Haskell to Clean Translation

Matthew Naylor

Supervisor: Dr. Malcolm Wallace

Report on a project submitted for the degree of
MEng in Computer Systems and Software Engineering

Department of Computer Science,
University of York

Words: 22,148 (excluding appendices)
as counted by `wc -w` applied to the `detex`ed output

May 2004

Abstract

Modular programming is a widely accepted approach in tackling large and complex problems in computer science. Haskell and Clean are two purely functional programming languages supporting higher order functions and lazy evaluation – two very powerful tools for modular programming.

Considering the similarity in the syntax and semantics of these two languages, and that Haskell'98 essentially provides a subset of Clean's features, it has been proposed that a translator from Haskell to Clean could be developed without significant difficulty.

The possibility of being able to share the Clean compiler with the Haskell community and Haskell libraries with the Clean community, *without significant difficulty*, is surely worth investigating – especially so considering the number of man-hours that go into the development of large compilers and development libraries.

In this project we develop a translator from Haskell'98 to Clean. In doing so we raise several new issues in Haskell to Clean Translation that have not been published before, and propose our solutions to range of issues, some of which were previously unresolved.

Keywords: *functional programming, program translation, program transformation, parsing, pretty-printing, haskell, clean, txl*

Contents

1 Introduction **1**
1.1 Haskell and Clean 2
1.2 This Project 3
1.3 Previous Work 3
1.4 Structure of this Report 4

2 Review **5**
2.1 An Overview of Haskell and Clean 5
2.1.1 Common Features 5
2.1.2 Haskell'98 Specifics 8
2.1.3 Clean Specifics 9
2.2 The Haskell Frontend for Clean 11
2.2.1 Translation Issues 11
2.2.2 Incorrect Claims 13
2.3 The Haskell Prelude for Clean 14
2.4 Program Translation 15
2.4.1 Parsing and Pretty Printing 16
2.4.2 Transformation 17
2.5 Current Situation 17

3 Problem Analysis and Design **19**
3.1 Formalising Translations 19
3.2 Translating Expressions 21
3.2.1 Unary Minus 21
3.2.2 List Syntax 22
3.2.3 Sections 22
3.2.4 Do Notation 24
3.2.5 Arithmetic Sequences 24
3.2.6 List Comprehensions 25
3.2.7 Expression type-signatures 25
3.2.8 Backtick notation 26
3.3 Translating Patterns 27
3.3.1 As-patterns 27
3.3.2 Irrefutable Patterns 29
3.3.3 Successor Patterns 30
3.3.4 Pattern Bindings 30
3.3.5 Prefixed Operators in Patterns 33

3.3.6 Backticked Constructors in Patterns . . . 33
3.4 Translating Type System Issues . . . 33
3.4.1 Built-in Types . . . 33
3.4.2 Curried Type-signatures . . . 34
3.4.3 Newtypes . . . 36
3.4.4 Multi-Variable Type-signatures . . . 37
3.4.5 Default Class Methods . . . 37
3.4.6 Numeric Literals and Default Declarations . . . 37
3.4.7 Automatic Instance Derivation . . . 39
3.4.8 Field Labelling . . . 40
3.4.9 Empty Classes . . . 40
3.5 Declaration Translation . . . 41
3.5.1 Order of Declarations . . . 41
3.5.2 Infix Definitions . . . 41
3.5.3 Fixity Declarations . . . 41
3.6 Translating the Module System . . . 43
3.6.1 Exporting Names . . . 43
3.6.2 Importing Names . . . 44
3.6.3 Qualified Imports . . . 45

4 Implementation **46**
4.1 Specifying Transformations: Comparing Haskell and Txl . . . 46
4.1.1 Parsing and Pretty-printing . . . 46
4.1.2 Translating Lists . . . 48
4.1.3 Translating Sections . . . 49
4.1.4 Translating “do” expressions . . . 50
4.1.5 Symbol table construction . . . 50
4.1.6 Summary . . . 51
4.2 Specifying Transformations: Our Choice . . . 52
4.3 A Hybrid Haskell and Clean Syntax . . . 52
4.4 A Framework for Translation . . . 54
4.5 Summary of Implementation . . . 55

5 Evaluation **57**
5.1 A Comparison of the Translator with the Frontend . . . 57
5.2 Translating “Real” Programs . . . 58
5.3 The Strengths and Weaknesses of our Work . . . 61

6 Further Work and Conclusions **63**
6.1 Further Work . . . 63
6.2 Conclusions . . . 64

Appendix A **67**

Appendix B **70**

List of Figures

3.1 Unary Minus Translation 23
3.2 List Constructor Translation 23
3.3 Section Translation 23
3.4 Arithmetic Sequences 28
3.5 Expression Type-signature Translation 28
3.6 Backtick Translation 28
3.7 As-pattern Translation 28
3.8 Irrefutable Pattern Translation 31
3.9 Successor Pattern Translation 31
3.10 Pattern Binding Translation 31
3.11 Prefixed Pattern Translation 35
3.12 Backticked Constructor Translation 35
3.13 Curried Type Translation (a) 35
3.14 Curried Type Translation (b) 35
3.15 Newtype Translation 38
3.16 Multi-Variable Type-Signature Translation 38
3.17 Default Class Method Translation 38
3.18 Declaration Order Translation 42
3.19 Infix Definition Translation 42
3.20 Fixity Declaration Translation 42

4.1 Data flow diagram for the translator. 53

5.1 Test: Irrefutable Patterns 59
5.2 Test: Successor Patterns 59
5.3 Test: Newtype Pattern Matching 59
5.4 Test: Automatic Instance Derivation 60
5.5 Test: Field Labelling 60
5.6 Test: Pattern-Binding Type-Signatures 60
5.7 Shinkman's White to move and mate-in-4 Problem 61

Chapter 1

Introduction

The essence of computer science is the study of computations. A computation is a process that determines the answer to a given question by following a systematic method known as an algorithm. Scientists often pose questions which are too complicated to solve by hand, but by defining an algorithm which they know will yield the right answer – no matter how long it takes – they can let the computer chug along and eventually find it.

This all seems rather easy, except that experience shows us that defining a correct algorithm is in fact very difficult. The problem is that as the questions become more complex, our brains struggle to keep track of all the things that our algorithm must do. This problem was realised as far back as 1969 when Dijkstra remarked:

> "we must organise the computations in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect" – E. W. Dijkstra[EWD69]

The early solution to the problem was "modular programming" which remains key in many modern programming languages. The main idea of this is to split programs (or algorithms) down into more manageable parts, each of which solves some smaller part of the problem and can be reused several times if necessary. It is then simply a case of "gluing" these modules together to produce the overall program. However, while many imperative languages have been developed to provide suitable facilities for splitting up programs (such as procedures and functions), Hughes[Hug89], a functional programming enthusiast, emphasises not only the importance of "splitting-up" but the importance of the "glue" that binds the pieces together.

Unlike imperative languages, where a function is really just a small block of code, functional languages provide functions as first-class citizens, i.e. a function is a value just like 3.1415. This means that functions can be passed as arguments to other functions and may even return functions as results. Hughes notes that this feature provides functional programmers with one very important kind of glue known as "higher-order functions". For example, by defining a very useful modularisation `foldr f z xs` which when given a list:

$$[x_0,\ x_1,\ \cdots,\ x_n]$$

returns the value:

$$\texttt{f}\ x_0\ (\texttt{f}\ x_1\ (\texttt{f}\ \cdots\ (\texttt{f}\ x_n\ \texttt{z})))$$

we can glue functions together in a variety of ways:

Gluing with `foldr`	Examples
`sum = foldr (+) 0`	`sum [1, 2, 3]` ⇒ `6`
`product = foldr (*) 1`	`product [1, 2, 3]` ⇒ `6`
`and = foldr (&&) True`	`and [True, True, True]` ⇒ `True`
`or = foldr (||) False`	`or [False, False, False]` ⇒ `False`

Hughes states that a second very important kind of glue is lazy evaluation. We illustrate this as follows. Suppose we have a genetic algorithm `evolve pop` which evolves the given population and for each generation, returns the average fitness over all the individuals in it. It is difficult to tell at what point the genetic algorithm should stop: one approach is to stop when the average fitness rises above some pre-defined constant, and a second approach is to stop when the difference in average fitness between any two successive generations is less than some pre-defined constant. To modularise `evolve` so that we can easily implement both methods, it would seem that we need modify it so as to take an extra parameter that it can use to determine when to stop. However, this modification will increase the complexity of our nice `evolve` function with uninteresting details that might obscure its original operation. Instead, it would be much nicer to be able to leave the `evolve` function alone and to define the stopping method in an separate function. By making use of lazy evaluation we can make the `evolve` function return an infinite list of fitnesses representing an infinite number of generations. Then, each stopping method can be defined separately to take as many results as they need.

So, to implement the first stopping method we can use the function `takeWhile f xs` which returns all the elements of the list `xs` as long as `f` holds when applied to each:

```
evolveUntil n = takeWhile (<= n) . evolve
```

And to implement the second stopping method:

```
evolveUntil' n = map snd . takeWhile ((>= n) . fst) . diff . evolve
  where diff xs = zip (zipWith (-) (tail xs) xs) xs
```

It is clear that lazy evaluation is very powerful tool for modularising programs since it allows abstraction to be implemented in a novel way.

So it would appear that due to the importance of modular programming – a widely accepted solution for managing complex programs – functional programming and lazy evaluation are of great value to the area of computer science.

1.1 Haskell and Clean

Haskell[Hs98] and Clean[Cl02] are two purely functional languages, allowing both higher order functions and lazy evaluation, i.e. Hughes' two important tools for modular programming and the reasons why functional programming matters.

Purely functional non-strict languages were around long before both Haskell and Clean were invented. In fact, it was because there were so many different such languages being used in the mid 1980's that Haskell came about. The Haskell committee, formed in 1987, believed that their field was being hampered by this lack of commonality, and their aim was to develop a common language for all researchers to use. Perhaps the language that was most closely followed in the design of Haskell was Miranda[Tur85].

The Concurrent Clean system was developed around the same time at the University of Nijmegen. In contrast to Haskell, Clean was developed as a target language to be used in the compilation of both eager and lazy functional languages, and so only supported the most basic constructs of functional programming. The original motivation of the Clean developers was to demonstrate by proof-of-concept that functional graph rewriting systems (with various extensions) could be used to efficiently implement functional languages. In Plasmeijer's book "Functional Programming and Parallel Graph Rewriting"[Pla93] a formal translation from Miranda to Clean is given.

Clean is no longer a target language but a modern, lazy, purely functional language in its own right. Both languages have evolved to such an extent that together they represent the state of the art in the area, with the latest standardisation of Haskell in 1998 and Clean in 2002. There are considerable similarities in syntax, semantics and features they provide. Historically Clean has always had one feature in particular that is not present in Haskell: uniqueness typing. Uniqueness typing is an extension to the type system whereby the programmer may specify that a function operates on unique data, i.e. the data passed as arguments must not be shared by more than one expression. Therefore, if the type-checker agrees that the data being passed to the function is indeed unique then it is safe to destructively update the data without breaking the purely functional semantics. This feature of Clean allows highly-efficient support for large data-structures, records and arrays since they can be destructively updated in constant-time. In contrast, Haskell can struggle with such structures as it will often duplicate an entire structure if only to change a single element.

1.2 This Project

Considering the similarity in the syntax and semantics of these two languages, and that Haskell'98 essentially provides a subset of Clean's features, it has been proposed that a translator from Haskell to Clean could be developed without significant difficulty.

Recalling Haskell's similarity with Miranda and that a formal translation from Miranda to Clean was once given, it perhaps should not be too much of a surprise to the reader that a translator from Haskell to Clean might be an achievable goal, although both languages have changed quite a bit since their first incarnations.

The combination of such a translator with the existing Clean compiler would essentially provide the Haskell community with a new Haskell compiler. The possibility of being able to make a new Haskell compiler available to the Haskell community and Haskell libraries available to the Clean community, *without significant difficulty*, is surely worth investigating – especially so considering the number of man-hours that go into the development of large compilers and development libraries.

The goal of this project is to develop a translator from Haskell'98 to Clean.

1.3 Previous Work

As recently as 2003, Divianszky and Hegedus have developed and published a "Haskell Frontend for Clean"[Div03], which allows Haskell programs to be included in Clean projects and compiled by the Clean compiler. This promising development is however incomplete and the documentation identifies a list of unsupported Haskell constructs. Furthermore, due to gross lack of documentation about how the system actually works, we remain sceptical as to whether all the issues in Haskell to Clean translation

have been discovered and dealt with correctly. In addition, being a modification to an existing compiler, it is difficult to tell how it works since we have to read the code to do so. This discourages open discussions, suggestions and improvements about the abstract translation of Haskell to Clean in general.

In this project we hope to overcome some of these problems by aiming to produce a complete list of translation issues and a well-defined and documented set of proposed solutions. By developing good documentation, and a standalone source-to-source translator, we hope to increase the liklihood of open discussions relating to the translation within the Haskell and Clean communities.

1.4 Structure of this Report

Chapter 2 – Review We give an overview of Haskell and Clean in an attempt to convince the reader of their similarity. In anticipation of the work ahead, we also identify some of the major differences too. We provide a detailed review and summary of the work carried out by Hegedus and more details about the Haskell Frontend for Clean. We review techniques and tools for parsing, pretty-printing and program transformation – all vitally important if we are to develop a large translator successfully.

Chapter 3 – Problem Analysis and Design This documents the most important work carried out in this project: a detailed description of numerous issues in Haskell to Clean translation, and our proposed solutions.

Chapter 4 – Implementation Here we are interested in how we can implement a translator that conforms to our solutions proposed in the previous chapter. We also discuss implementation decisions that have allowed us to develop so much in so little time, and give details regarding the completeness of our final artifact.

Chapter 5 – Evaluation We evaluate our translator by comparing it with the Haskell Frontend for Clean, and by using it to translate some "real" Haskell programs. The speed of the resulting programs pose some interesting questions about the efficiency of Haskell compilers in comparison to the Clean compiler.

Chapter 6 – Further Work and Conclusions We provide a list of issues that remain to be resolved/implemented and suggest possible ways in which this can be done. Finally, we consider how our work might contribute in the wider context of computer science.

Chapter 2

Review

2.1 An Overview of Haskell and Clean

While there is considerable documentation pertaining specifically to each language, it remains unclear to what degree the two are similar. In this section we attempt to investigate this degree of similarity by reviewing the common features of the two languages and also those features that are specific to each one. However, we emphasize that this is an *overview* of features and is *not* intended to cover either language with a fine-tooth-comb. Rather that task will be the topic of subsequent sections of this report. The basis for the following comparison is taken primarily from the official reports[Hs98, Cl02] of each language.

2.1.1 Common Features

In this section, by reviewing the common features of Haskell and Clean, we show how similar the two languages are, thereby providing motivation that the construction of a translator for a large portion of Haskell might be possible without considerable difficulty.

Pattern Matching and Type Inference Consider the following definition of the function `power` which is valid in both Haskell and Clean.

```
power x 0 = 1
power x n = x * power x (n - 1)
```

When given two numbers, it will raise the first (the base) to the power of the second (the exponent) and return the result. For example:

```
power 2 8 ⇒ 256
```

The use of pattern matching allows the definition to be split into two smaller definitions: one which covers the exponent being 0 (the base case) and the other which covers all other cases (the recursive case).

Notice that the type of the `power` function has not been explicity specified, despite the fact that Haskell and Clean are strongly typed languages. In fact, both Haskell and Clean are so strongly typed that the

type of any arbitrary definition or expression can be inferred automatically (note: extensions in Clean mean that this is not always the case).

