1. Taking a function from a Roman Numeral package:

```
fromRoman = foldr fromNumeral 0 . maxmunch . map toUpper
```

When converted to Yhc Core this becomes:

```
Roman.fromRoman =
    Prelude._LAMBDA27191
      (Prelude._LAMBDA27191
```

```
         (Prelude.map Data.Char.toUpper)
         (Roman.maxmunch Prelude.Prelude.Num.Prelude.Int))
      (Prelude.foldr
         (Roman.fromNumeral
            Prelude.Prelude.Num.Prelude.Int
            Prelude.Prelude.Ord.Prelude.Int)
         0)

Prelude._LAMBDA27191 v22167 v22166 v2007 = v22166 (v22167 v2007)
```

The introduced `LAMBDA` is similar to the composition function `(.)`, only with inverted order of application: `_LAMBDA27191 f g x = g (f x)`

When convert to Javascript we get:

```
/* fromRoman, code was formatted manually */

var F_g8=new HSFun("F_g8", 0, function(){
  return (F_e9)._ap([(F_e9)._ap([new HSFun("F_gz", 0,
    function(){
      return (F_gz)._ap([F_Z]);
    }), new HSFun("F_g9", 0,
    function(){
      return (F_g9)._ap([F_dC]);
    })]), new HSFun("F_gp", 0,
    function(){
      return (F_gp)._ap([new HSFun("F_g7", 0,
        function(){
          return (F_g7)._ap([F_dC, F_d1]);
        }), 0]);
    })]);
});

/* _LAMBDA27191 */

var F_e9=new HSFun("F_e9", 3, function(_b3, _b2, _b0)
         {return (_b2)._ap([(_b3)._ap([_b0])]);});
```

During the conversion to Javascript, all identifiers found in Yhc Core are renamed to much shorter ones consisting only of alphanumeric characters and thus surely valid for Javascript (identifiers in Yhc Core often are very long, or contain special characters, etc.)

While it is hard to understand anything from the Javascript for the `fromRoman` function (other than that the Javascript backend already makes a good obfuscator), something may be seen in the Javascript for the composition function. It builds an application of its first argument to the third, and then the application of the second to the previous application, and returns the latter.

Example 2. An example of a function whose implementation was replaced via the Overlay technique is the `isSpace` function:

```
global_Data'_Char'_isSpace = f . ord
  where f a = unsafeJS "return uIsSpace(exprEval(a));"
```

Translated to Core:

```
Data.Char.isSpace =
    Prelude._LAMBDA27191
      Data._CharNumeric.ord
      StdOverlay.StdOverlay.Prelude.287.f
```

Translated to Javascript:

```
var F_W=new HSFun("F_W", 0, function(){
    return (F_e9)._ap([F_bh, F_hk]);});
```

In the Haskell code, the `global_Data'_Char'_isSpace` identifier tells the Core Overlay engine that the function with qualified name `Data.Char.isSpace` is to be replaced with a new implementation. In Yhc Core, the previously reviewed reversed composition function can be seen which composes the `ord` function, and an inner function that actually invokes the Javascript function which in turn performs the Unicode properties lookup for a given character numeric code.

Browser compatibility

Our implementation is compatibile with major web browsers such as Netscape, Mozilla, Firefox, Microsoft Internet Explorer, and Opera. Simple programs such as echoing an input string, Roman to Decimal number conversion, a simple counter with increment and decrement buttons, etc. were used to test browser compatibility. Some of these programs accessed the DOM directly, other used a subset of the Fudgets API.

Opera generally showed the fastest execution of Javascript, but no representative data sample has been collected yet. Microsoft Internet Explorer showed memory leaks when closures over DOM elements were involved in versions up to 6, but

one source reported that in version 7 memory leaks were greatly reduced, and the speed of Javascript execution increased. The XML HTTP request technique was positively tested on Netscape and Microsoft Internet Explorer. Konqueror has never been tested. Compatibility with Safari has not been achieved so far: attempts to execute any of the test programs mentioned above resulted in obscure error messages delivering no information about the nature of incompatibility.

