Transforming Haskell Structures to Clean

Hajnalka Hegedűs

Abstract— **Both Clean and Haskell are pure functional, lazy programming languages. Haskell is a so called standard in functional programming with many libraries, Clean has an efficient compiler and a user-friendly graphical interface. This paper is to draw a parallel between the two languages with respect to show how can a Haskell code be converted to a Clean code (or something, the Clean compiler can compile), to make it possible to use Haskell libraries mixed with Clean modules using the Clean compiler and user interface. So the advantages of both languages can be exploited at the same time.**

I. Introduction

THIS work's main aim is to give a summary about how to transfer Haskell to Clean. I will explain the differences and similarities between the two languages and give equivalents (in Clean) for Haskell structures where it's possible.

Here I will only give a theoretical background for the transformation, but it is a base of a real implementation of a Haskell to Clean front end. For the implementation I used the existing Clean 2 compiler[1]. My aim was to generate parse trees for Haskell sources, the Clean compiler can compile. These parse trees can be generated based on the equivalences explained in this paper. I have rewritten the lexical and the syntactical analyzer of the compiler to accept Haskell codes. The syntactical analyzer generates parse trees suitable for further processing for the Clean compiler. Some more work needed, since there are parts of Haskell that can't be directly transformed to Clean, only after preprocessing[2].

The paper is based on the language reports on Clean [1] and Haskell [2]. I also used textbooks on the two languages ([4] and [5]) to understand the concepts better, and a paper which compares the two languages [3] for checking the correctness of my statements.

II. General Properties

Both Haskell and Clean are lazy, functional languages with static typing (though Clean 2 will allow dynamic typing under certain conditions as well). Both languages handle parametric and *ad hoc* polymorphism and overloading[3]. Strictness can be annotated by `!`. Both language allows layout-sensitive[4] and layout-insensitive[5] styles.

H. Hegedűs is a student of University Eötvös Loránd, Department of General Computer Science, Budapest, Hungary. E-mail: heha@inf.elte.hu. This article is supported by OMFB 01548.

[1] Still under development

[2] Especially the parts where I couldn't give Clean equivalents

[3] For overloading see also section VIII.

[4] Using the offside-rule.

[5] Using braces and semicolons.

A. Literate Scripts

A Haskell code can be written as a literate script: everything is considered a comment in the text unless the line begins with a '>' mark.

For example:

```
This is a literate script.
>   double :: Int -> Int
>   double x = 2 * x
```

Or a literate script can be written in a LaTeX-like format:

```
\documentclass{article}

\begin{document}

This is a literate script.

\begin{code}
double :: Int -> Int
double x = 2 * x
\end{code}

\end{document}
```

This feature is missing from Clean, but literate scripts can be converted to the usual form. The first type is easy to transform: just skip the lines which don't begin with `>`, and keep the lines beginning with `>`, after the `>` were cut. (This will remove the comments from the source, but if we only give this code to a compiler and do not read it, it won't cause any problem. If we want the transferred code to be as readable as the original, we simply need to keep the lines which don't begin with `>` and insert a `--␣` to the beginning of theese lines, which indicates that the line is a comment.) While transforming the second type, only the interior of `code` environment should be kept.

B. Using Unicode Characters

An other problem is that Haskell accepts Unicode characters under certain conditions while Clean doesn't. Unicode characters can appear in identifiers as well as in literals and some of them are regarded as white spaces. One way to solve this problem is simply using Haskell codes without special Unicode characters, or transform the identifiers containing Unicode characters into something without illegal Clean characters. In this case possible name-conflicts must be taken into account.

III. Comments and White Spaces

A. Comments

There are two ways to write a comment. The first is to put a comment notation to the beginning of the line (`//`

in Clean, two or more `-` ended by a space in Haskell). In this case the line is considered as a comment. Multi-line comments can be written as well in both languages, they are bounded with `/*` and `*/` in Clean and `{-` and `-}` in Haskell, so there are only syntactical differences between Haskell and Clean comments.