Partial Application If we want a more specific version of `power`, for example one that always raises 2 to power of the exponent then the following definition, valid in both Haskell and Clean, can be used:

```
twoToThe = power 2
```

The `twoToThe` function then takes one argument which is actually the second argument to `power`. Note the textual equivalence of `twoToThe` and `power 2` in the following application:

```
twoToThe 8 ⇒ 256
```

Higher Order Functions Now suppose that we want a more specific version of `power` that always squares a number. So now the second argument needs to be partially applied, not the first. The following definition, valid in both Haskell and Clean, may be used:

```
square = (flip power) 2
```

`flip` is an example of a higher order function, that is, a function which takes a function as an argument. The function returned by `flip power` will raise its second argument to the power of the first (compare this to the definition of `power`).

Perhaps the most famous higher order function is `map`, which has an almost identical definition in both Haskell and Clean (only differing with a minor syntactic issue):

Haskell'98	Clean 2.1
`map f []     = []` `map f (x:xs) = f x : map f xs`	`map f []     = []` `map f [x:xs] = [f x : map f xs]`

Note again the support for pattern matching in both languages. `map` can be used to apply a function to each element in a list. The following reduction occurs in both Haskell and Clean:

```
map (\x -> x + 1) [1, 2, 3] ⇒ [2, 3, 4]
```

In this case each element is incremented. Note the use of a lambda expression to specify an anonymous increment function.

Polymorphism The Clean type definition of `map` is only syntactically different to its Haskell'98 counterpart:

Haskell'98	Clean 2.1
`map :: (a -> b) -> [a] -> [b]`	`map :: (a -> b) [a] -> [b]`

The type variables show that `map` can be applied to a list of any type. So in addition to `map` being applied to a list of integers (above), it could also be applied, for example, to a list of booleans:

```
map not [True, False, True] ⇒ [False, True, False]
```

Again, this is valid in both Haskell and Clean.

Lazy Evaluation and Non-strictness A function is said to be *strict* in one of its arguments if, when the value of that argument is undefined, then the result is also undefined. In Haskell and Clean an undefined function, `bot`[1], could be defined as:

```
bot = bot
```

Functions in Haskell and Clean can be non-strict in their arguments. To see this, first consider a strange looking function `f` which just ignores its argument and returns zero:

```
f x = 0
```

Now, `f` is non-strict in its only argument since:

```
f bot ⇒ 0
```

If the expression `f bot` were *eagerly* evaluated (as opposed to *lazily* evaluated) then `bot` would be evaluated before the body of `f`, resulting in a non-terminating reduction. It is clear that lazy evaluation can return results when eager evaluation would just loop, so the support for non-strict semantics in both languages is important if we are able to translate between them easily.

Algebraic Data Types The definition of an algebraic data type differs only due to syntax in the two languages:

Haskell'98	Clean 2.1
`data List a = Nil` `             | Cons a (List a)`	`:: List a = Nil` `          | Cons a (List a)`

In both languages an algebraic data type may be created using one of possibly several different data constructors (e.g. `Nil` or `Cons`) and recursively defined constructors are allowed (e.g. `Cons`). Furthermore, type constructors are allowed, whereby the types themselves can take other types as parameters (e.g. `List Int` for a list of integers).

Type Class System Both Haskell and Clean use type classes to provide a means of overloading the definition of a commonly used function name. For example, the equality function == makes sense in the context of almost any type be it an integer, boolean, or even a list of integers. So, to allow == to be overloaded an `Eq` class could be defined:

Haskell'98	Clean 2.1
`class Eq a where` `  (==) :: a -> a -> Bool`	`class Eq a where` `  (==) :: a a -> Bool`

Note that the `Eq` class could be extended to include the /= function as well since it too is related to equality. Now, for a type to allow the == function to be defined on it, it should be made a member of the `Eq` class. For example, suppose that we wish to make our `List` type a member:

[1] `bot` is short for 'bottom", the name given to the symbol $\perp$ which is used to represent the least value in a Partially Ordered Set. Since the undefi ned function is the least defi ned function, we will call it `bot`.

Haskell'98	Clean 2.1

`instance Eq a => Eq (List a)`
`where`
`  (==) Nil Nil = True`
`  (==) (Cons a as) (Cons b bs) =`
`    a == b && as == bs`
`  (==) _ _ = False`
	`instance Eq (List a) | Eq a`
	`where`
	`  (==) Nil Nil = True`
	`  (==) (Cons a as) (Cons b bs) =`
	`    a == b && as == bs`
	`  (==) _ _ = False`

Although there is a slight syntactical difference, both define a context in the instantiation which says that `List a` is only a member of the `Eq` class provided that `a` is already a member.

Module Imports and Abstract Data Types A module system is provided by both langauges to help mangage namespace and promote modular programming. Importing a set of entities from another module brings their names into the scope of the current module:

Haskell'98	Clean 2.1
`import Stack (push, pop, Stack)`	`from Stack import` `  push, pop, :: Stack`

Here, the constructors of the `Stack` data type are not imported and so its internal representation is hidden by the module system. This enforces `Stack` to be used as an abstract data type.

2.1.2 Haskell'98 Specifics

In this section we focus on the *significant* features of Haskell'98, for which no *direct* equivalents in Clean are known. Note that although there are only a handfull features mentioned here, a considerable number of other (less-significant) constructs are also specific to Haskell e.g. sections, do-notation and class defauts. It is even debatable how significant the features that do appear here are!

Arbitrary Precision Integers Haskell'98 has a built-in data type for arbitrary precision integers (the `Integer` type). For example, 2^{100} can't be stored in the 32-bit `Int` type available in Haskell and Clean, but can be in Haskell's `Integer` type:

```
power 2 100 ⇒ 1267650600228229401496703205376
```

Automatically Derived Classes Since it makes sense to define the equality function `==` on so many different types of data, it would be very convenient if a data type could be automatically instantiated under the `Eq` class. Indeed, Haskell'98 provides such a feature, not limited to the `Eq` class (there are 5 others including `Ord`), through the deriving clause:

```
data Weekday = Mon | Tue | Wed | Thu | Fri
  deriving (Eq, Ord)
```

Since `Weekday` has been automatically instantiated under `Eq` and `Ord` we can immediately use the functions they provide:

```
Mon == Tue ⇒ False
Mon < Tue ⇒ True
```

Field Labelled Types Sometimes it can be tiresome to keep referring to fields of a data constructor by position. Instead, it would be a lot more convenient to give each field a name with which it could be referred by. Indeed, Haskell'98 allows this, as is illustrated by the following simplified data structure for bibliographic entries:

```
data BibEntry = Book { title, author, ISBN :: String, pages :: Int }
              | Thesis { title, author, school :: String }
```

A popular entry might be:

```
bir98 = Book {title = "Introduction to Functional Programming",
              author = "Richard Bird",
              ISBN = "0-13-484346-0", pages = 460}
```

Field labelled types are commonly referred to as records. The following expressions illustrate some syntax that Haskell provides for querying and manipulating records:

```
pages bir98 ⇒ 460                    -- Read page count
(\(Book pages = x) -> x) bir98 ⇒ 460 -- Pattern match
pages (bir98 pages = 461) ⇒ 461      -- Change page count
```

Qualified Names It perhaps shouldn't be too surprising that name clashes arise quite often when importing multiple modules, for example, if modules `M1` and `M2` are both imported and both export the same name `x`. Realising this, Haskell provides qualified names whereby the reference to `x` can be disambiguated by prefixing it with the module name, e.g. `M1.x` and `M2.x`.

2.1.3 Clean Specifics

The purpose of this section is to review the significant features of Clean for which there are no direct equivalents in Haskell'98. This may seem unnecessary since we are interested in a *Haskell to Clean* translator, however it is included for two reasons: firstly, these features may provide inspiration for translating certain Haskell'98 specifics, and secondly, for the sake of interest.

Uniqueness Typing Consider the following expression:

```
let x = 10 in x + 1
```

Should `x` be destructively updated to `11` and returned, or should a new anonymous variable with the value `11` be created and returned? It is clear that the former can be done more efficiently (as only one unit of storage is required), however, using this strategy, what then is the value of the following expression:

```
let x = 10 in (x + 1, x + 2)
```

Depending on the *order* of evaluation it could be `(11, 13)` or `(13, 12)`, when the result should be `(11, 12)`. It would appear that destructive updates can ruin the semantics of functional programs.

However, if we somehow knew that there was only one *unique* reference to `x` when the expression is about to be evaluated then it could safely be updated since no other expression would know about it. Uniqueness typing provides a means by which the user can specify if a value is unique through its type.

The following type definition of `map` allows Clean to destructively update the given input list:

```
map :: (*a -> b) [*a] -> [b]
```

Now the following expression would raise a type error at compile time:

```
let xs = [1, 2] in
  (map inc xs, map square xs)
```

This is because the variable `xs` is not unique, since two different expressions are trying to destructively update it.

Arrays and Records Support for arrays and records is built in to the Clean language. For example, an array:

```
a :: {Int}
a = {1, 2, 3}
```

The element at index `i` may be retrieved using the expression `a.[i]` and updated using {`a & [i] = 10`}. Since Clean allows destructive updates on unique objects, array updating can be very efficient. Records in Clean essentially provide tuples with named fields, for example:

```
:: Book = { author, title, ISBN :: String, pages :: Int }
```

The author field in a book record `b` can be retrieved using the expression `b.author` and updated using {`b & author = "Bird"`}. Records differ from Haskell's field labelled types as they provide only provide a single data constructor (which is not like a normal data constructor in the sense that it cannot be "curried").

Eager Evaluation and Strictness Sometimes lazy evaluation can be inefficient since an unevaluated expression often requires more space than the value of that expression. The Clean language report recommends that functions known to be strict should be explicitly marked as such (although the compiler is clever enough to do it automatically in many cases). To illustrate a strict function (evaluated eagerly) recall the earlier definition of `f` which is now marked as strict:

```
f :: !Int -> Int
f x = 10
```

Now the expression `f bot` loops infinitely as it is evaluated eagerly (i.e. `bot` is evaluated before the body of `f`).

Generics Recall that Haskell'98 uses the `deriving` clause to allow the programmer to automatically instantiate their data types under a set of commonly used classes (e.g. `Eq`). Clean provides a similar but much more flexible approach called generics which is not limited to a small number of commonly used classes: many functions can be defined generically (i.e. over all data types), be they predefined or defined by the user.

Several common generic functions are defined in the library `GenLib`, including the generic equality function `gEq`. Here is an example of how a generic function can be derived for the `Weekday` data type:

```
derive gEq Weekday
```

2.2 The Haskell Frontend for Clean

This section reviews the development of the "Haskell Frontend for Clean" developed by Divianszky and Hegedus. The first work on this development was published in the form of Hegedus' MSc thesis[Heg01a] and subsequently in the refined form of an article[Heg01b] entitled "Transforming Haskell Structures to Clean".

The Frontend[Div03], now maintained by Divianszky, is based on the modification of the Clean compiler to parse a Haskell program to a Clean syntax tree, which is then suitable for the unmodified backend of the Clean compiler.

It seems that the motivation behind this approach is that only a small number of changes to the Clean lexer and parser will be required. However, Hegedus notes that it is difficult to resolve certain issues by modifying the lexer and parser alone:

> "there are parts of Haskell that can't be directly transformed to Clean, only after preprocessing" – Hegedus[Heg01b]

For example, in Clean, type signatures must appear directly above the corresponding function definition. This issue can't be easily resolved by the parser. So, to handle these more difficult issues, a preprocessing step is proposed.

Two methods are suggested for implementing this preprocessor: firstly at the lexical analysis stage, and secondly using a completely separate program. The first method seems quite obscure: would it not be easier to manipulate the program *after* it has been parsed? Under this approach one could imagine that the Clean syntax could be extended to allow the more difficult Haskell constructs which could then be translated to the original Clean syntax in what you might term a post-parsing stage.

Alternatively, if a completely separate program has to be developed then the original idea of creating a Frontend as opposed to, for example a source to source translator, surely needs to be rethought. However, one advantage of using a Frontend is that the compiler can support mixed Haskell/Clean developments, whereby Haskell functions may be called from a Clean module and vice-versa.

In the remainder of this review we will focus on the non-trivial translation issues that have been discovered and documented within Hegedus' article.

2.2.1 Translation Issues

We have summarised and paraphrased the issues raised in the article in Table 2.1. A few informal solutions to some of these issues are also provided.

One issue that has been dealt with in reasonable detail is that of resolving qualified names. Hegedus proposes to rename qualified names (which aren't supported in Clean) such as `A.x` to `A_x`. Then, for each module in the program, a new module should be created that renames all the exported functions

Issue	Description
Positioning of type signatures	In Clean, type signatures must appear directly before the function definition, whereas in Haskell they may appear anywhere within the same module.
Prefix notation of infix operators	In Haskell, an infix operator can be used in a prefix form by enclosing it within parenthesis, as illustrated by the equality `x + y = (+) x y`. Clean doesn't support this notation.
Reserved words	There are several Clean reserved words that are not reserved in Haskell. Such words may not be used as identifiers in Clean.
Qualified names	Clean doesn't support qualified names which are used in Haskell's module system to resolve naming conflicts of identifiers between modules. For example, if modules `A` and `B` both define a function `x` then they may be accessed using the expressions `A.x` and `B.x` respectively.
Unary minus	Clean doesn't support the unary minus operator, for example in the definition `neg n = -n`.
Arbitrary precision integers	Clean doesn't have built in support for arbitrary precision integers as provided by Haskell's `Integer` type.

Default class members	In Haskell, a class definition may include default definitions, for example:
	`class Eq a where`
	`  (==), (/=) :: a -> a -> Bool`
	`  x /= y = not (x==y)`
	Here the `/=` function has a default definition, and so instantiations of this class need only define the `==` function, but may override the definition of `/=` if they wish. Clean lacks support for this short-hand notation.

Deriving classes	In Haskell common type classes, such as the `Eq` class defined above, can be derived automatically for any user defined data type. Clean doesn't support this feature directly.
Successor patterns	Clean doesn't support successor patterns, for example in the definition `dec (n+1) = n` that decrements a natural number.
Irrefutable patterns	Clean doesn't support irrefutable patterns, for example in the definition `f ~(x:xs) = 1` that returns `1` even when passed the empty list despite the fact that `x:xs` only matches a non-empty list.
Module export list	In a Haskell module a list of all names to be exported can be listed after the module name. However, in Clean, exported names have to be placed in a separate file (a specification file, which includes the types of the names as well).
Hiding clause	In Haskell, an `import` statement may specify to import all definitions from a module *except* those defined in a given `hiding` list.
Tuples	Clean doesn't support tuples of more than 32 elements, whereas Haskell does.

Table 2.1: Our summary of the issues raised by Hegedus[Heg01b]

to their qualified equivalents (using the underscore notation). When a module is imported, its corresponding module of qualified names should also be imported.

The majority of the remaining solutions focus on overcoming the syntactic issues between the languages.

2.2.2 Incorrect Claims

Although the article nicely describes several incompatibilities between Haskell and Clean, it also makes some assertions which we believe to be either incorrect, misleading or unjustified.

Infix Function Notation

> "Any Haskell and Clean function can be used in infix form, if its name is enclosed between 's [back-ticks]" – Hegedus[Heg01b]

This assertion is only true for Haskell and not for Clean.

The Difference between Data and Newtype

> Regarding `data` and `newtype` declarations: "the only difference between the two is that the second one is implemented in a more efficient fashion" – Hegedus[Heg01b]

At the very least this quote is misleading. Firstly, the data constructor in a `newtype` declaration is strict in its one and only argument and a constructor in a `data` declaration, by default, is not strict in any of its arguments. Secondly, the pattern matching semantics for a `newtype` value is different to that of a `data` value.

