HNH Report

Francisco Ferreira
FERF13027601

April 26, 2010

1 Introduction

1.1 The Name

In the grand GNU tradition the HNH name is a recursive acronym. Standing for "HNH's not Haskell". More than homage to the Free Software Foundation's project this is a consequence of the huge task that would be to implement a proper Haskell compiler. So if HNH pretended to be the full language implementation the limitations section would certainly exceed the maximum ten pages allocated to it. It is important to notice that the discussions of the limitations of the current implementation will probably take a commanding share of the available space.

1.2 Using the compiler

After a successful make, the executable hnh is generated. The usage is as follows:

```
./hnh <cmd> <program source>
```

where the supported comands are:

Command	Description
`c`	compile to code.c
`cd`	compile to code.c and print all code transformations

In order to execute the included sample one needs to call:

```
./hnh c qsort.hnh
```

This will print any errors found during the compilation process, or if no errors are found it will compile the code to the file code.c. If the cd command

Like Haskell	Unlike Haskell
It's a functional language	It's strict
It's pure	No type classes
It shares many syntactic elements of Haskell	Pattern matching is severely simplified (even limited)
Supports user defined operators with specified precedence and associativity	No guards for functions
	Diminutive prelude

Table 1: Haskel and HNH compared

is used, the result of each of the 14 stages of compilation will be printed to the console for debugging purposes.

Finally the command `make runtime && ./runtime` is used to produce the final executable and execute it.

2 The Language

Even not being Haskell by definition, HNH shares a lot of things with Haskell; the table 1 shows the more important bullet points of the similarities and differences.

2.1 The Lexical Structure

HNH lexical structure is mainly based in that from the Haskell Report[1] but some keywords are not part of the language. Block and line comments are now supported (only line comments were supported in the interpreter).

2.1.1 Reserved words

The supported list of reserved words is `infix infixl infixr data type if then else let in case of`. None of these words may be used as an identifier or function name.

2.1.2 Variables,operators and type constructors

Variables are the valid identifier names of the language and consist of a sequence of latin letters (a to z) and ' beginning with a small case letter.

Constructors are the valid identifiers for types and type constructors and consist of a sequence of latin letters (a to z) and ' beginning with a big case letter.

As HNH supports user-defined operators, that are a sequence of one or more of: "`! ^# $ % & * + . - ~/ \| < = > ? @`"

2.1.3 Literals

HNH recognizes literals of four types (besides tuples and lists that will be discussed in the syntax section) integers, floating point numbers, strings and characters.

Integers can be expressed in either base 10 or base 8 by prefixing the number with `0o` or `0O` or base 16 by prefixing them with `0x` or `0X` and using small or big case letters from `a` to `f` for the hex digits. Negative integers use the ~as unary negation operator.

Float numbers, internally represented as Haskell's Doubles, as usual with other languages are composed of a mantissa and an exponent. The exponent can be absent, and negative exponents use the - sign, but negative floating point numbers are indicated with ~. (a tilde and a dot)

While Strings and Chars are similar to Haskell's, a String uses double quotes as delimiters where a Char uses single quotes.

2.2 HNH's Syntax

2.2.1 Type declarations

For user defined types, there is support for two constructs, type synonyms, and algebraic datatypes (`data` declarations). Type synonyms are declared with the `type` keyword. Note that in the current version of the compiler type synonyms are parsed but not used by the compiler.

$$\texttt{type T } u_1 \ldots u_k \texttt{ = } t$$

Where T is the new name for the type t and u_k are t's parameters. The current implementation of the interpreter ignores these declarations as it is dynamically typed.

Algebraic type declarations use the `data` keyword and have the form:

$$\texttt{data T } u_1 \ldots u_k \texttt{ = } K_1 t_{11} \ldots t_{1k_1} | \ldots | K_n t_{n1} \ldots t_{nk_n}$$

Where T is the new declared type and $K_1 \ldots K_n$ are the different constructors and its parameters.