B. Whitespaces

Both of the two languages accepts comments, the system dependent newline, horizontal and vertical tab and form feed as whitespace. Haskell accepts carriage return and line feed as well.

IV. Naming Rules

In Clean as well as in Haskell names for variables and type variables begin with a lowercase letter (including `_`) followed by any number of lower or uppercase letters or digits or `^` and `‘` in Clean; `’` in Haskell. In Haskell names which contain `’`, `’` can be replaced with `‘` to get a valid Clean identifier[6]. In Clean, the type definition of a function (if present) must be right before the function body, in Haskell they only have to be in the same module. Haskell modules need some preprocessing to move function body right after function definition. (Otherwise we can only deal with Haskell codes in which the location of function bodies are OK.)

Constructors are the same as variables, except they begin with an uppercase letter. In Haskell module names are the same as the constructors; in Clean, module names don’t have to begin with a capital letter. Type constructors and type classes are the same as constructors in both languages. Naturally reserved words can’t be identifiers neither in Haskell nor in Clean.

A. Prefix and Infix Functions

Any Haskell and Clean functions can be used in infix form, if its name is enclosed between ‘s. For example: one can write

```
x ‘f‘ y
```

instead of

```
f x y.
```

In Haskell operators can be written in prefix form as well, if the symbol is enclosed between parentheses. This feature is missing from Clean (except class definitions and class instances), so there’s no equivalent for this in Clean.

B. Reserved Words

Reserved words can’t appear as identifiers in any language. Unfortunately the set of reserved words isn’t the same for Haskell and Clean. This yields to that there are valid identifiers in Haskell which aren’t valid in Clean and and in Clean which aren’t valid in Haskell:

- The common reserved words are:
  - `case`
  - `if`
  - `import`
  - `in`
  - `infix`
  - `infixl`
  - `infixr`
  - `instance`
  - `let`
  - `module`
  - `of`
  - `where`
- Furthermore Haskell has the following reserved words:
  - `class`
  - `data`
  - `default`
  - `deriving`
  - `do`
  - `else`
  - `newtype`
  - `then`
  - `type`
  - `as`
  - `qualified`
  - `hiding`
- In Clean the further reserved words are:
  - `from`
  - `definiton`
  - `implementation`
  - `system`
  - `code`
  - `let!`
  - `with`
  - `at`
  - `P`
  - `I`

(the last three are reserved only in process annotations).

C. Symbols

For functions and constructors (and even for variables in Haskell) one can use symbols as well (reserved symbols are not to be used). Both in Haskell and Clean the following characters can be used to build these symbols: `!`, `#`, `$`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `>`, `=`, `?`, `@`, `\`, `^`, `|`, `-` and `~`. In Haskell variable symbols can include `:` (not as first character) and constructor symbols must begin with a `:` followed with a same structure as a variable symbol.

D. Reserved Symbols

There also are reserved symbols.

- The commons:
  - `:`
  - `::`
  - `=`
  - `<-`

[6]Haskell doesn’t make distinction between function and variable names while Clean does. In Clean a function name can begin with an uppercase letter too.

– `->`
– `=>`
- The proper ones in Haskell:
– `_`
– `..`
– `\`
– `|`
– `@`
– `~`.
- The proper ones in Clean:
– `/*`
– `*/`
– `:==`
– `=:`
– `\\`
– `<-:`
– `.`
– `&`
– `{|`
– `|}`
– `#`
– `#!`
– `*`
– `!`.

E. Namespaces

Clean's namespaces are:
- *ModuleNames*
- *FunctionNames*, *ConstructorNames*, *SelectorVariables*, *Variables*, *MacroNames*
- *FieldNames*
- *TypeNames*, *TypeVariables*, *UniqueTypeVariables*
- *ClassNames*

Haskell uses namespaces for:
- *variables*
- *constructors*
- *type variables*
- *type constructors*
- *type classes*
- *module names.*

The only constraint is that one can't use the same identifier as a name of the type constructor and a class in the same scope. (Both languages has lexical scoping.) These namespaces are more or less equivalent to each other.