Records and Field Labelled Types

> Regarding Haskell's field labelled types: "they can be handled as valid Clean records" – Hegedus[Heg01b]

Note that the labelled field syntax in Haskell is simply syntactic sugar surrounding an algebraic data type. To investigate the above remark we refer to the Clean language report, which states:

> "A record type is basically an algebraic data type in which exactly one constructor is defined" – Clean Language Report[Cl02]

Since a Clean record can only have one data constructor, then it can't "handle" an algebraic data type which may have many data constructors.

The use of the word "basically" in the Clean quote leads us to take the sentence in the context of providing an intuition for records rather than a formal definition. Indeed, the later statement "records cannot be used in a curried way" is taken to mean that if a record was treated as a data constructor then it could not be partially applied, thus distancing itself even further from an algebraic data constructor.

List Comprehensions

> Regarding list comprehensions: “In Clean, a filter always belong [sic] to one and only one generator” – Hegedus[Heg01b]

We take this sentence, possibly incorrectly, to mean that a filter may only refer to variables in its adjacent generator. However, the following valid Clean expression contradicts this:

```
[(i, j) \\ i <- [1, 2, 3], j <- [1, 2, 3] | i < j]
```

Notice that the filter, although adjacent to the generator `j`, refers to both `i` and `j`. In essence, we can’t see the issue that is raised in the article. We do however note that Haskell’s list comprehensions allow `let` statments in them where local variables may be declared and that Clean doesn’t appear to allow this.

Successor and Irrefutable Patterns

> Regarding successor and irrefutable patterns: “these patterns are obsolete so it’s not really important to deal with them” – Hegedus[Heg01b]

With successor patterns this justification is understandable as even the Haskell’98 report itself says:

> “Many people feel that `n+k` [successor] patterns should not be used. These patterns may be removed or changed in future versions of Haskell” – Haskell Language Report[Hs98]

However, irrefutable patterns are used to specify important constructs, such as the lazy nature of `let` expressions, in Haskell and so we do not consider them obsolete. Although, admittedly, explicit irrefutable patterns are probably not that commonly used in Haskell programs (but can be necessary for certain styles of programming).

Floating Point Precision Finally, the article states that double precision floating-point numbers are not available in Clean. Actually, it is *single precision* floating-point numbers that are not available. This is important, since single-precision numbers can be represented using a double-precision ones, but not vice-versa, and so there is less of an incompatibility than that implied in the article.

2.3 The Haskell Prelude for Clean

One would hope that once the Haskell Frontend for Clean had been built, all Haskell modules, such as the Prelude (which is used by almost all Haskell programs), could simply be copied across to the Clean library directory where they can be easily located by the compiler.

Unfortunately however, Simon notes in his article[Sim03] that the Haskell Prelude is not suitable to be handled directly by the Haskell to Clean Frontend. The first reason for this is due to some of the features of Haskell that are not handled by the Frontend:

> ”Unfortunately, structural differences and some extra features of Haskell (e.g. deriving) disappoint the hope of importing the Haskell Prelude by the Haskell front-end” – Simon[Sim03]

The second reason is due to the use of primitives in the Prelude:

> ”The Standard Prelude contains many primitives, i.e. functions and types that are not definable in Haskell and are implemented in a low-level, system-dependent way” – Simon[Sim03]

Simon proposes that both these problems can be overcome by translating the Haskell Prelude to Clean by hand. Indeed this proposal was carried out, and a mostly compatible Prelude written in Clean was produced.

However, we note that the latest version of the Frontend *does* import a Prelude written in Haskell. This Prelude is essentially taken from the Hugs[Hugs98] distribution with some minor modifications, and leaves the primitive type-signatures present while not providing definitions for them. This, of course, means that the Frontend must have been adapted to support Haskell's deriving clause and primitive functions.

Nevertheless, using the Prelude for Clean, we can provide some nice insight into how the I/O monad can be translated to Clean, and as a result, how the `main` function in Haskell can be translated to the `Start` function in Clean:

Haskell'98

```
main :: IO ()
main = do putStr "Name? "
          s <- getLine
          putStr ("Hello " ++ s)
```

Clean 2.1 using Prelude for Clean

```
Start :: *World -> *World
Start world = snd (doIO main world)

main :: IO Trivial
main = putStr ['Name? '] >>
       getLine >>= \s ->
       putStr (['Hello '] ++ s)
```

It is encouraging that so many issues have been resolved in the above example. Firstly, Clean's unique `World`[2] object can be used to implement Haskell's I/O monad. Secondly, the trivial type `()` in Haskell can be translated to a type called `Trivial` in Clean. Thirdly, Haskell strings can be translated to character lists in Clean. Finally, the monadic *do* notation in Haskell can be translated to the classical style in Clean.

2.4 Program Translation

In this project we are interested in the semantics-preserving translation between two programming languages. The use of the term *translation* in this context emphasises the similarity in meaning between the input and the output of the computation. In addition, it also implies that the input and output are instances (or sentences) of two different languages.

A more general form of translation where the languages could be the same, and whose meanings may vary, is termed *transformation*. For example, in program optimisation transformations, the input and output languages are the same although the meaning of the program will be preserved.

The general idea of transformation encompasses three main steps; first to understand the input (or put the input in a form that can be easily read and manipulated), secondly to perform the transformation

[2] Clean's `World` object can be used to interact with the "outside world". It is an abstract data type in much the same way as Haskell's I/O monad. Of course, the `World` can be descrutively updated since it's specifi ed as unique.

(or translation, for example, *hello* is *bonjour* in French), and thirdly to produce meaningful output (or display it in a human-readable form).

The first and last of these steps are commonly termed *parsing* and *pretty-printing* respectively. In the following sections, we will first review various approaches to parsing, pretty-printing, and program (or tree) transformation.

2.4.1 Parsing and Pretty Printing

It is envisaged that the source-to-source translator will need to perform two fairly complex tasks: to parse full Haskell'98 and pretty-print Clean 2.1. In Computer Science, and particularly in the area of functional programming, much research has been performed on these two tasks.

A common approach used in functional programming is to define a set of higher order functions (combinators) that capture the most common operations used in a particular application domain. These sets of combinators are often called Domain Specific Languages (DSLs), of which parsing and pretty-printing are both examples. A typical DSL allows rapid development of clear and concise programs. To illustrate this point we will show how *if*-expressions can be parsed and pretty-printed using Bird's parser combinators[Bir98] and Hughes's pretty-printing combinators[Hug95] respectively:

Parsing Example (*a la* Bird)	Pretty Printing Example (*a la* Hughes)

`parseIf =`
`  do symbol "if"; e0 <- exp`
`     symbol "then"; e1 <- exp`
`     symbol "else"; e2 <- exp`
`     return (IfExp e0 e1 e2)`
	`pp (IfExp e0 e1 e2) =`
	`  text "if" <+> pp e0 <+>`
	`  text "then" $$ nest 4 (pp e1) $$`
	`  text "else" $$ nest 4 (pp e2) $$`
	`  empty`

It is arguable if Bird's combinators are suitable, in terms of efficiency and completeness, to parse full Haskell'98 since they were developed primarily for teaching purposes. However, from the outside, the combinators appear very similar to those that have been developed in the PARSEC library which is described as an "industrial strength, monadic parser combinator library". Although not using PARSEC, the nhc98[nhc98] compiler has its parser defined by monadic parser combinators.

In a completely different approach, the Happy parser generator[Gil95], which generates bottom-up parsers from a BNF-like specification (in a similar style to Yacc), has been used to produce an efficient Haskell'98 parser written in Haskell. Incidently, this parser has been used in the popular documentation tool Haddock[Mar02] which creates fully-hyper-linked and marked-up source code in HTML format from a Haskell'98 program. The fact that Haddock is so widely used suggests that its parser must be fairly complete and reliable. Furthermore, the parser used by the Glasgow Haskell Compiler[GHC] has also been generated using Happy (indeed the Haddock parser is based on this one), although it also parses several extensions to Haskell'98.

In regard to pretty-printing, Hughes's combinators have been efficiently implemented by Peyton Jones[Pey97] and have subsequently been used to define a complete pretty-printer for Haskell'98. Incidently, this pretty-printer and the Happy-generated parser are both distributed as modules with GHC, and naturally, they both use the same abstract syntax of Haskell'98.

2.4.2 Transformation

A common approach for specifying transformations is known as syntax-directed transformation. Under this approach, each syntactic element in the input language is mapped to some syntactic element in the output language. Haskell, with its support for symbolic pattern matching, can specify such mappings very elegantly.

Indeed, in other applications, Haskell has been used to transform Haskell programs. For example, the Hat tracer[Chi02] transforms a given Haskell'98 program to one which calculates its own trace when executed.

The Tree Transformation Language (Txl [Cor03]) is a "programming language specifically designed to support computer software analysis and source transformation tasks". Txl allows the format of the input and output languages to be specified using an annotated BNF-like grammar definition, from which a parser and pretty-printer will be derived automatically. Transformations may then be specified using rules that state a pattern to be matched and replaced.

We refrain from giving an in-depth description of how Haskell and Txl can be used to specify program transformations, since this will be covered in a detailed comparison of the two languages in a later section (4.1) of this report.

2.5 Current Situation

Encouraged By showing how similar Haskell and Clean are, we have provided the necessary inspiration that a Haskell to Clean translator could be an achievable goal, without finding too many difficulties that would make such an effort not worthwhile.

We take further encouragement from the work of Hegedus who has documented several translation issues between the two languages, and also from Divianszky who has built a reasonably complete Haskell Frontend for the Clean compiler. Simon warns us that primitive definitions in the Haskell Prelude, and in particular the I/O monad, need to be dealt with specially. By writing the Haskell Prelude in Clean, Simon allows us to take a sneak preview of how the Haskell `main` function of type `IO ()` can be translated to the Clean `Start` function with the unique type `*World -> *World`.

Envisaging that a source-to-source translator requires a full Haskell'98 parser and Clean 2.1 pretty-printer, we note that such a parser and pretty-printer for Haskell'98 are present in the form of libraries distributed with GHC, and this parser's success in the Haddock documentation tool. In addition, parsing and pretty-printing combinator libraries (or Domain Specific Languages embedded in Haskell) have been thoroughly researched and developed, to the extent that they have been used in anger to define large-scale parsers (e.g. nhc98) and pretty-printers (e.g. the Haskell'98 pretty-printer with GHC).

Furthermore, Haskell has been used in various program transformation tools such as the Hat tracer, showing that its support for symbolic pattern matching makes it well-suited for such applications. Txl, the Tree Transformation Language, is designed specifically for the area of program transformation, and might too prove very suitable for defining translations.

More work to be done We have raised several issues, if not errors, regarding the work of Hegedus that leave us unsatisfied in several areas. Firstly, incorrect assertions lead us to believe that some issues

that have been raised have not been dealt with correctly (e.g. newtypes and field labelled types), or have, but with insufficient formalism. Secondly, some issues that have been raised have not been dealt with at all (e.g. irrefutable patterns and the deriving clause). Thirdly, the approach taken, i.e. adjusting the Frontend of Clean, may not be the best way to approach the problem. And finally, there is no evidence which convinces us that all the issues between the two languages have been discovered, documented, and resolved especially when considering the lack of documentation for the Haskell Frontend for Clean.

Chapter 3

Problem Analysis and Design

Recall from the review that the only previous investigation with documented findings relating to the translation of Haskell to Clean is that by Hegedus. Several issues, if not errors, were uncovered when this work was analysed in detail. But perhaps the most concerning point is that we were unconvinced that all the issues had indeed been raised. Therefore, it would seem unwise to base our entire translator purely around this previous work, rather it would more sensible to use it as a starting point. In addition to performing our own thorough investigation into issues in Haskell to Clean translation, we aim to provide, where appropriate, formal solutions to these issues (what we mean by "formal solutions" will be explained shortly).

Essentially the only way to perform such a thorough investigation is to refer to the Haskell language report, and for each construct described, check for the presence or absence of an equivalent construct or for semantic or syntactic differences by comparison with the Clean language report.

Indeed, this is the approach that we take in this chapter. However, rather than including an exhaustive list of every issue we come across, we only bring forward those issues that are of particular interest (the remaining issues are in Appendix A). We class an issue as "interesting" if

- it is not obvious,
- it has not been discovered or documented before,
- we cannot find a good solution for it,
- our proposed solution is not immediately obvious.

3.1 Formalising Translations

Since, in future sections of this chapter, we will be aiming to specify translations from Haskell to Clean it is worthwhile to first consider how we can do so effectively. We know from experience that it is difficult to write natural language solutions to technical problems without being verbose, ambiguous or unclear. Instead, it is often useful to use a model through which we can specify precisely what we mean with minimal clutter and repetition. As functional programmers, perhaps the most obvious model to use is a functional one. A functional model known as denotational semantics has been used with great success to define programming languages formally. Indeed, much of the notation that we introduce here is inspired from the definitions of formal semantics.

Meta-variable	Type
e	Expression
p	Pattern
d	Declaration
x,n,v,f	Variable
t	Type
op	Operator name
c	Constructor
rhs	Right-hand-side of definition
k	Numeric literal
s	Symbol table

Table 3.1: Meta-variable Convention. Note that plurals may also be used, e.g. *es* represents zero or more expressions.

The Haskell'98 report uses mathematical equalities to specify how its non-core constructs can be translated to the Haskell'98 kernel. For example:

$$\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{case } e_0 \texttt{ of True -> } e_1 \texttt{ ; False -> } e_2$$

Notice here that the concrete part of the construct is written in `typewriter` font, and the so-called meta-variables are in *italics*. The meta-variables e_0, e_1 and e_2 range over expressions. The convention is that if a variable begins with "e" then it represents any expression (a full convention is shown in Table 3.1).

The use of equalities is quite appropriate since the two sides of the equation are equal under the definition of Haskell. However, an equality no longer seems accurate if, for example, we are trying to say that Haskell `if`-constructs can be translated to Clean `if`-constructs since the two sides of the equation aren't "equal":

$$\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{if (}e_0\texttt{) (}e_1\texttt{) (}e_2\texttt{)}$$

More accurately, we are trying to say that the left-hand-side expression under the definition of Haskell is semantically equivalent to the right-hand-side expression under the definition of Clean. Perhaps a more appropriate notation is to define a translation function (for expressions) with the following type:

$$\mathcal{T}_{exp} : \text{HsExp} \rightarrow \text{ClExp}$$

That is, when given a Haskell expression the translation function (for expressions) will return a semantically equivalent Clean expression. Now the earlier translation of `if`-constructs can be specified as follows:

$$\mathcal{T}_{exp}[\![\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2]\!] \quad = \quad \texttt{if (}\mathcal{T}_{exp}[\![e_0]\!]\texttt{) (}\mathcal{T}_{exp}[\![e_1]\!]\texttt{) (}\mathcal{T}_{exp}[\![e_2]\!]\texttt{)}$$

Notice that the right-hand-side of this equation is quite verbose since we have to apply $\mathcal{T}_{exp}$ to all the meta-variables. To help avoid this syntactic clutter we will not, in future, explicitly specify that each meta-variable is to be translated. Instead we shall assume an implicit translation of each meta-variable on the right-hand-side by the translation function of the appropriate type (see Table 3.1).

In anticipating that some translations may require information that is stored in the symbol table, we make an adjustment to the type of the translation function so that it takes a symbol table as a parameter as well:

$$\mathcal{T}_{exp} : \text{HsExp} \rightarrow \text{Symtab} \rightarrow \text{ClExp}$$

3.2 Translating Expressions

In this section we raise issues with translating Haskell expressions to Clean expressions, and specify our solutions by defining the $\mathcal{T}_{exp}$ function. Expressions in both languages are formulas that can be reduced (or evaluated) to a value. Recall that both languages use a non-strict evaluation strategy (by default) and so our primary concern is to translate an expression so that its value is preserved.