Future plan: Fudgets

We plan to port some portion of Fudgets, so it becomes possible to write Web applications using this library. Several experiments showed that the Stream Processors (SP), and some parts of Fudget Kernel layers work within a Javascript application. More problems are expected with porting the toplevel widgets due to differences in many concepts between a Web browser and X Windows, for which the Fudgets library was originally developed.

Conclusion and Future Goals

Yhc.Core is a library which has been used by quite a few people, it is still evolving – moving useful operations from the individual programs into the common library. There are at least five additional projects we are aware of that make use of this library including: static checkers, program validation, Java generation, optimisation and user hinting. We are always looking for more users – we hope that by providing the dull stuff (interfacing to Haskell), others will provide the cool applications. If you are tempted to use Core, we are most interested to know: `yhc@haskell.org`.

The original structure of nhc was as one big set of modules – some were broadly divided into type checking/parsing etc, but the overall structure and grouping was weaker than in other compilers. One of our first actions was to split up the code into hierarchical modules, introducing Type.*, Parse.* etc to better divide the components. We hope that some of these other sections can be repositioned as libraries, allowing others to make use of them. This approach attracts us, and we see this as the future direction of our compiler.

Acknowledgements

We would like to thank Mike Dodds and Tom Shackell for making comments on earlier drafts of this article.

Thanks also to everyone who has submitted a patch, become a buildbot, reported bugs or done anything else to benefit the Yhc project. We've put together a list

of most of the people (if we've missed you, we apologise, but definitely value your contribution!)

Andrew Wilkinson, Bernie Pope, Bob Davie, Brian Alliet, Christopher Lane Hinson, Dimitry Golubovsky, Gabor Greif, Goetz Isenmann, Ian Lynagh, Isaac Dupree, Kartik Vaddadi, Krasimir Angelov, Malcolm Wallace, Michal Palka, Mike Dodds, Neil Mitchell, Robert Dockins, Samuel Bronson, Simon Marlow, Stefan O'Rear, Thorkil Naur, Tom Shackell, Twan van Laarhoven

About the Authors

Dimitry Golubovsky is a software engineer, originally from St-Petersburg, Russia. He is currently working in the United States as a SAS consultant. He received MS in Electronics Engineering from St-Petersburg State Electrical Engineering University, formerly LEEI, in 1989. He only practices functional programming in his spare time, but it helps him a lot during his day job.

Neil Mitchell has an MEng in Computer Science and Software Engineering from the University of York. He is still there, working towards a PhD, under the supervision of Colin Runciman.

Matthew Naylor is a member of the programming languages and systems group at the University of York.

References

[1] The Yhc Team. The York Haskell Compiler - user's guide. `http://www.haskell.org/haskellwiki/Yhc`.

[2] Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In **Proc. FPCA '95**, pages 282–292. ACM Press (1995).

[3] Hat – the Haskell Tracer. `http://www.haskell.org/hat`.

[4] Andrew Tolmach. An External Representation for the GHC Core Language (September 2001). `http://www.haskell.org/ghc/docs/papers/core.ps.gz`.

[5] XUL Overlays. `http://developer.mozilla.org/en/docs/XUL_Overlays`.

[6] Philip Wadler. How to replace failure by a list of successes. In **Proc. of a conference on Functional programming languages and computer architecture** (1985).

[7] AJAX applications in Haskell. `http://www.haskell.org//pipermail/haskell-cafe/2006-August/017286.html`.

[8] Re: AJAX applications in Haskell. `http://www.haskell.org//pipermail/haskell-cafe/2006-August/017287.html`.

[9] Hajax. http://www.haskell.org/haskellwiki/Hajax.

[10] Continuations. http://haskell.org/haskellwiki/Continuation.

[11] Fudgets Home Page. http://www.md.chalmers.se/Cs/Research/Functional/Fudgets/.

[12] World Wide Web Consortium. http://www.w3.org.

[13] Object Management Group. http://www.omg.org/gettingstarted/omg_idl.htm.

[14] W3C Document Object Model. http://www.w3.org/DOM/.

[15] HaskellDirect. http://www.haskell.org/hdirect/.

[16] Hugs 98. http://www.haskell.org/hugs.

[17] The Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[18] Unicode Home Page. http://www.unicode.org.