F. Qualified Names

In Haskell one can use qualified names[7]. There is no such feature in Clean, so qualified names should be avoided, or transformed to something else using a preprocessor. A question is how to transform qualified names into valid Clean names. If we just remove the qualifier, there can be naming conflicts. If we rename the identifier to something else (for example `qualifier_name`) how will the compiler know, what definition it refers to? One possible solution is to make a 'firewall' module, which imports (but not exports) the needed functions, 'rename' them to the new names, and import this module from the module the qualified names where. For example if the original files are `Foo` and `My`, one can create a new file `Foo2`, and rewrite `My`:

<table>
<tr><td><pre>module Foo where

const :: Int
const = 5</pre></td><td><pre>module My where

import
  qualified Foo

my_const
  = Foo.const + 4</pre></td></tr>
<tr><td><pre>module Foo2
  (foo_const) where

import Foo

foo_const = const</pre></td><td><pre>module My where

import Foo2

my_const
  = Foo_const + 4</pre></td></tr>
</table>

When one uses a qualifier which differs from the name of the imported module[8], the qualifier should be used instead of the module-name in file `Foo2`

V. Literals

A. Numeric Literals

Integer (of type Int in both languages) literals can be decimal, octal, or hexadecimal numbers in both languages, but in Clean sign is a part of the literal while in Haskell, negative sign is a unary operand (though sign is optional in Clean). Octal numbers begin with `0O` or `0o` in Haskell and `0` in Clean, hexadecimal numbers with `0x` or `0X` in Haskell and `0x` in Clean. The remaining part of any integer literal is a sequence of digits.

Real (or Float in Haskell) literals are sequences of digits including a `.` in themselves. It can be followed with `E` or `e` in Haskell and `E` in Clean, a sign and a decimal number (sign is optional).

Differences between numeric types [9] are only syntactical, so after syntactical conversion, Haskell literals can be used as Clean literals.

There is a two more standard data-types in Haskell, not present in Clean: Integer for arbitrary precision integers[10], and Double for double precision floats.

B. Character Literals

A char literal can be any character surrounded by `'` marks, except `'` and `\`. They also can be special characters: `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\\`, `\"` and `\'` (plus `\a` in Haskell) or `\o octal number` or `\x hexadecimal number`, which give the code of the character to be stored. `\a` is missing from Clean, so it can be replaced with its octal code while processing the Haskell text. There is an other way to give

[7] See also subsection IX-C.

[8] See also subsection IX-C.

[9] Clean and Haskell Int, Clean Real and Haskell Float.

[10] There is a (non-standard) library in Clean for this type.

escape codes in Haskell, missing from Clean. For example `\DEL` represents the escape code of *Delete*. These can be replaced with their octal codes by using a list of abbreviations, where the n^{th} element of the list represents the character with decimal ASCII-code n.

C. Strings

A string literal can be any sequence of characters (including the special characters) surrounded by "s in both languages. The problem is type String is an unboxed array of characters in Clean and a list of characters in Haskell, so it is useful to replace every occurrence of String with [Char] in Haskell codes.

VI. Composed and User Defined Types

A. Algebraic Types

In each language algebraic types can be defined, but in different ways: Haskell uses the `data` or the `newtype` keyword (the only difference between the two is that the second one is implemented in a more efficient fashion), Clean uses `::`. Both are followed by the type's name, an = sign, and the type's constructors separated with `|`. For example both

`data Store = Sto [ (Int,Int) ]` and

`newtype Store = Sto [ (Int,Int) ]`

can be transformed to

`::Store = Sto [ (Int, Int) ]`

in Clean. Type synonyms can be created as well.
- in Haskell: `type T u1...un=t`. For example:

`type IntPair = ( Int, Int )`

- in Clean: `::IntPair:==( Int, Int )`

B. Lists

Lists can be created in both languages, using the same list constructors (`:`). But Haskell allows `(a:b:cs)` where `a` and `b` are of the same type and `cs` is a list of the same type, while in Clean it's only can be written in the following form: `(a:(b:cs))` (which is valid in Haskell as well). In both languages infinite and partially defined lists can be used (it's possible because of lazy evaluation). List comprehension is possible too, but with different syntax:

- in Clean:

```
[exp  \\  generator|filter
       ,  generator|filter
       ,  ...
]
```

- in Haskell:

```
[exp  |  generator or filter
      ,  generator or filter
      ,  ...
]
```

For example:
- in Clean: `[x\\x<-[1..10] | x<>8]`
- in Haskell: `[x|xs<-[1..10], x/=8]`

There are a semantic difference between Haskell and Clean list generators as well. In Clean, a filter always belong to one and only one generator. In Haskell, filters and generators are mixed. So somehow filters has to be binded to generators, which is not trivial to automatise[11].

C. Tuples and Labelled Fields

Tuples also can be written in both languages, with the same syntax. In Haskell, an algebraic type can be defined with labelled fields. It's something similar to records. (And they can be handled as valid Clean records, only the additional possibilities of Clean records can't be used.) The syntax is:

```
data C=F{f1::T1,...,fn::Tn}
```

In Clean, it can be defined as

```
::C={ f1 :: T1|
      ...     |
      fn :: Tn }.
```

Lists and tuples and records are predefined types (**Prelude.hs** both in Haskell and **StdEnv** in Clean; records are only in Clean).

VII. Operators and Functions

A. Operators

Operators are defined in **Prelude.hs** in Haskell. If every part of **Prelude.hs** compiles, we don't have to deal with them separately. It seems a far target, because there are language elements which are difficult to transform to Clean. Though there aren't too many differences between the operators of the two languages. Function composition is notified with a `o` in Clean and `.` in Haskell. Not equal is `<>` in Clean and `/=` in Haskell (they even have different precedence; the first has 2 and the second has 4). There are also operators in Haskell which are missing from Clean especially ones related to monads (for example `>>` or `>>=`).

B. Functions

Functions are defined the same way and making a function to be infix is the same too in both languages. In Haskell, we can give a context (which means one or more of the type variables belong(s) to a class). For example

```
books::Eq a => [(a,b)]->a->[b].
```

In Clean it's a little bit different:

```
books::[(a,b)]->a->[b] |(== a).
```

[11] There is a way to generate lists 'parallelly' from other lists in Clean but not in Haskell.

VIII. Classes

Classes can be defined the same way, but defaults are not the same. in Clean an instance can be default, while in Haskell it means, functions defined in class definition are defaults, so they don't have to be redefined in instances (in that case the default one will be used), moreover the defaults can be defined as the function of other class functions mutually. For example:

```
class Eq a where
    (==), (/=)        :: a->a->Bool
        -- Minimal complete definition:
        --       (==) or (/=)

    x /= y            = not (x==y)
    x == y            = not (x/=y)
```

[2]. This semantical difference is really hard to solve, I can't give any Clean equivalent on it. In Haskell there is a possibility to reuse existing class in new algebraic types by using the keyword `deriving`: `data c=...deriving(c1,...,cn)`, which is missing from Clean. Existing class instances can be used in new class definitions in both languages:

Haskell	Clean

`class (Oldclass a)`
`=> Newclass a`
`where ...`
	`class Newclass a`
	`| Oldclass a`
	`where ...`

This is only a syntactical difference again, it can be solved by syntactical parsing.

IX. Module System

Both Clean and Haskell programs are sets of modules, but the module system is different. That's because the two languages use different methods for information hiding.