3.2.1 Unary Minus

The Haskell syntax allows the unary minus operator which may be applied to an argument just as if it were any other unary (prefix) function. In Clean, such an operator is not allowed, and instead, the Clean syntax only allows numeric literals to be prefixed with the minus sign. For example, the following definition is valid in both Haskell and Clean:

```
minusOne = -1
```

However, the following definition is valid only in Haskell:

```
abs x | x >= 0 = x
      | x < 0 = -x
```

One solution would be to use the following translation:

$$\mathcal{T}_{exp}[\![-x]\!]s \quad = \quad \texttt{(0-}x\texttt{)}$$

However, the Haskell'98 report states that binary minus may be rebound by the module system by redefining binary minus locally and that unary minus is *not* affected by any local definition of binary minus. Instead the report states that unary minus is identical to the `negate` function defined in the Prelude. This means that even if `negate` is redefined locally, unary minus will still refer to the `negate` function in the Prelude.

As a result, this issue cannot be resolved at the level of expressions. Instead we must have some way to refer to the Prelude's `negate` function. This can be achieved by giving it another name that cannot possibly be used in the Haskell program, for example, a reserved word in Haskell that is not reserved in Clean. We notice that Clean identifiers are allowed to begin with the prime symbol and that this is not allowed in Haskell. This provides a convenient translation shown in Figure 3.1.

In fact, unary minus is not the only expression that refers to a specific function in the Prelude. So the backtick-prefixing approach could well be useful in other translations too.

3.2.2 List Syntax

In both Haskell and Clean the empty list is written `[]`. However, in Clean the "Cons" list construction is written `[h:t]` where `h` is the head of the list and `t` is the tail, whereas in Haskell no special syntax is used, and it is simply written `h:t`. For example:

Haskell'98	Clean 2.1
`[1, 2] == 1 : 2 : []` ⇒ `True`	`[1, 2] == [1 : [2 : []]]` ⇒ `True`

It may seem that a trivial syntactic translation will be sufficient to overcome the problem. However, we note that there is a precedence issue (due to the *fixity* of the constructor) which needs to be considered. To illustrate this, we consider two possible translations of a list expression:

$$\mathcal{T}_{exp}[\![\texttt{f 1 : []}]\!]s = \texttt{f [1 : []]}$$

and

$$\mathcal{T}_{exp}[\![\texttt{f 1 : []}]\!]s = \texttt{[f 1 : []]}$$

Under Haskell, the compiler (or interpreter) would correctly resolve the precedence issue (and essentially implement the second translation above). This step is commonly called *fixity resolution*. However, since the translator must choose one or the other, it too must be able to perform fixity resolution. This is not a trivial task and is not usually handled directly by most parsers.

It would be nice to avoid having to perform fixity resolution if at all possible, and so a different solution arises. This is to make use of Clean macros, which are like functions except that they are evaluated at compile-time rather than run-time. The idea is to avoid making the decision about where to put the brackets, and define a macro which looks more like Haskell's "Cons" constructor. This macro is shown in Figure 3.2 and is named `@` since it is an operator which is allowed in Clean but reserved in Haskell. Now we let the Clean compiler decide how to resolve the ambiguity so that the translator doesn't have to. For example:

$$\mathcal{T}_{exp}[\![\texttt{f 1 : []}]\!]s = \texttt{f 1 @ []}$$

Finally we note that definition of `@` is made possible only by Clean macros. If it were defined as a function then it couldn't be used in patterns. For example, the macro allows correct compilation of the following Clean code, whereas a function would raise an error:

```
head (x@xs) = x
```

3.2.3 Sections

Sections in Haskell'98 provide a convenient syntax for partial application of binary operators. For example, consider the definition of the `reciprocal` function and the `half` function:

```
reciprocal = (1 /)
half = (/ 2)
```

Here `reciprocal` is defined with a "left section" and `half` with a "right section". Both are not allowed in Clean, but have have a straightforward translation to the Haskell'98 kernel resulting in expressions that are accepted in Clean:

$$\mathcal{T}_{exp}[\![\texttt{-}e]\!]s \quad = \quad \texttt{`negate}\ e$$

Where `‘negate` points to the definition of `negate` in the Prelude.

Figure 3.1: Unary Minus Translation

$$\mathcal{T}_{pat,exp}[\![x\texttt{:}xs]\!]s \quad = \quad x\texttt{@}xs$$
$$\mathcal{T}_{pat,exp}[\![\texttt{(:)}]\!]s \quad = \quad \texttt{(@)}$$

Where the macro `@` is defined in Clean as:

```
(@) infixr 5
(@) h t :== [h:t]
```

Figure 3.2: List Constructor Translation

$$\mathcal{T}_{exp}[\![\texttt{(}op\ \ e\texttt{)}]\!]s \quad = \quad \texttt{((`flip}\ op\texttt{)}\ \ e\texttt{)}$$
$$\mathcal{T}_{exp}[\![\texttt{(}e\ \ op\texttt{)}]\!]s \quad = \quad \texttt{((}op\texttt{)}\ \ e\texttt{)}$$

Where `‘flip` points to the `flip` defined in the Prelude.

Figure 3.3: Section Translation

$(op\ e) = \lambda x \rightarrow x\ op\ e$
$(e\ op) = \lambda x \rightarrow e\ op\ x$
Where x is a variable that does not occur free in e.

The translation would be simple but for the fact that we must create a variable that does not occur free in e. Although this is not a major problem, a simpler translation that doesn't require such a unique variable is shown in Figure 3.3. Using these translations on `reciprocal` and `half` we get (note, the `flip` function has be "inlined"):

```
reciprocal = ((/) 1)
half = (((\ f x y -> f y x) (/)) 2)
```

3.2.4 Do Notation

do notation was introduced in Haskell'98 as syntactic sugar for specifying monadic expressions, and it is not supported in Clean. However, long before this syntactic sugar a classical style for laying out monadic code was used. This classical style is just as valid under Clean as it is under Haskell:

Haskell'98	Clean 2.1

`do putStr "Name? "`
`   s <- getLine`
`   putStr ("Hello " ++ s)`
	`putStr ['Name? '] >>`
	`getLine >>= \s ->`
	`putStr (['Hello '] ++ s)`

The Haskell'98 report defines how *do* notation may be de-sugared, resulting in valid Clean code too.

3.2.5 Arithmetic Sequences

Both Haskell and Clean provide a shorthand notation for specifying lists where the values in the list correspond to some arithmetic sequence. For example, the following is valid in both languages:

```
nats = [0 ..]
digits = [0 .. 9]
evens = [0, 2 ..]
oddsUpTo99 = [1, 3 .. 99]
```

Under both languages, the types of data that can appear in these expressions is any type which is a member of the `Enum` class. However, the `Enum` classes in the two languages are different, and as a result, the way in which arithmetic sequences is implemented is slightly different. Haskell's `Enum` class provides methods that correspond directly to each type of arithmetic sequence shown above whereas in Clean a set of polymorphic functions are defined over all instances of the `Enum` class. So, whereas a Haskell programmer is free to give an arbitrary definition for an arithmetic sequence of a user defined type, a Clean programmer is restricted to a predefined function.

Although the language support is very similar, the small issue raised above favours the translation to the Haskell'98 kernel as specified in the report (shown in Figure 3.4).

3.2.6 List Comprehensions

Support for list comprehensions (based on the familiar mathematical set comprehension notation) is provided in both languages, although they are called ZF-Expressions under Clean. We are familiar with the set comprehension notation for the cross product of two sets:

$$X \times Y = \{(x,y) \mid x \in X, y \in Y\}$$

List comprehensions are similar:

Haskell'98 – List Comprehension	Clean 2.1 – ZF-Expression
`cross xs ys =` `  [(x, y) | x <- xs, y <- ys]`	`cross xs ys =` `  [(x, y) \\ x <- xs, y <- ys]`

Some more differences in syntax can be seen when a boolean filter is added, for example in the definition of `ordCross`, which is just like `cross` except only ordered pairs are returned. This is illustrated by comparing the Haskell definition:

```
ordCross = [(x, y) | x <- xs, y <- ys, x < y]
```

with the Clean definition:

```
ordCross = [(x, y) \\ x <- xs, y <- ys | x < y]
```

It appears that the two forms of expression are only syntactically different. However, Haskell's list comprehensions allow `let` statements to appear as qualifiers, for example:

```
biRelate r xs ys = [r x y | let ps = cross xs ys, (x, y) <- ps]
```

Clean doesn't allow such a `let` statement, and so a suitable translation is not obvious. One approach is to use the translation to the Haskell'98 kernel (as defined in the Haskell'98 report) which yields valid Clean code too. However, since `let` statements are the only issue that we can see regarding a direct translation from list comprehensions to ZF-expressions, a hybrid approach could be used. Under this approach the list comprehension can be checked to see if it contains a let expression, and if it does then use the translation to the kernel, otherwise use a ZF-expression. From experience, we would imagine that the vast majority of list comprehensions don't use a `let` statement.

3.2.7 Expression type-signatures

Under Haskell the programmer may resolve internal-overloading by specifying the type of an expression as part of the expression. For example:

```
(fromInt 10) :: Float
```

`fromInt` is an overloaded function that converts an integer to a number (could be a `Float`, `Double`, `Int`, or `Integer`). Here the expression type-signature has specified that we want to convert `10` to a `Float`.

Clean doesn't support expression type-signatures, but the Haskell'98 report defines the following simple translation for them using `let` expressions:

$$\mathcal{T}_{exp}[\![e\ \texttt{::}\ t]\!]s \;=\; \texttt{let}\ x\ \texttt{::}\ t\texttt{;}\ x\ \texttt{=}\ e\ \texttt{in}\ x$$

However, we note that the variable x should be chosen so that it does not occur free in e (this fact has not been stated in the report). So in our translation (Figure 3.5) we use the variable ` for x since it isn't a valid Haskell identifier (but is valid in Clean). This variable cannot occur free in e provided that no other translation makes it a free variable.

3.2.8 Backtick notation

In Haskell, any prefix function or constructor can be enclosed in backticks and then used as an infix operator. For example, recall the `power` function which raises its first argument to the power of its second:

```
power 2 8 ⇒ 256
```

It could be used as an infix operator using backticks:

```
2 `power` 8 ⇒ 256
```

Furthermore, the infix operator may also be given a fixity (i.e. precedence and associativity):

```
infixr 8 `power`
```

so that the following expression behaves as expected:

```
2 `power` 2 `power` 2 - 1 ⇒ 15
```

In Clean, variable names can be used to define infix operators. For example:

```
(power) infixr 8 :: Int Int -> Int
```

and the following expression behaves as expected:

```
2 power 2 power 2 - 1 ⇒ 15
```

However, unlike Haskell's backticked functions, infix operators in Clean can't be used in the familiar prefix notation. For example, the Clean expression `power 2 8` will raise a compile-time error when `power` is defined as an infix operator. If `power` is to be used as a prefix operator then it must be done just as if it were a symbolic operator, i.e. `(power) 2 8`. It may seem that the following straightforward translation should be used for Haskell's infix notation:

$$\mathcal{T}[\![e1\ \ `f`\ \ e2]\!]s = (f\ \ e1\ \ e2)$$

Again we note that this translation is complicated by the fact that the translator would have to perform fixity resolution to implement it, and again we would like to avoid having to do this if at all possible. An alternative approach is to create a new infix operator in Clean for every binary infix function that is defined in Haskell. Since Clean variable names can contain (and begin with) backticks, a familiar looking notation can be used when defining this new operator:

Haskell'98	Translation to Clean 2.1

`infixr 8 `power``
`maxBin bits = 2 `power` bits - 1`
	`(`power`) infixr 8`
	`(`power`) = power`
	`maxBin bits = 2 `power` bits - 1`

This translation could get quite verbose since it will generate an infix operator for *every* binary function in the Haskell program. It might be neater to only generate the definition when it is required. This translation is illustrated by the following example:

Haskell'98	Translation to Clean 2.1

`infixr 8 ‘power‘`
`maxBin bits = 2 ‘power‘ bits - 1`
	`maxBin bits = 2 ‘power‘ bits - 1`
	`  where (‘power‘) infixr 8`
	`        (‘power‘) = power`

The latter translation is slightly more favourable since it means we needn't worry about exporting the new infix operators to other modules. This has been formalised in Figure 3.6.

3.3 Translating Patterns

In this section we raise issues with translating Haskell patterns to Clean patterns and specify our solutions by defining the $\mathcal{T}_{pat}$ function. Patterns in both languages can be used to match structured values and to bind variable names to their component parts. For example `x:xs` in Haskell is a pattern that matches any list with a head and a tail (and binds `x` to the head and `xs` to the tail).

The semantics of patterns in Haskell is defined using the `case` expression (which is also supported in Clean), and all patterns are eventually translated to the kernel using `case` expressions. In what follows we will often define translations on patterns in general by simply defining them on `case` expressions.

3.3.1 As-patterns

Sometimes it can be convenient to be able to both deconstruct a value to its component parts and still to able to refer to the value as a whole. For example, consider the function that takes a list and returns that list with its first element duplicated:

Haskell'98	Translation to Clean 2.1
`dup xxs@(x:xs) = x:xxs`	`dup xxs=:[x:xs] = [x:xxs]`

In the above definitions, `x` refers to the first element of the list, `xs` to the tail of the list, and `xxs` to the entire list. The as-pattern notation differs only due to syntax between the two languages. To see the semantics of as-patterns, replace every occurrence of the as-pattern on the right-hand-side with the actual pattern. This can be used to show the meaning of the above code for `dup`:

```
dup (x:xs) = x:x:xs
```

However, while as-patterns can be used in pattern bindings in Haskell, they cannot in Clean (there are other issues with pattern bindings too – see section 3.3.4). Here is a contrived but illustrative example of the issue:

$$\mathcal{T}_{exp}[\![\texttt{[}e_1\ \texttt{..]}]\!]s = \texttt{`enumFrom}\ e_1$$
$$\mathcal{T}_{exp}[\![\texttt{[}e_1\texttt{,}\ e_2\ \texttt{..]}]\!]s = \texttt{`enumFromThen}\ e_1\ e_2$$
$$\mathcal{T}_{exp}[\![\texttt{[}e_1\ \texttt{..}\ e_2\texttt{]}]\!]s = \texttt{`enumFromTo}\ e_1\ e_2$$
$$\mathcal{T}_{exp}[\![\texttt{[}e_1\texttt{,}\ e_2\ \texttt{..}\ e_3\texttt{]}]\!]s = \texttt{`enumFromThenTo}\ e_1\ e_2\ e_3$$

Where the backtick-prefixed names point to the non-backtick-prefixed names in the Prelude.

Figure 3.4: Arithmetic Sequences

$$\mathcal{T}_{exp}[\![e\ \texttt{::}\ t]\!]s = \texttt{let `}\ \texttt{::}\ t\texttt{; ` =}\ e\ \texttt{in `}$$

Figure 3.5: Expression Type-signature Translation

$$\mathcal{T}_{decl}[\![f\ \ ps\ \texttt{=}\ e\ \texttt{where}\ ds]\!]s = f\ \ ps\ \texttt{=}\ e\ \texttt{where}\ ds\texttt{;}ds'$$
$$\mathcal{T}_{exp}[\![\texttt{let}\ ds\ \texttt{in}\ e]\!]s = \texttt{let}\ ds\texttt{;}ds'\ \texttt{in}\ e$$

Where

$$ds' = \textit{map backtickDef (getInfixNames e)}$$
$$\textit{backtickDef n} = \textit{fixityDef n; infixDef n}$$
$$\textit{fixityDef n} = \texttt{(`}n\texttt{`)}\ \textit{assoc}\ \ \textit{prec}$$
$$\textit{(assoc, prec)} = \textit{getFixity s n}$$
$$\textit{infixDef n} = \texttt{(`}n\texttt{`) =}\ n$$

getFixity s n returns the fixity of the of the symbol n as stored in the symbol-table s, or if the symbol is not found, then it returns the default fixity. *getInfixNames e* returns the names of all backticked functions within the expression e (but ignoring nested `let` expressions).