For instance the two built in types Bool and List are declared as follows

```
data Bool = True | False ;
data List a = Cons a List | Nil ;
```

2.2.2 Patterns

The core of a program are function and variable declarations, both use the concept of patterns. Patterns in HNH are very simple version compared to those from Haskell. Patterns can be:

- A simple variable name that matches any value
- A type constructor that matches its parameters (i.e. (Cons d rest) matches d to the head of the List and rest to the tail of the list)
- (d:rest) matches to a list and bind d to the head and rest to the tail, it can be considered syntactic sugar for the previous example
- [] Matches the empty list and produces no variable bindings
- _ matches anything but produces no bindings
- (a, b) matches to a tuple and binds a to the first element and b to the second, tuples can have an arbitrary number of elements

Parameters can not contain other patterns, it is not possible to match (Cons a Nil) to the list of one element. This is the main simplification with respect to Haskell's patterns, other differences include the lack of list parameters and the patterns.

2.2.3 Functions and variables

A function declaration is:

$x\ ; p_{11} \dots p_{1k} = e;$

$\dots$

$x\ p_{n1} \dots p_{nk} = e;$

where x is the name of the functions and the p's are the different patterns.

Variable definition is simpler:

$pattern = e;$

2.2.4 User defined operators

User defined operators are sequences of one or more symbols as per section 2.1.2. The precedence and associativity are declared with the `infix, infixr, infixl` exactly in they Haskell does. Operators can be used where variables are required by surrounding them in parenthesis, and functions can be used *infix* by surrounding them with backquotes. One limitation is that *fixity* declarations can only be used in the top-level. Currently the prelude declares the precedence and associativity of the built-in operators which have no built-in precedence otherwise.

2.2.5 Expressions

Expressions are similar to Haskell's

- `let`, `case`,`if`, `case` expressions
- lambda expressions with the same pattern restriction as function declarations
- function applications and infix operators are also unchanged, only that currying infix operators require surrounding them with parenthesis

2.2.6 Type declarations

Type declarations in symbol declarations are parsed but not used, the only type declarations are done by enclosing an expression in parenthesis and annotating the type. (i.e. `(exp :: type)`).

2.2.7 Semicolons everywhere!

HNH does not support the layout rules of Haskell expressions and declarations have to be separated with a semicolon. { } braces are used in `let` and `case` expressions too.

2.3 The Implementation

The compiler is implemented in 14 phases not counting the integrated lexing and parsing.

The lexing phase and the parsing phase are integrated and implemented using Parsec in the file Parser.hs. The parser is implemented in a rather straightforward (i.e. simplistic) way, this makes it easy to understand, easy to debug and easy to extend. One of the notable limitations of this parser is the

cryptic error reports. Better error reports shouldn't be difficult (though time demanding) to implement given the simplicity of the current implementation.

2.3.1 Code Transformations

Fourteen simple transformations are currently done to the parsed program:

- `correctPrecedence` which corrects the precedence of the operators
- `toPrefix` which converts infix operations to function calls
- `funToLambda` which converts all function declarations to lambda expressions
- `simplifyLambda` which simplifies the patterns in the lambda expressions with the restrictions mentioned in section 2.2.2
- `oneVarLambda` which transforms all the lambda expressions in one parameter lambda expressions
- `addIdentifiers` which replaces variable names by unique identifiers
- `performTypeInference` which, rather obviously, performs type inference
- `progToLet` which transforms the program to a single expression (i.e. `main = let ...`)
- `cpsTransform` as the name implies performs a CPS transformation
- `removeVarK` which removes VarK nodes as they are not needed
- `closureConversion` surprisingly this phase does the Closure Conversion of the code to eliminate free variables in functions
- `codeGen` this phase performs hoisting and the C code generation

To implement the transformations, there is a monad `TransformM` in TransformMonad.hs that greatly simplifies writing code transformations that don't require keeping track of the environment.