A. Information Hiding

Information hiding is slightly different in the two languages. Clean uses definition and implementation modules, Haskell has export lists in module declaration (in Clean class instances can be exported). For example from

```
module Ex ( fun1 ) where

fun1 :: a -> a
fun1 x = x

fun2 :: Int -> Int
fun2 x = 3 + x
```

only `fun1` will be exported. In a normal Clean module everything can be seen from outside. This problem can be solved by defining a 'firewall' module, where only the functions of the export list are imported, and use this module instead of the original one. In this case it's important to note that class instances are always exported in Haskell. An other alternative is to use a definition module, where only the exported items are defined.

B. Modules

Both language uses the `module` keyword, in Clean it's followed by the module's name and the module body, in Haskell it's followed by the module's name too, an export-list (optional, everything is exported when missing) the `where` keyword and the module body. [12]

C. Imports

In both of the two languages one can choose what he or she wants to import from a module, or he or she imports the whole module. In Clean, the first choice is to use
`from` *module_name* `import` *import_list*,
the second is to use
`import` *module_name*.
In Haskell –because of the lack of definition and implementation modules– more can be said in import declarations: we can use the `hiding` keyword, to hide information we don't want to use. An import can be `qualified` so qualifiers have to be written in front of the identifiers. It is the module's name, or after an `as` keyword, we can give an other qualifier name. For example using

```
import qualified Prelude as Prel
   hiding (words)
```

means, that everything from **Prelude.hs** is imported except `words`, and references to the name `funny` must be typed as `Prel.funny`. The problem of qualified names can be solved by using a 'firewall' module too[13] The problem of hidden identifiers is more difficult, since intersection should be used.

X. Other Language Concepts

A. Special Expressions

In both of the two languages can be written a conditional expression, but with different syntax.
In Clean it is `if` *cond cmd1 cmd2*,
in Haskell `if` *cond* `then` *cmd1* `else` *cmd2*.
Case expressions also can be written in both languages, using the same syntax. Let and λ-expressions can be used the same way.

B. Trivial Type

There is a data-type present in Haskell, but not in Clean, the trivial type, which has only one element: `()`[14]. (The type constructor is also `()`.)

[12] In Clean a module can be `definition`, `implementation` and `system`.

[13] See subsection IV-F for details.

[14] It is defined in the standard library **Prelude.hs** (and it can be defined in Clean as well), but it's important to mention it here, since () is normally parsed as something else in Clean.

C. Pattern Matching

Pattern matching is more or less the same in the two languages: both languages use `_` for wild-card. Haskell has two patterns (n+k pattern and irrefutable pattern indicated by `~`), what Clean doesn't, but these patterns are obsolete so it's not really important to deal with them. The as pattern is marked by `=:` in Clean and `@` in Haskell.

XI. Conclusions

The front end, based on these result, is still under construction. I have a working lexical analyzer and a partly-working syntactical analyzer (parser). I replaced the Clean 2 compiler's parser with a small application, which decides if the Clean or the Haskell parser should be called (both parsers use their own lexical analyzer), so the mixing of modules written in the two languages is done, since both parsers generate the same type of parse trees. The preprocessor is for future work, now one has to make the preprocessing by hand. In the end, the front end should work on a reasonable part of Haskell (not including `deriving`, and with some restrictions)

References

[1] Rinus Plasmeijer, Marko van Eekelen, *Language Report Concurrent Clean, version 1.3*, Technical Report CSI-R9816, Computing Science Institute, Faculty of Mathematics and Informatics University of Nijmegen, 1988.

[2] Simon Peyton Jones, John Hughes and others, *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, `http://www.haskell.org/definition/`, 1 February 1999.

[3] Peter Achten, *Clean for Haskellers and vice versa*, January 23, 2001.

[4] Simon Thompson *Haskell, The Craft of Functional Programming*, 2nd ed., Addison-Wesley 1999.

[5] *Functional Programming in Clean*, Draft, `http://www.cs.kun.nl/~clean/Manuals/Clean_Book/body_clean_book.html`, July 1999.