Figure 3.6: Backtick Translation

$$\mathcal{T}_{pat}[\![n\texttt{@}p]\!]s = n\texttt{=:}p$$
$$\mathcal{T}_{decl}[\![n\texttt{@}p\ \texttt{=}\ rhs]\!]s = n\ \texttt{=}\ rhs$$
$$p\ \texttt{=}\ n$$

Figure 3.7: As-pattern Translation

Valid Haskell'98	Invalid Clean 2.1
`f = x:xxs` `  where xxs@(x:xs) = [1, 2, 3]`	`f = [x:xxs]` `  where xxs=:[x:xs] = [1, 2, 3]`

There are two possible solutions to this issue, both of which involve splitting the pattern binding into two definitions: one that binds the name of the as-pattern and the other which binds the pattern. One approach is as follows:

$$\mathcal{T}_{decl}[\![n\texttt{@}p \;\texttt{=}\; rhs]\!]s \;=\; \begin{array}{l} p \;\texttt{=}\; rhs \\ n \;\texttt{=}\; patToExp\;\; p \end{array}$$

Here the function *patToExp* converts a pattern to an expression. An alternative translation, which we prefer since it doesn't require *patToExp*, is:

$$\mathcal{T}_{decl}[\![n\texttt{@}p \;\texttt{=}\; rhs]\!]s \;=\; \begin{array}{l} n \;\texttt{=}\; rhs \\ p \;\texttt{=}\; n \end{array}$$

We illustrate this latter translation on the function `f` (defined above):

```
f = [x:xxs]
  where xxs = [1, 2, 3]
        [x:xs] = xxs
```

The full translation is shown in Figure 3.3.1.

3.3.2 Irrefutable Patterns

Haskell provides a notation that can be used to explicitly mark a pattern match as *irrefutable*, i.e. even if the pattern doesn't match a value, the evaluation will proceed as if it had matched. Here is a contrived but illustrative example of an irrefutable pattern:

```
dup' xxs@(~(x:xs))
  | xxs == [] = []
  | otherwise = x:xxs
```

In `dup'` the pattern `x:xs` will match the empty list because it has been marked as irrefutable. Normally this pattern would not match the empty list since it represents a list with a head `x` and a tail `xs`.

Clean doesn't have any equivalent notation, however we note that Clean does support `let` expressions which, in the Haskell'98 report, are defined using irrefutable patterns. The implicit irrefutable nature of `let` expressions is shown in the code below which is valid under both languages (but for the syntax of `if`):

```
dup'' xxs = let (x:xs) = xxs in
  if xxs == [] then [] else x:xxs
```

The `let` expression is lazy: it will only evaluate the pattern if it needs to. This describes exactly the semantics of irrefutable patterns which motivates the translation defined in Figure 3.8. We illustrate it by translating the `dup'` function defined above:

```
dup` xxs=:`irr_xxs
```

```
  | xxs == [] = let [x:xs] = ‘irr_xxs in []
  | otherwise = let [x:xs] = ‘irr_xxs in [x:xxs]
```

3.3.3 Successor Patterns

Haskell provides the notion of successor patterns to aid the inductive definition of functions over integers. Here is an example of a successor pattern used in the recursive definition of the function `take n xs` which returns the first `n` items of the list `xs`:

```
take (n+1) (x:xs) = x : take n xs
```

Here `n+1` is the successor pattern. Since Clean doesn't support successor patterns we will have to derive a translation. To do this we consider the precise semantics of successor patterns as stated in the Haskell'98 report.

> "Matching an $n+k$ pattern (where n is a variable and k is a positive integer literal) against a value v [sic] succeeds if $x \geq k$, resulting in the binding of n to $x-k$, and fails otherwise."
> – Haskell'98 Report[Hs98]

Firstly we note that the pattern match only succeeds if the value being matched x is greater than or equal to the integer constant k. This can be achieved by guarding the pattern match with the appropriate condition (i.e. $x \geq k$). Secondly the successor variable n must be bound to the value being matched minus the k. This can be achieved trivially using a `let` expression. This translation is formalised in Figure 3.9, and is illustrated by applying it to the recursive definition of `take`:

```
take v [x : xs]
  | (v >= 1) = let n = v - 1 in [x : take n xs]
```

3.3.4 Pattern Bindings

Both Haskell and Clean allow pattern bindings, whereby variables in some pattern may be bound to certain parts of the value of some expression. For example, in an alternative definition of `take` might use `splitAt` and a pattern binding:

```
take n xs = left
  where (left, right) = splitAt n xs
```

Here, the declaration in the `where` clause is a pattern binding. The problem is that the semantics of pattern bindings is a lot more restrictive in Clean than in Haskell. We have already resolved the issue regarding the fact that pattern bindings cannot be "as-patterned" (see section 3.3.1), however there are several other issues:

- In Clean, top-level pattern bindings are not allowed. Rather, they can only be used within a local scope e.g. in a `where` clause or a `let` expression.
- In Haskell, pattern bindings, like function definitions, may be guarded and have local definitions under a `where` clause. This is not the case in Clean.
- In Clean, the pattern in the binding must not contain any 0-arity data constructors (i.e. a constructor that takes no arguments). For example, the following bindings contain 0-arity data constructors:

$$
\begin{aligned}
\mathcal{T}_{exp}[\![\texttt{case}\ e_0\ \texttt{of}\ p\ \texttt{->}\ e_1]\!]s \;=\; & \texttt{case}\ v\ \texttt{of}\ p'\ \texttt{->} \\
& \quad \texttt{let}\ p_0\ \texttt{=}\ x_0 \\
& \qquad \vdots \\
& \qquad p_n\ \texttt{=}\ x_n\ \texttt{in}\ e_1
\end{aligned}
$$

Where p_0 to p_n are all the irrefutable patterns in p and x_0 to x_n are unique variable names that do not occur free in e_1. p' is like p except that sub-patterns p_0 to p_n have been replaced with the variables x_0 to x_n.

Figure 3.8: Irrefutable Pattern Translation

$$
\begin{aligned}
\mathcal{T}_{exp}[\![\texttt{case}\ v\ \texttt{of}\ p\ \texttt{->}\ e]\!]s \;=\; & \texttt{case}\ v\ \texttt{of}\ p' \\
& \quad \texttt{|}\ x_0\ \texttt{>=}\ k_0\ \texttt{\&\&} \\
& \qquad \vdots \\
& \quad\ x_n\ \texttt{>=}\ k_n\ \texttt{->}\ \texttt{let}\ n_0\ \texttt{=}\ x_0\ \texttt{-}\ k_0 \\
& \qquad\qquad \vdots \\
& \qquad\qquad n_n\ \texttt{=}\ x_n\ \texttt{-}\ k_n\ \texttt{in e}
\end{aligned}
$$

Where p' is like p except that all successor patterns in p have been replaced by the unique variables x_0 to x_n (which do no occur free in e), and n_0 to n_n and k_0 to k_n are the corresponding successor variables and constants respectively.

Figure 3.9: Successor Pattern Translation

$$
\begin{aligned}
\mathcal{T}_{decl}[\![p\ \texttt{=}\ e]\!]s \;=\; & x\ \texttt{=}\ e \\
& x_0\ \texttt{=}\ \texttt{case}\ x\ \texttt{of}\ p\ \texttt{->}\ x_0 \\
& \quad \vdots \\
& x_n\ \texttt{=}\ \texttt{case}\ x\ \texttt{of}\ p\ \texttt{->}\ x_n
\end{aligned}
$$

Where $p\ \texttt{=}\ e$ is either a top-level pattern binding or p contains one or more 0-arity data constructors. x_0 to x_n are all variables present in the pattern p. x is a new variable that does not clash with any other names in scope.

Figure 3.10: Pattern Binding Translation

```
[x] = tail [1, 2]  -- Pattern contains []
(1:xs) = [1, 2]    -- Pattern contains 1
```

The Haskell'98 report (section 4.4.3.2) describes a suitable translation to remove the local `where` clause and guards from a pattern binding (and the resulting code is valid in Clean). This resolves the second issue stated above.

A possible solution for the 0-arity constructor issue is to replace the pattern-binding with one that removes the 0-arity constructors and by moving the original pattern to a case expression on the right-hand-side.

$$\mathcal{T}_{decl}[\![p \texttt{ = } e]\!]s \quad = \quad (\textit{varTuple } p) \texttt{ = case } e \texttt{ of } p \texttt{ -> } (\textit{varTuple } p)$$

The function *varTuple p* returns a tuple containing all the variables that appear in the pattern *p*. Since under Haskell the pattern binding cannot fail to match, moving the pattern to the right-hand-side is safe. Here is an example using this translation:

Haskell'98	Translation to Clean 2.1

`xs = y where (1:y:ys) = [1, 2, 3]`	`xs = y`
	`  where (y, ys) =`
	`    case [1, 2, 3] of (1:y:ys) ->`
	`      (y, ys)`

To resolve the top-level binding issue it is necessary to extract all the variables from the pattern and to define each one separately. We illustrate this translation on the definition of `xs` above:

```
xs = y
  where y = case [1, 2, 3] of (1:y:ys) -> y
        ys = case [1, 2, 3] of (1:y:ys) -> ys
```

There are three points to notice about this translation. Firstly, that it will also resolve the 0-arity constructor issue. Secondly, that the resulting code is less efficient (since the expression in the `case` expression is evaluated multiple times). Thirdly, that it allows a type-signature to be specified for each variable in the pattern which is important since Haskell allows type-signatures for variables in pattern bindings and Clean doesn't. Since the efficiency issue can be resolved by extracting the expression in the `case` expression outwards (so that it is only evaluated once), we opt to use this approach and scrap our previous 0-arity constructor solution. We formalise this in Figure 3.10.

To summarise issues with pattern bindings and as-patterns, let us consider the translation of the following function which yields the famous Fibonacci sequence:

```
fib@(1:tfib) = 1 : 1 : [a + b | (a, b) <- zip fib tfib]
```

Firstly we might resolve the as-pattern using the translation shown in Figure 3.7:

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib tfib]
(1:tfib) = fib
```

Secondly we might resolve the pattern binding (which is both top-level and contains a 0-arity constructor) using the translation shown in Figure 3.10:

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib tfib]
x = fib
```

```
tfib = case x of (1:tfib) -> tfib
```

This code can now be easily translated to Clean (using other translations that we have already defined).

3.3.5 Prefixed Operators in Patterns

Under both Haskell and Clean constructors are just functions and so infix constructors can be used in a prefix form by wrapping them in parentheses. However, whereas Haskell extends this notation to patterns (as illustrated by the following example), Clean does not:

```
head ((:) x y) = x            -- Valid in Haskell but not Clean
```

Since such operators in patterns cannot be curried, and since function application must be completely unambiguous (i.e. no fixity resolution performed), the prefix operator can just be made infix by the translation shown in Figure 3.11.

3.3.6 Backticked Constructors in Patterns

Just like Haskell allows any prefix function to be used in an infix form (by wrapping it in backticks), Haskell also allows such notation to be used on constructors in both patterns and expressions. Of course we have already mentioned that Clean doesn't support this backtick notation. For example:

```
head (x `Cons` xs) = x
```

Where we assume that `Cons` is defined elsewhere. With this issue we cannot, as before, resolve backticked functions as and when they are used. Instead it seems necessary to define an backticked version of every binary constructor ever defined. Furthermore, in recalling an old trick of using a macro rather than a function, we will be able to use the backticked version in both patterns and expressions. The full translation is formalised in Figure 3.12.

3.4 Translating Type System Issues

3.4.1 Built-in Types

A character type (`Char`), finite-precision integer type (`Int`) and boolean type (`Bool`) are present as predefined types in both languages and are compatible.

Haskell supports single and double-precision floating point types (`Float` and `Double` respectively) whereas Clean only supports the latter (named `Real`). The obvious translation is to convert both `Floats` and `Doubles` to `Reals`, and the main difference in the resulting Clean program is that the result could be more precise than in the Haskell program (if a `Float` is translated to a `Real`).

Haskell also includes support for arbitrary-precision integers via the `Integer` type. Clean doesn't have an equivalent type and so one would have to be defined by the translator. The Windows version of the Clean compiler comes with an "Extended Arithmetic" library which defines an arbitrary precision integer type called `BigInt`. This can be used in the translation of the `Integer` type.

Both languages support a string type (`String`), although in Haskell it is represented as a list of characters and in Clean as an array of characters. The simplest translation is to define a new "list of characters"

type in Clean (named `CharList`), and translate all `String`s to `CharList`s and then all string literals, e.g. `"hello"` to `['hello']`.

Finally Haskell has a pre-defined nullary type denoted `()` which contains only one value also denoted `()`. An equivalent type `Null` could be defined in Clean:

```
:: Null = Null
```

However, we must then be sure that any indentifers with the same name in the Haskell program (i.e. `Null`) get appropriately renamed to avoid name conflicts.

3.4.2 Curried Type-signatures

In Clean, the type of a function may be specified in a curried or uncurried form:

Curried	Uncurried
`mkPair :: a -> b -> (a, b)` `mkPair a = \b -> (a, b)`	`mkPair :: a b -> (a, b)` `mkPair a b = (a, b)`

Notice here, that if the curried type is specified, then a curried form of definition must be given. Likewise, if the uncurried type is specified then the uncurried form of definition must be given.

We will call the number of arguments that aren't separated by a "`->`" the *arity* of the type. For example, the arity of the curried type above is 1, and of the uncurried type is 2. In Clean, the arity of the type *must* be equal to the number of arguments specified on the left-hand-side in the corresponding function definition. We also note that the 0-arity type of the fully curried definition is specified in Clean as follows:

```
mkPair :: (a -> b -> (a, b))
mkPair = \a b -> (a, b)
```

The problem is that Haskell does not enforce such a relationship between a function definition and its type signature. Indeed, *all* type signatures in Haskell are "curried", and it doesn't care how many arguments are or aren't specified on the left-hand-side of a function definition. To resolve this problem one could translate all function definitions (that take one or more arguments) to equivalent functions that only take one argument (making the Haskell type signature correct). Alternatively, and a lot more naturally, one could translate the type-signatures to have an arity equal to the number of arguments specified in the corresponding function definition. The latter translation is formalised in Figure 3.13.

There is syntactic issue that arises from uncurried types. Since some arguments are separated by space instead of an arrow a problem arises when an argument is not a 0-arity type. For example, suppose we have the 1-arity type `List a` that represents a list of values with type `a`, then the following signature for `map` illustrates the problem:

```
map :: (a -> b) List a -> List a
```

Now it appears that `map` takes three arguments, and so `List a` needs to be parenthesised (in a curried type there is no such problem, since every argument is separated by an arrow).

A further problem is introduced when type classes are considered, since the type signature could represent *several* function definitions, all of which may be defined with a different numbers of arguments on

$$\mathcal{T}_{pat}[\![(op)\ \ e_0\ \ e_1]\!]s \quad = \quad (e_0\ \ op\ \ e_1)$$

Figure 3.11: Prefixed Pattern Translation

$$\begin{array}{rcl}\mathcal{T}_{decl}[\![\texttt{data}\ t\ \ xs\ \texttt{=}\ cs]\!]s & = & \texttt{data}\ t\ \ xs\ \texttt{=}\ cs \\ & & (\text{`}c_0\text{`})\ \ assoc\ \ prec \\ & & (\text{`}c_0\text{`})\ \ \texttt{:==}\ c_0 \\ & & \vdots \\ & & (\text{`}c_n\text{`})\ \ assoc\ \ prec \\ & & (\text{`}c_n\text{`})\ \ \texttt{:==}\ c_n\end{array}$$

Where c_0 to c_n are all the binary constructors in cs. *assoc* and *prec* are obtained by calling *getFixity* as defined in Figure 3.6. In addition, when a constructor is exported in the module system, so too must the new macros.