2.3.2 Code Transformations (The highlights)

The compilation phases can be grouped according to its purpose. The first 6 phases (including addIdentifiers) prepare and simplify the program, eliminating some syntactic sugar and leaving a simpler program representation though a more verbose one.

PerformTypeInference, will then infer the types supporting polymorphism, without type classes or overloading thus the need for different operators for different types (i.e. + and +.). qsort.hnh contains a simple example of polymorphic functions.

After validating the typing of the program the code is generated in a straightforward way CPS Conversion →Closure Conversion →Code Generation. The almost complete lack of optimizations generates a very naïve code, the only optimization is the elimination of VarK nodes that only rename variables. Simple optimizations like inlining all functions called only once would improve the quality of the generated code. It's important to notice that even if they are not implemented, the compiler structure is prepared to host those optimizations without having to introduce big changes.

2.3.3 Code Representation

Two code different types are used. `Program` from Syntax.hs before the CPS transformation and `KExp` from CPSRep.hs that is used after CPS conversion. Once the program passes the type inference inference the typing information is no longer used.

One thing that would improve code readability would be to use an extra representation, that would be introduced after oneVarLambda, as at this point the program has been transformed to used a subset of the features of the original, and it would help diminish some boilerplate code afterwards.

Additionally, typing information would be helpful after CPS conversion to help improve the runtime representation of values by eliminating the need for some boxing.

2.4 The Runtime

The runtime representation is fully boxed, and uses `value *` to represent a value. A `value` contains a `tag` field with the type and a union field with the representation of the stored value. A better representation with more efficient tagging and less indirections could greatly improve performance, and this would not be difficult to add to the runtime by only modifying the c code. The `main` symbol is evaluated and its result is printed by the the final

continuation `HaltK`, the runtime contains printing facilites that print lists, strings, tuples and values in general, so the result will be the value of the symbol main. If this value is a function, the memory address of the function will be printed. The typing information is discarded in the CPS conversion, but as the program was deemed correct then, no type checks are needed (though some asserts are present in the code for debugging purposes).

Each function in the CPS form is compiled to a separate function and in order to avoid the exhaustion of the call stack a trampoline is used in the main function, where each called function never calls another but returns the address of the next.

2.5 Built-ins and the prelude

Several low level operations are implemented as intrinsic, all of them can be found in BuiltIn.hs. Here the compiler defines arithmetic and compare operations, are and the List and Bool types. This is intended to be minimal, as the operations could not be implemented in HNH, and the built in types are used by the syntax of the language (i.e. pattern matching, and if expressions). The arithmetic operations is not absolutely minimal for performance purposes as for example defining the multiplication as a sequence of additions would be unnecessarily slow.

The prelude can be found in prelude.hnh and is loaded before the source of the program, and contains several utility functions, in its current form is not very complete, but the current functions suffice for the sample program.

2.6 The Garbage Collector

The algorithm used is Stop & Copy. The garbage collector uses three memory zones, one for global, permanent objects, a front segment and back segment. When a garbage collection is triggered in the trampoline, the segments are swapped and all the alive objects are copied from the from the back page to the front page. Then the back segment is emptied. The objects in the permanent segment are never garbage collected. The roots are the parameters to the function that include the closure, and the continuation. Additionally all the configuration parameters for memory management are in the file config.h.

2.7 Conclusion

The idea of sequentially transforming the code, the use of powerful techniques as type inference, and CPS and Closure conversion have made possible this

compiler, even if most things were done in a simplistic way. I think that the resulting compiler uses many techniques and has allowed me to learn a lot by implementing the code, and by deciding what to leave out (better error reporting, typed CPS conversion, code optimizations, better closures, a more compact runtime representation, and the list goes on and on). The resulting compiler is simple to understand and simple to extend, and I am very happy with what I learned and with the result (which may be not ready yet to compile code for medical equipment but it was a great vehicle for learning).

2.8 Odds and Ends

The code is also available online at http://www.github.com/fferreira/hnh under the GPL license.

References

[1] Simon Peyton Jones. Haskell 98 language and libraries: the revised report, 2002.