HNH Report

Francisco Ferreira
FERF13027601

February 27, 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 interpreter/compiler. So if HNH pretended to be the full language implementation the limitations section would certainly exceed the maximum six 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 interpreter

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

```
./hnh <program source> <symbol to evaluate> [show transformations]
```

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

```
./hnh qsort.hnh ord
```

The third parameter if it is present triggers the dump of all the code transformations of the source code, and it is used for debugging purposes only, this paremeter can be any string, once present the transformation dump is made (`./hnh qsort.hnh ord dump`).

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

2 The Language

But 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. Additionally block comments are not supported and only line comments are.

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.

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

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 ;
```

Though, these types are declared internally by the interpreter (in the env0 variable declared in EvalEnv.hs), and have no type validation for the parameters in the current interpreter.

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. One limitation of HNH is that we use patterns other variable patterns, the interpreter supports functions with only one parameter. This limitation may be removed for the final compiler.

Variable definition is simpler:

$pattern = e;$

Currently only variable patterns are supported, this limitation also might be removed for the compiler.

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 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 interpreter implements the following phases:
program source →threaded lexer and parser →code transformations →interpreter

2.3.1 The lexer and parser

The lexer is generated with Alex from Lexer.x, and the parser with Happy from Parser.y. HNH uses a monadic parser based in the Happy examples, the current implementation does not take advantage of all the features the parser could offer. Modifications to the parsing code could allow to add better error messages with code locations, and the layout rule. The grammar is implemented in a straight-forward way, with no reduce/reduce conflicts as those render the generated parser is unpredictable for the programmer.

But there are some/many shift/reduce conflicts that are not as problematic because the parser always does shift instead of reduce, with the help of parenthesis and semicolons the programmer can easily be sure of the resulting parse. The lexing and parsing happen as one phase with the lexer doing the work incrementally as the parser demands it.

2.3.2 Code Transformations

Three simple transformations are currently done to the AST:

- `toPrefix` which corrects the precedence of the operators and 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

Current transformation objective is to simplify the language used and to remove special cases for expression evaluation by eliminating infix operation, all functions are lambdas, and all complex pattern are processed in a case expression.

The final phase is the evaluation itself that is simple and straightforward.

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.

2.3.3 The pretty printing

The pretty printing feature uses a representation of the code different from the source code, that would be difficult to use when writing a program, but it is easy to read when debugging the code transformations. Also minimal effort was required for its implementation as allowed by the use of Haskell's pretty printing libraries (the result from the Show class is way to verbose and hard to follow for big ASTs)

2.3.4 Odds and Ends

The code is also available online in http://www.github.com/fferreira/hnh

References

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