Figure 3.12: Backticked Constructor Translation

$$\mathcal{T}_{decl}[\![f\ \ \texttt{::}\ \ \ t]\!]s \quad = \quad f\ \ \texttt{::}\ \ \ t'$$

Where t' is a type definition like t except that the leftmost *numArgs f* arguments are separated by space rather than `->`. *numArgs f* returns the number of arguments on the left-hand-side of the definition of f. If *numArgs f* is 0 then t' should be enclosed in parenthesis.

Figure 3.13: Curried Type Translation (a)

$$\mathcal{T}_{decl}\left[\!\!\left[\begin{array}{c}\texttt{class}\ t\ \texttt{where} \\ f_0\ \texttt{::}\ \ t_0 \\ \vdots \\ f_n\ \texttt{::}\ \ t_n\end{array}\right]\!\!\right]s \quad = \quad \begin{array}{l}\texttt{class}\ t\ \texttt{where} \\ \quad f_0\ \texttt{::}\ \ (t_0) \\ \quad\quad \vdots \\ \quad f_n\ \texttt{::}\ \ (t_n)\end{array}$$

$$\mathcal{T}_{decl}\left[\!\!\left[\begin{array}{l}\texttt{instance}\ c\ t\ \texttt{where} \\ \quad f_0\ \ ps_0\ \texttt{=}\ rhs_0 \\ \quad\quad \vdots \\ \quad f_n\ \ ps_n\ \texttt{=}\ rhs_n\end{array}\right]\!\!\right]s \quad = \quad \begin{array}{l}\texttt{instance}\ c\ t\ \texttt{where} \\ \quad f_0\ \texttt{=}\ \text{`}\ \texttt{where}\ \text{`}\ \ ps_0\ \texttt{=}\ rhs_0 \\ \quad\quad \vdots \\ \quad f_n\ \texttt{=}\ \text{`}\ \texttt{where}\ \text{`}\ \ ps_n\ \texttt{=}\ rhs_n\end{array}$$

Figure 3.14: Curried Type Translation (b)

the left-hand-side. In this case it seems that we must apply a translation to the definitions rather than the type-signatures. If all type-signatures in a type class are enclosed in parenthesis (i.e. meaning that the function is fully curried) then when it comes to translate an instance of that class we can just adjust the definition so as to use no arguments on the left-hand-side. For example:

Before Translation	After Translation

`class Add a where`
`  add :: a -> a -> a`
 	
`instance Add Int where`
`  add x y = x + y`
	`class Add a where`
	`  add :: (a -> a -> a)`
	
	`instance Add Int where`
	`  add = ‘`
	`    where ‘ x y = x + y`

This translation is formalised in the general case in Figure 3.14.

3.4.3 Newtypes

It is often desirable to create a new type that is identical to an existing one, e.g. a natural number type defined as an integer. However, a type synonym might not be correct since we may wish to define a natural number under a new instance of the `Num` class (and, for example, make a function give an error message when the number becomes negative). Haskell allows a `newtype` declaration as follows:

```
newtype Natural = N Integer
```

The problem is that Clean doesn't support `newtype` declarations. However, it would seem that a simple translation to a `data` declaration is possible:

```
data Natural’ = N’ Integer
```

However, a `newtype` has slightly different semantics. Firstly, a newtype constructor is defined to be strict to improve efficiency. So, an improved translation is to make the constructor strict using a "strictness annotation":

```
data Natural’ = N’ !Integer
```

Secondly, pattern matching on `newtyped` values behaves slightly differently. We illustrate this in the following definitions:

```
f (N i) = 1
f’ (N’ i) = 1
```

We observe that `f` $\perp$ evaluates to 1 whereas `f’` $\perp$ evaluates to $\perp$. The new pattern matching semantics are as follows: if the pattern inside the `newtype` constructor is irrefutable then the pattern including the newtype constructor is also irrefutable. So the `newtype` pattern in *f* needs to be made irrefutable:

```
f’’ ~(N’ i) = 1
```

In contrast, the following function which contains a refutable pattern inside a `newtype` constructor should not be made irrefutable:

```
f2 (N (n + 1)) = 1
```

This translation is formalised in figure 3.15.

3.4.4 Multi-Variable Type-signatures

Haskell allows a single type-signature to be used to define the type of a list of variables as follows:

```
f, g :: Int
```

In Clean a type-signature can only be given to a single variable in the following form:

```
f :: Int
g :: Int
```

This translation has be formalised for the more general case in Figure 3.16.

3.4.5 Default Class Methods

A class, by intuition, is a set of related functions that operate over a large number of different data types. Sometimes these functions are related to such an extent that it seems unnecessary to always have to define them all. For example, the `Eq` class contains functions for equality (`==`) and inequality (`/=`) which are obviously closely related. Haskell allows a default definition in a class so that it may be omitted in the instance declaration:

```
class Eq a where
  (==) :: a -> a -> a
  (/=) :: a -> a -> a
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Here a default definition is given for both functions. This means that either a definition for `==` or `/=` must be given (or both). Any function which is omitted in an instance declaration will simply take its default class definition.

Clean supports a similar but different feature in that it allows macro definitions in class declarations. This functionality differs mainly in that when a default macro is given in Clean, it *cannot* be overridden in an instance declaration. Clearly, with this restriction (not to mention that macros are different to functions), there is quite an incompatibility between the languages.

To overcome the problem we really need to implement Haskell's mechanism for default methods. This can be done in two steps. Firstly, fill in the omitted definitions in an instance declaration with the corresponding class defaults. Secondly, remove all function definitions from class declarations. This translation is formalised in Figure 3.17.

3.4.6 Numeric Literals and Default Declarations

In Haskell integer literals are overloaded, i.e. the literal `1` can be interpreted as either an `Int`, `Integer`, `Float` or `Double`. Haskell achieves this by implicitly passing the literal to the `fromInteger` function of the following type:

```
fromInteger :: Num a => Integer -> a
```

$$\mathcal{T}_{decl}[\![\texttt{newtype } t_0 \texttt{ = } c \;\; t_1]\!]s \;=\; \texttt{data } t_0 \texttt{ = } c \;\; \texttt{!} t_1$$

$$\mathcal{T}_{pat}[\![c \;\; p]\!]s \;=\; \begin{cases} \texttt{\~{}(} c \;\; p \texttt{)} & \text{if } \mathit{isIrrefutable}\ p \text{ and } \mathit{isNewtype}\ s\ c \\ c \;\; p & \text{otherwise} \end{cases}$$

Where *isIrrefutable p* is true if *p* is an irrefutable pattern and false otherwise. *isNewtype s c* returns true if the constructor *c* is a `newtype` constructor in the symbol table *s*.

Figure 3.15: Newtype Translation

$$\begin{aligned} \mathcal{T}_{decl}[\![f_0 \texttt{,} \;\cdots\texttt{,}\; f_n \;\texttt{::}\; t]\!]s \;=\; & f_0 \;\texttt{::}\; t \\ & \vdots \\ & f_n \;\texttt{::}\; t \end{aligned}$$

Figure 3.16: Multi-Variable Type-Signature Translation

$$\mathcal{T}_{decl}[\![\texttt{class } c \;\; t \texttt{ where } ds]\!]s \;=\; \texttt{class } c \;\; t \texttt{ where } ds'$$

$$\mathcal{T}_{decl}[\![\texttt{instance } c \;\; t \texttt{ where } ds]\!]s \;=\; \texttt{instance } c \;\; t \texttt{ where } ds'$$

Where in the class translation, *ds'* is like *ds* except that all function definitions have been removed. In the instance translation, *ds'* is like *ds* except that it also includes the default definitions corresponding to any type-signatures in the class *c* (determined using the symbol table *s*) which are not defined in *ds*.

Figure 3.17: Default Class Method Translation

Since, in Clean, the literal `1` isn't overloaded and is of type `Int`, there is a slight issue here. So integer literals in Clean should be overloaded by explicitly passing them to `fromInteger`.

Haskell provides a mechanism to resolve internal overloading of a numeric expression using the `default` declaration. Every Haskell module assumes the following `default` declaration unless another is explicitly specified:

```
default (Integer, Double)
```

If the type of an expression is ambiguous then the default is used, e.g. the type of the expression `1 + 1` would default to `Integer` (and the addition function that operates over `Integers` would be called). In previous versions of Clean a single default for a class could be given, however this feature has now been removed in Clean 2 (due to a syntax conflict with a new feature). There is no obvious way to resolve this issue without disambiguating the Haskell source using a type-checker.

3.4.7 Automatic Instance Derivation

Recall that Haskell includes a facility whereby instances of a common set of classes (e.g. the `Eq` class) can be derived automatically for user-defined types. These classes are: `Eq`, `Ord`, `Enum`, `Read`, `Show` and `Bounded`. Clean provides a more general facility[1] by allowing the user to define their own generic functions, i.e. functions which operate over all types. A generic library (called `GenLib`) exists that includes the generic functions `gEq` (for equality), `gParse` (like the `read` function of the `Read` class, `gPrint` (like the `show` function of the `Show` class) and `gLexOrd` (like the `compare` function of the `Ord` class).

`GenLib` inspires confidence that the deriving of classes `Eq`, `Ord`, `Read` and `Show` can be easily translated to instantiations that use generic functions. For example:

Before Translation	After Translation

`data List a = Nil`
`            | Cons a (List a)`
`  deriving (Eq)`
	`:: List a = Nil | Cons a (List a)`
	`derive gEq List`
	`instance Eq List where`
	`  (==) x y = gEq{|*|} x y`

To automatically derive the `Enum` class, two generic functions would have to be developed so that the minimum `Enum` instantiation of `toEnum` (converts an integer to a corresponding construction of the user-defined type) and `fromEnum` (opposite of `toEnum`) could be given. Likewise, to derive the `Bounded` class two generic functions would have to be developed so that the functions `minBound` and `maxBound` could be defined on any type.

Instead of using Clean's generics one might instead use a pre-processing tool, such as DrIFT (which was used before the `derive` clause was integrated into Haskell), which would automatically generate the code for the instance declaration.

We admit that while inspiration has been given here on how the deriving clause can be translated, a fully formalised translation has not been given.

[1] At the time of writing, Clean's generics facility only works in the Windows version of the compiler.

3.4.8 Field Labelling

Often it can be more convenient to refer to fields of a data constructor by a name rather than by argument position and so Haskell provides a field labelling syntax. For example, a bibliography entry might be defined as:

```
data BibEntry = Book { title, author, ISBN :: String, pages :: Int }
              | Thesis { title, author, school :: String }
```

The Haskell'98 report defines a translation for field labelling constructs to the Haskell'98 kernel, resulting in code which is also valid in Clean.

Clean has a similar feature which it calls records. However, a Clean record is a more restricted form of an algebraic data type (whereas in Haskell, field labelled data types *are* just algebraic data types). The restriction has two major implications: firstly that a Clean record may have only one constructor, and secondly that this constructor (if it can be called a constructor) is not like a typical function as it requires special syntax to be used, and cannot be partially applied. Here is an example Clean record (for address book entries):

```
:: AddrEntry = { name, address :: String, age :: Int }
```

Now we can observe the difference between the construction of Clean records and Haskell field labelled types:

<table>
<tr><th>Haskell'98</th><th>Clean 2.1</th></tr>
<tr><td><pre>
Thesis {title = "T",
        author = "A",
        school = "S"}
</pre></td><td><pre>
{AddrEntry | name = "N",
             address = "A",
             age = 10}
</pre></td></tr>
</table>

The `Thesis` constructor can be treated just like any other function, for example, partial application:

```
bobsThesis = Thesis "Bob's Title" "Bob"
```

In contrast, `AddrEntry` can't be treated like any other function, and data of this type must be created using the special syntax.

It is not immediately obvious how a Clean record could be easily used to represent a Haskell labelled field data type, primarily for the fact that the latter may have several constructors, whereas the former may only have one. This is not to say that it is not possible, just that such a translation wouldn't be very clear. Perhaps a compromise is to only translate single-constructor labelled-types to Clean records, although the other issues (i.e. partial application of constructor) would still have to be resolved (and maybe this isn't so difficult: just create a macro to mimic the behaviour of the Haskell data constructor.

Nevertheless, we have settled for the translation as defined by the Haskell'98 report.

3.4.9 Empty Classes

If a class is completely defined in terms of other classes, i.e it does not provide any specific functionality of its own, then it might be defined as follows in Haskell:

Before Translation	After Translation
`class (Eq a, Enum a) => EqEnum a`	`class EqEnum a | Eq a & Enum a`

In this contrived example the `EqEnum` class is a child class of parent classes `Eq` and `Enum` and represents pure combination of its parents. However, whilst in Haskell an instance declaration still needs to be given for types to become members of `EqEnum`, in Clean it is assumed and so need not be written explicitly. To resolve this issue, `instance` declarations without a `where` clause should be ignored by the translator, provided that the corresponding class has no default methods. If it does have default methods, then the normal translation for these, as described in section 3.4.5, should be carried out.

3.5 Declaration Translation

3.5.1 Order of Declarations

In Haskell the type-signature of a function can be placed anywhere within the same scope as the function definition, whereas in Clean it must appear directly above the function definition. The translation is simply one of re-ordering and is formalised in Figure 3.18.

3.5.2 Infix Definitions

In Haskell functions may be defined in an infix form, e.g. in the definition of append (++):

```
[] ++ ys = ys
```

In addition, Haskell also allows the following definition of sequential composition (`.`):

```
(f . g) x = f (g x)
```

Since Clean doesn't support this notation, both forms should be translated into the familiar prefix notation. A formal translation is given in Figure 3.19.

3.5.3 Fixity Declarations

In both languages the fixity of an operator (i.e. its precedence and associativity) may be specified. For example, here is the fixity declaration for the append operator (which is right-associative and has a precedence of 5):

Haskell'98	Clean 2.1
`infixr 5 ++`	`(++) infixr 5`

There is only a syntactic difference between the two languages in the example above. However, there are three other differences:

- In Clean, the fixity of an operator must be integrated into the type-signature of that operator, whereas in Haskell it must appear on its own (as in the example above). If a type-signature is not specified, then the fixity declaration must appear on its own in Clean. For example:

$$\mathcal{T}_{decl}\left[\!\!\left[\begin{array}{l} f\ \ ps\ \texttt{=}\ rhs \\ \vdots \\ f\ \texttt{::}\ \ t \end{array}\right]\!\!\right] s \quad = \quad \begin{array}{l} f\ \texttt{::}\ \ t \\ f\ \ ps\ \texttt{=}\ rhs \end{array}$$

Where the type-signature could also be above the function definition too.

Figure 3.18: Declaration Order Translation

$$\begin{array}{rcl} \mathcal{T}_{decl}[\![p_0\ \ op\ \ p_1\ \texttt{=}\ rhs]\!]s & = & \texttt{(}op\texttt{)}\ \ p_0\ \ p_1\ \texttt{=}\ rhs \\ \mathcal{T}_{decl}[\![\texttt{(}p_0\ \ op\ \ p_1\texttt{)}\ \ ps\ \texttt{=}\ rhs]\!]s & = & \texttt{(}op\texttt{)}\ \ p_0\ \ p_1\ \ ps\ \texttt{=}\ rhs \end{array}$$

Figure 3.19: Infix Definition Translation

$$\mathcal{T}_{decl}\left[\!\!\left[\begin{array}{l} assoc\ \ prec\ \ op \\ \vdots \\ op\ \texttt{::}\ \ t \end{array}\right]\!\!\right] s \quad = \quad \texttt{(}op\texttt{)}\ \ assoc\ \ prec\ \texttt{::}\ \ t$$

$$\mathcal{T}_{decl}\left[\!\!\left[\begin{array}{l} assoc\ \ prec\ \ op \\ \vdots \\ \texttt{data}\ t\ \texttt{=}\ cs_0\ \texttt{|}\ t_0\ \ op\ \ t_1\ \texttt{|}\ cs_1 \end{array}\right]\!\!\right] s \quad = \quad \begin{array}{l} \texttt{::}\ \ t\ \texttt{=}\ cs_0 \\ \qquad \texttt{|}\ \texttt{(}op\texttt{)}\ \ assoc\ \ prec\ \ t_0\ \ t_1 \\ \qquad \texttt{|}\ cs_1 \end{array}$$

Where the fixity declaration could also be above the operator definition or data declaration too. *assoc* is one of `infixr`, `infixl` or `infix` and *prec* is an integer from `0` to `9`. In addition, if the fixity is not specified then it should default to `infixl 9`.

Figure 3.20: Fixity Declaration Translation

```
(++) infixr 5 :: [a] [a] -> [a]
```

- In Clean, a data constructor operator must have its fixity inlined at the point where it is defined in the type declaration. For example:

```
:: List a = Nil | (:) infix 5 a (List a)
```

- In Haskell, a fixity declaration for an operator is optional (if one is not given then `infixl 9` is assumed).

The translations for each of these issues are formalised in Figure 3.20.

3.6 Translating the Module System

In this section we raise the issues regarding the translation of Haskell's module system. The aim of the module system in both languages is essentially the same: to help structured programming by providing a means by which common functions and data types can be placed together, exported to other modules and imported by other modules thereby allowing well-defined abstract interfaces to user-defined types. However despite this common aim, there are several features specific to each language. Although we raise the issues and suggest possible solutions, these solutions are not finalised nor formal (of course we still believe that this is necessary, it is simply the case that we don't have time to do it!).

3.6.1 Exporting Names

Names in Haskell can be explicitly exported from a module by listing them after the module name declaration, e.g.

```
module Example (id) where
  id x = x
  fix f = f (fix f)
```

Here, the module `Example` only exports the name `id` (and not `fix`). In contrast, Clean modules are composed of separate files: a declaration module (the types of all the names to be exported) and an implementation module (the actual definitions of these names). So, the the translation of the above Haskell code might be:

Definition File (*.dcl*)	Implementation File (*.icl*)

`definition module Example`
`id :: a -> a`
	`implementation module Example`
	`id :: a -> a`
	`id x = x`
	`fix f = f (fix f)`

The general approach to the translation is to place the types of all names that the Haskell module exports into the definition module (and the implementation module can remain untouched except for syntactic differences). If the Haskell module doesn't list the names to export explicitly, then all names within the module are implicitly exported (but *not* those names which are imported from other modules).

One particular issue to notice is that the type of the function `id` needs to be specified in both Clean files, whereas it is never even mentioned in the original Haskell module. This is because all exported names

must have explicit type-signatures in Clean. So, unless type-signatures are enforced in the Haskell source (which is considered good practice), we will require a type-checker in the translator to perform the above translation.

Of course, there are five other entities besides functions which can be exported (classes, methods, types, constructors and modules). This is illustrated in the following example (we omit irrelevant details using three dots, i.e. `...`):

Haskell'98	Clean 2.1 (almost equivalent)

`module Example (`
`  module Monad,`
`  -- Classes and methods`
`  Eq((==)), Ord(..),`
`  -- Types and constructors`
`  List(Cons, Nil), BTree(..)`
`) where ...`
	`definition module Example`
	`import Monad`
	`class Eq a where ...`
	`class Ord a where ...`
	`:: List a = ...`
	`:: BTree a = ...`

Notice that the Haskell module exports an entire module (the `Monad` module) by putting in its export list, whereas the Clean definition module does so by importing it.

However, the names that have been exported by each language differ in this example because the Haskell module only exports the `==` method of the `Eq` class and the Clean module exports both methods (i.e. `==`, and `/=`). This represents an incompatibility between the module systems, since Clean doesn't allow a subset of a data-type's constructors (or a class's methods) to be exported: either all must be exported or none. A reasonable solution to this issue will arise in the next section. Another important point is that the Clean definition module must specify any instance declarations which should be exported whereas in Haskell all instances are implicitly exported.

3.6.2 Importing Names

In the same way that a module can specify what names it wishes to export, it may also specify which names it wishes to import from other modules, e.g.

Haskell'98	Clean 2.1 (almost equivalent)

`import Prelude (`
`  map,`
`  Eq(..), Ord(compare),`
`  Maybe(Nothing, Just)`
`)`
	`from Prelude import`
	`  map,`
	`  class Eq(..), class Ord(compare),`
	`  :: Maybe(Nothing, Just),`
	`  instance Eq Int`

So, although Clean can't export a subset of a data-type's constructors (or a class's methods) it can import a subset. This means that the issue in the previous section can be translated by modifying all imports of the module `Example` so as to only import the subset of constructors or methods that have been exported by the Haskell module.

The only difference between the two code fragments above is that the Clean code has to explicitly specify that it wishes to import the `Int` instance of `Eq`, whereas in Haskell all instances in the Prelude will

be implicitly imported. So, the translator must remember to explicitly import and export all instance declarations of classes that have been imported or exported.

Sometimes we wish to import a large list of names and omit just a few. To avoid having to type out such a list, Haskell provides a `hiding` clause which may be used as follows:

```
import Prelude hiding (map)
```

Here, all the names exported by the Prelude are imported except for `map`. Since Clean has no support for this, the translator will have generate the list of all names exported by the Prelude apart from `map` so that an equivalent `import` statement can be produced.

3.6.3 Qualified Imports

Despite fine-grained control over imports and exports, Haskell programs still suffer from namespace conflicts all too often. To combat this, Haskell provides what it calls "qualified names" which are names which combine the module name with the name of the entity that has been imported. For example, when the `Example` module is imported not only is the name `id` added to the namespace, but so too is the qualified name `Example.id`. If we only want the latter name to be added to the namespace then we can write:

```
import qualified Example
```

Furthermore if we want a short-hand notation for referring to `Example.id` such as `E.id` then we can write:

```
import qualified Example as E
```

The issue here is that Clean doesn't support qualified names. Recall from the review our summary of Hegedus's proposal for resolving this issue: Rename qualified names such as `A.x` to `A_x`. Then, for each module in the program, a new module should be created that renames all the exported functions to their qualified equivalents (using the underscore notation). When a module is imported, its corresponding module of qualified names should also be imported.

We illustrate this proposal on the `Example` module. First of all we create a new module which we call `Example_` defined as follows:

Definition File (*.dcl*)	Implementation File (*.icl*)

`definition module Example_`
`  Example_id :: a -> a`
	`implementation module Example_`
	`  import Example`
	`  Example_id :: a -> a`
	`  Example_id x = id x`
	`  Example_fix f = fix f`

Then, whenever `Example` is imported then `Example_` should also be imported. However, if `Example` is imported as qualified only then only `Example_` should be imported. If `Example` "as" `E` (i.e. as above) then another module should be created in the same style as `Example_` called `E_`, and it should be imported.

Chapter 4

Implementation

We now turn our attention to the implementation of the translation functions ($\mathcal{T}_\star$) that were formalised in our design. Perhaps the most important step in this process is to choose a programming language that enables us to specify the translation using a notation that is as similar as possible to $\mathcal{T}_\star$, thereby increasing our confidence that the implementation conforms to it. Indeed, due to the formality of $\mathcal{T}_\star$, perhaps even a provably correct implementation is not out of the question.

Recall from the review that program translation falls within the more general area of program transformation in which there are two programming languages of particular interest: Haskell and Txl. Choosing one language over the other is, at the moment, not a very easy task, yet is of great importance to the success of the project. So, in the next section we provide a in-depth comparison of the two, after which we will be in a more suitable position to commence the implementation of our translator.

4.1 Specifying Transformations: Comparing Haskell and Txl

The purpose of this section is to compare the two languages, Haskell and Txl, for specifying transformations in general. In order to demonstrate features of each language we will show how a small number of Haskell'98-like expressions (which we will refer to as *hs* expressions) can be transformed to the Haskell'98 kernel.

4.1.1 Parsing and Pretty-printing

Txl allows the format of the input and output to be specified together as a single, potentially ambiguous, context-free grammar in a BNF-like format. As an example, consider the following Txl definition of *hs* expressions. Note that the specification contains pretty-printing annotations that inform Txl to display the *if* expression across multiple lines with indentation:

```
define exp
    if [exp] then       [NL][IN]
      [exp]             [NL][EX]
    else                [NL][IN]
      [exp]             [EX]
  | '[ [list exp] ']    % A list
```

```
    | [number]              % A numeric literal
    | ...                   % Other types of expression
  end define
```

Indeed, this piece of code defines both the parser and the pretty-printer for *hs* expressions. The pretty-printing annotations [NL], [IN] and [EX] stand for new line, indent and exdent respectively. By default Txl will attempt to keep output within 80 characters of width by indenting progressively by 2 spaces for each line that is wrapped. Furthermore, it will provide sensible spacing between terminals and non-terminals in the grammar. However, if required, raw output can be enabled so that the exact output format can be specified by the programmer (further formatting annotations include [TAB] and [SP] for tab and space).

The sample definition also illustrates some predefined non-terminals; [number] simply represents a positive numeric literal and [list exp] represents a comma separated list of exp's. Non-terminals such as list, repeat (repetition – not comma separated), and opt (optionality) serve well as frequently used abstractions. Txl includes a wide range of other predefined non-terminals. In addition, terminals may be expressed using regular expressions.

Under Haskell the parser and pretty-printer for *hs* expressions would be specified separately using parser and pretty-printing combinators respectively. First we will consider the definition of the pretty printer for *if* expressions using Wadler's prettier printer combinators:

```
pretty (IfExp b e1 e2) =
  text "if" <+> pretty b <+> text "then" <>
  nest 4 (line <> pretty e1) </>
  text "else" <>
  nest 4 (line <> pretty e2) <> line
```

It is immediately noticeable that this pretty-printer is slightly more verbose than the one defined in Txl. There are two main reasons for this: first that terminals must be explicitly stated using the text combinator, and secondly that sequential composition must also be made explicit using the <>, <+> (combine with space between) and </> (combine with a newline between) combinators.

Now we consider a parser for list expressions using Bird's monadic parser combinators:

```
listExp = do
  token (char '[')
  es <- manywith (token (char ',')) exp
  token (char ']')
  return (ListExp es)
```

The parser specification is again slightly more verbose than that in Txl. The token combinator in the parser allows whitespace to be accepted on either side of the given parser (in this case a character). The combination of token and char is likely to be quite common and so a new combinator ch can be defined that incorporates both:

```
ch = token . char
```

The manywith combinator serves a similar role to the list non-terminal under Txl, except that manywith is more general since it allows the use of any list separator you like (not just comma-separated). A new combinator can be introduced to mimic the Txl list non-terminal:

```
list = manywith (ch ',')
```

Now the original definition can be made considerably more compact:

```
listExp = do {ch '['; es <- list exp; ch ']'; return (ListExp es)}
```

In the Txl specification an expression can take one of several forms (list, if, or number) each of which is separated by a `|`. Likewise, a choice combinator is present in Bird's library named `orelse`, and so an *hs* expression can be defined as follows:

```
exp = ifExp `orelse` listExp `orelse` natExp `orelse` ...
```

Any parsing ambiguity between types of expression will be resolved by the order of the above `orelse` sequence. It is worth stating that in some cases, for the sake of efficiency, the parser needs to be re-factored to remove left-recursion or to avoid repeating unnecessary work when the beginning of two parsers are similar. Under Txl these conditions are handled automatically and so needn't be considered.

4.1.2 Translating Lists

Now, we are finally in a position to turn our attention to the topic of transformations. We will first consider the translation of *hs* lists to the Haskell'98 kernel as defined in the report:

$$[e_1,...,e_k] \quad = \quad e_1 : (e_2 : (... : (e_k : [\,])))$$

The following Txl rule results quite naturally from this equality:

```
rule trList
  replace [exp]
    '[ H [exp], T [list exp] ']
  by
    ( H : '[ T '] )
end rule
```

This rule states that all syntactic constructs of type `exp` which are of the form `[H, T]`, where `H` is single expression and `T` is a comma separated list of expressions, should be replaced with the construct `(H:[T])`. Rules in Txl will be applied repeatedly to any matching construct in the syntax tree, no matter how deep, until there are no occurrences of the match left in the entire tree. One restriction in a replacement rule is that the new construct must be of the same type as construct that is being replaced (the *homomorphic* restriction).

In Haskell, the translation has a similar structure:

```
trList (ListExp ns) =
  foldr (\x y -> ParenExp (InfixExp x ':' y)) (ListExp []) ns
```

This definition is perhaps not quite as clear as the Txl one since the abstract syntax data type is a bit more long-winded than simply using the syntactic forms directly. Nevertheless, the definition is short and concise.

The `trList` function is only defined here on expressions of the form `ListExp`. So, if there happens to be a list expression deep within an *if* expression, for example, then this function cannot be applied directly. Instead the function will need to be defined over all forms of expression and recursively call itself on all the sub-expressions of each. This can be a nuisance if you want to define only a few transformations over a large abstract syntax and so Txl has a slight advantage here.

However, this problem is widely known within the Haskell community. The extra code required traverse the entire expression tree is commonly referred to as "boilerplate" code, i.e. code which has to be written but requires no special knowledge to write it. Peyton Jones provides a solution to this problem which he refers to as "scrap your boilerplate"[Pey03]. The solution requires an extension to Haskell'98 and has been implemented in version 6 of GHC. Before we can use `trList` without boilerplate, we must extend the function so that if it isn't applied to a `ListExp` then it will just act like the identity function:

```
trList (ListExp ns) = ...
trList e = e
```

The following example shows how, using the extension, the `trList` function can be used transform *all* list expressions within an arbitrarily large *hs*-expression:

```
trAllLists :: Exp -> Exp
trAllLists = everywhere (mkT trList)
```

Here, the `mkT` function can be read as "make a transformation". The result of applying it to `trList` yields a new function that acts as follows: if applied to a value of type `Exp`, it behaves like the `trList` function, otherwise it behaves like the identity function. Now, using `everywhere`, the function returned by `mkT` can applied to *every* node in the `Exp` tree, thereby yielding an expression with all its `ListExps` translated to the format specified by the Haskell'98 kernel.

4.1.3 Translating Sections

Sections in Haskell'98 provide a convenient syntax for partial application of binary operators and have a straightforward translation to the Haskell'98 kernel:

$$(op\ e) \ = \ \lambda x \rightarrow x\ op\ e$$

However, this equality only holds for an identifier x that does not occur within the expression e. Fortunately Txl has a predefined function `[!]` such that when applied to an identifier will return that identifier appended with a suffix that makes it unique over all identifiers in the input. This can be used so to create an identifier that doesn't occur within e:

```
rule trLeftSection
  replace [exp]
    (Op [op] E [exp])
  construct Id [id]
    x
  construct UniqId [id]
    Id [!]
  by
    (\ UniqId -> UniqId Op E)
end rule
```

To create a unique identifier, the rule says to construct an identifier `x` called `Id` and then construct a unique identifier like `Id` called `UniqId`. There is no obvious equivalent to this feature under Haskell without first traversing the input to retrieve all identifiers.

4.1.4 Translating "do" expressions

Do notation was introduced in Haskell'98 as syntactic sugar for specifying monadic expressions. The following equality shows how one kind of monadic expression can be translated to the kernel:

$$do\ \{p \leftarrow e;\ stmts\} \quad = \quad e \gg\!= \lambda p \rightarrow do\ \{stmts\}$$

The corresponding Txl translation rule is remarkably easy to specify, and really highlights the elegance of Txl:

```
rule trDo3
  replace [exp]
    do {P [pat] <- E [exp]; S [repeat stmt]}
  by
    E >>= \P -> do {S}
end rule
```

Again the Haskell translation is a bit more verbose, but is still short and concise (note that this definition uses an actual abstract syntax for full Haskell'98):

```
trDo (HsGenerator loc p e : stmts) =
  HsInfixApp e (op ">>=") (HsLambda loc [p] (trDo stmts))
```

Before the days of the *do* notation programmers would use the direct lambda expression style and lay it out nicely across multiple lines, for example:

```
putStr s          >>
getLine           >>= \l ->
return (words l)
```

This raises an interesting issue regarding pretty-printing: how can the expressions be laid out so that when a `>>` or `>>= \e ->` is encountered then a newline is inserted and the next line be correctly indented? Txl's pretty-printing annotations are too restrictive for such a complex layout rule, whereas this could certainly be programmed under Haskell.

4.1.5 Symbol table construction

A transformation rule will often require information that is not immediately available in the sentence that is to be transformed. For example, when transforming Haskell patterns it may be necessary to know if a constructor within a pattern is a *newtype* constructor or not. Such information is usually stored in a symbol table.

To construct a symbol table under Txl, the following approach seems appropriate:

```
rule main
  replace [program]
    P [program]
  construct Symtab [symtab]
    P [getSymTab]
  by
    P [trProgram Symtab]
end rule
```

It says that a program should be replaced by first constructing a symbol table from the program, and then applying the `trProgram` rule with the symbol table as a parameter. However, the above Txl extract is incorrect: it breaks the homomorphic restriction since, in order to construct the symbol table of type `symtab`, we need to transform a program of type `program`.

In order to overcome this restriction the `symtab` type could be made an alternative of the `program` type, which seems quite unnatural as the parser would then parse symbol tables in the input sentences! To avoid this problem Txl allows the grammar to be redefined so that the output may contain constructs that are not allowed to occur in the input. However, it still seems unnatural to define a symbol table inside the grammar of the input/output language.

One possible solution would be to use a global variable to store the symbol table which can be read using the `import` construct and written to using the `export` construct. However, the sudden introduction of mutable state into Txl seems quite unsatisfying: a very powerful transformation system with a neat and simple semantics is apparently, somewhat rashly (I believe), amended to support the complicated concept of state.

In Haskell, this homomorphic restriction does not apply to functions and so an input program can easily be mapped to a symbol table which could then be passed as a parameter to the translation function.

4.1.6 Summary

Txl allows parsers and pretty-printers to be specified by a single BNF-like grammar definition. Pretty-printing annotations allow layout of the output to be specified conveniently with a minimal syntax. Although these annotations can be quite restrictive, they do allow for the most common forms of pretty-printing.

Txl has a host of predefined non-terminals which allow the vast majority of programming language constructs to be easily specified. Since terminals may be specified using regular expressions, Txl provides the combined power of lexical and syntactic specifications. Furthermore, Txl will handle left-recursive parsers automatically and is known to produce efficient parsers.

Txl uses a "transform by example" paradigm where the programmer can specify rules that replace a certain input construct with a given output construct. A rule can then be applied using a variety of "search and replace" strategies (for example, replace the first occurrence or replace all occurrences). Since program constructs can be written in their syntactic forms, rather than some verbose abstract syntax data type, the resulting rules are very compact and elegant.

However, due to the homomorphic restriction, Txl apparently struggles to specify computations where the input type and the output type differ. A good example of this is when trying to create the symbol table of a program.

In contrast to Txl's unified parser/pretty-printer specifications, the Haskell approach is to define them separately making use of the appropriate combinator libraries (or Domain Specific Languages). We note that these libraries have been thoroughly researched and developed, and allow the rapid and coherent development of parsers and pretty-printers.

Haskell's support for symbolic pattern matching makes it very well suited to specifying syntax-directed translations, although since the source and target sentences are specified using an abstract syntax, definitions can appear more cluttered than their Txl counterparts. We note that while Haskell requires

unnecessary verbosity (boilerplate code) to specify traversals over large syntax descriptions, this problem has been quite nicely overcome in recent extensions to GHC Haskell.

4.2 Specifying Transformations: Our Choice

It would now appear that, due to the discovery of Peyton Jones' "scrap your boilerplate" work (see section 4.1.2), we not only have two possible implementation languages, but three: standard Haskell, GHC Haskell (with Generics), and Txl. We make our choice according to the following criteria:

- The ease of parsing Haskell'98
- The ease of pretty printing Clean 2.1
- The notational similarity with the definition of $\mathcal{T}_\star$

If a full Haskell'98 parser is to be written from scratch then Txl would perhaps stand strongest since it has the simplest parser specification notation, allowing the programmer to disregard issues such as efficiency and left-recursive grammars. However this advantage is countered by the fact that several well-tested Haskell'98 parsers (written in Haskell) are available in a variety forms (e.g. as a GHC library, or in the nhc98 compiler).

The difficulty in writing a full pretty-printer for the Clean language depends really on how pretty we want the output. However, we notice that many of the translations defined by $\mathcal{T}_\star$ produce valid Haskell'98 code as well as valid Clean code. Furthermore, much of the Clean-only output from $\mathcal{T}_\star$ does not differ greatly from a similar construct in Haskell. This suggests that perhaps a Clean pretty-printer can be heavily based on a Haskell'98 pretty-printer. Since there is a widely-used pretty-printer for Haskell'98 (written in Haskell'98), the choice of Haskell would again seem more favourable.

In our third and final criterion, Txl and Haskell with Generics have a considerable advantage over standard Haskell'98. This is because $\mathcal{T}_\star$ is defined "by example" i.e. we only define translations on small number of constructs and not on the entire syntax tree (as would have to be done in standard Haskell'98). Likewise, translations under Txl and Haskell with Generics are also specified "by example" and so we needn't even mention those constructs which are equivalent between the languages (just as with $\mathcal{T}_\star$).

Based on these arguments it would seem that GHC Haskell with Generics is the most favourable solution. However this approach was not taken due to its rather late discovery. In fact, the time constraints on the project would not permit any approach other than overlapped research and implementation phases.

So we choose standard Haskell'98 as our implementation language.

4.3 A Hybrid Haskell and Clean Syntax

Perhaps the most obvious approach to develop a translator is to consider it as three separate sub-tasks: a Haskell'98 parser, a translator from Haskell'98 syntax to Clean 2.1 syntax, and a Clean 2.1 pretty printer. However, the design shows us that actually many of the translations produce not only valid Clean code, but also valid Haskell code. In addition, the languages look remarkably similar with often only minor syntactic differences apparent. Based on these observations it would seem overkill to

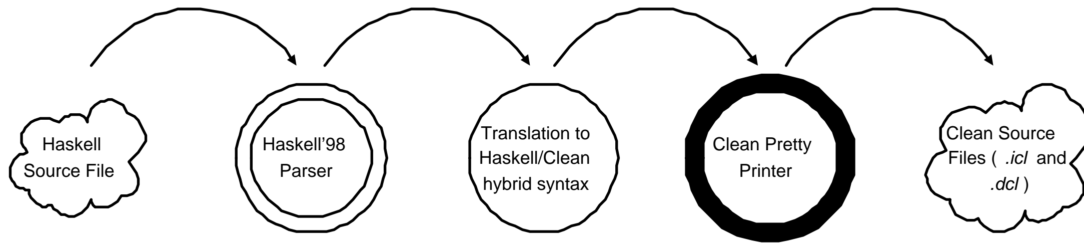

Figure 4.1: Data flow diagram for the translator.

develop a Clean pretty-printer from scratch (when one could be based on fully working Haskell pretty-printer) and unnecessary to translate from the Haskell syntax data-type to a completely different Clean syntax data-type (when many of the constructs are so similar and don't even need to be translated).

Instead, we translate Haskell syntax to a hybrid Haskell/Clean syntax, i.e. the Haskell syntax with a few extensions required for Clean-specific constructs. This means that we don't need to translate equivalent pieces of Haskell and Clean syntax. In addition, the Haskell'98 pretty-printer need only be modified to print the Haskell syntax slightly differently (for the minor syntactic differences) and augmented to support the new Clean-specific constructs. A data flow diagram illustrating the operation of the translator is shown in Figure 4.1. Our implementation uses the Haskell'98 parser and pretty-printer in GHC's hierarchical library[GHC].

In fact, even by simply parsing and pretty-printing Haskell'98 (not even modified for Clean), and performing no translation whatsoever, several issues between Haskell and Clean are resolved, for example string gaps and numeric literals:

Original Program	After Parsing and Pretty-Printing

`str = "hello \`
`      \world"`
`x = 1.1e-1`
	`str = "hello world"`
	`x = 0.11`

We made three extensions to the Haskell syntax to allow for Clean-specific constructs, each of which were required to overcome issues shown below:

Haskell Syntax	Clean Syntax

`infixr 5 ++`
`(++) :: [a] -> [a] -> [a]`
`infixr 5 :+`
`data List a = Nil | a :+ (List a)`
	`(++) infixr 5 :: [a] [a] -> [a]`
	`:: List a = Nil`
	`          | (:+) infixr 5 a (List a)`

Firstly, since Clean type signatures can include fixity information directly (such as that for ++ above), a new constructor `ClTypeSig` was added to the `HsDecl` type. Likewise, since Clean constructor declarations can include fixity information (such as that for `:+` above), a new constructor `ClConDecl` was added to the `HsConDecl` type. Finally, since Clean allows both curried and uncurried type-signatures (illustrated by ++ above) a new constructor `ClUncurriedTyFun` was added to the `HsType` type. Further extensions may be required in future to support other non-Haskell constructs (e.g. Clean's `derive` clause). The remaining syntactic differences were simply resolved by adjusting the Haskell pretty-printer.

4.4 A Framework for Translation

Recall for our design that the type of our translation function for expressions was:

$$\mathcal{T}_{exp} : \text{HsExp} \rightarrow \text{Symtab} \rightarrow \text{ClExp}$$

Indeed the types of the other translation functions (e.g. for declarations and patterns) were quite similar. However, since we never formalised our translations for the module system issues, we never needed to mention that some of the translation functions not only return a piece of Clean code but actually two pieces of Clean code (one each for the implementation and definition modules). So, perhaps a more suitable type for the declaration translation function $\mathcal{T}_{decl}$ is:

$$\mathcal{T}_{decl} : \text{HsDecl} \rightarrow \text{Symtab} \rightarrow (\text{ClImpDecl} \times \text{ClDefDecl})$$

In practice it is also useful if our translation functions are able to return a textual report detailing any problems encountered, for example the renaming of an identifier (perhaps because it is Clean keyword) or that a particular issue resolution has not been implemented yet (such as the deriving clause). So we again adjust the type of $\mathcal{T}_{decl}$ (and the other translation functions):

$$\mathcal{T}_{decl} : \text{HsDecl} \rightarrow \text{Symtab} \rightarrow (\text{ClImpDecl} \times \text{ClDefDecl} \times \text{Report})$$

It appears that the types of the translation functions have now become quite obscure due to their increased detail. Likewise, the definitions of the translation functions will become more complex as they have to return even more information. This is especially annoying since most of the time a translation function will not need to return code for the definition module nor return a textual report and probably not even need to use the symbol table. It would be much nicer to only have to mention these optional inputs and outputs as and when we need to. Indeed, this problem is not specific to our translator, but a well-known problem of purely functional programming languages. Wadler describes this problem very nicely:

> "Pure functional languages have this advantage: all flow of data is made explicit. And this disadvantage: sometimes it is painfully explicit." – Wadler[Wad93]

Wadler goes on to show how monads can be used to overcome this problem. A monad is an abstract data type (i.e. a data type whose representation need not be known in order to use it) with a set of functions defined on it (called monadic functions). Because the representation is abstracted away from, new functionality can be added to the data type without affecting already-working code. A monad also allows information to be passed around implicitly: instead of passing around lots of different values, they can all be stored in a single monad which can be passed around instead. Information can be added or extracted (using monadic functions) to and from the monad as and when required.

A widely-used general monad is the state monad which stores a given data type as a state. By providing a sequential composition monadic function, computations may proceed as normal but have side-affects on the internal state of the monad and so the order in which computations are performed becomes significant.

We define our state to be a triple containing a Haskell symbol table, a textual report and a list of Haskell/Clean hybrid declarations (representing the Clean definition or *.dcl* file):

```
newtype HacleState = HacleState (HsSymtab, Report, [HsDecl])
```

Our state monad named `Hacle` is defined identically to a typical state monad (whose state is `HacleState`) and so is not given here. We define a host of monadic functions on `Hacle` including functions for adding and removing items from the symbol table, adding text to the report, and adding definitions to the *.dcl* file. The monadic function `report` takes a string and adds returns nothing, but has the *side-effect* of adding the given string to the translation report:

```
report :: String -> Hacle ()
```

The `symtabAddSym` function takes a symbol and the information we wish to store about this symbol and returns nothing, but has the *side-effect* of adding the information to the symbol table:

```
symtabAddSym :: HsSym -> HsSymInfo -> Hacle ()
```

The `symtabUnusedVarSym` function takes a string and returns a string like the one that was given, except it will be appended with a number so that is unique compared to all the symbols in the symbol table:

```
symtabUnusedVarSym :: String -> Hacle String
```

These are just a select few of our monadic functions. We now illustrate our monad in action in part of the definition for the translation of Haskell expressions:

```
trExp :: HsExp -> Hacle HsExp
trExp (HsNegApp e) =
  do report "Translating unary minus to ‘negate"
     e’ <- trExp e
     return (HsParen (HsApp (var "‘negate") (HsParen e’)))
```

The function `trExp` when given a Haskell expression will return a hybrid Haskell/Clean expression *and may also have some side-effect on the state of the translator*. In this example, the side-effect is that the textual report gets amended.

4.5 Summary of Implementation

It is easier and more useful to state what translations were not implemented, rather than an large list of all those that were:

- Arbitrary precision integers – see section 3.4.1
- Automatic instance derivation – see section 3.4.7.
- Module system issues – see section 3.6.
- The Haskell Prelude.

The translator also has one other major limitation: it can only translate single-module programs. So, if symbol table information regarding entities defined in another module is required for a translation, then the translator will not work correctly. Since no module system resolutions were implemented, we saw no point in providing multi-module operation at this stage.

Our translator is not yet at a suitable stage to attempt to translate the Haskell Prelude, and so we incorporate Simon's “Prelude for Clean” that we discussed in the review (section 2.3). This Prelude, although not ideal because it is not compliant with some of the translations we make (e.g. default

class definitions), is better than no Prelude at all. As of yet, we don't attempt to translate Clean's `Start` function to Haskell's `main` function, and instead will have to perform this step manually in any translations we perform.

Some of our translations require some auxiliary definitions, for example, the definition of the list constructor macro `@` (see section 3.2.2). We define these auxiliary definitions in a single module `Hacle` that should be imported by all modules which are translated. Included in this module are definitions for:

- The `@` macro for list construction.
- The `~` function for functional composition since Haskell's fullstop symbol is not allowed in Clean. `~` is an operator which is allowed in Clean but is reserved in Haskell, so it serves as a suitable replacement for fullstop.
- The uniquely named pointers to the corresponding Prelude functions for:
  - `‘negate` points to `negate`
  - `‘i` points to `fromInteger`
  - `‘enumFrom`... points to `enumFrom`...
- `CharList` is defined to be a type synonym for `[Char]`

The final artifact is a command line program, written in Haskell'98 and compatible with both Hugs and GHC, that takes a Haskell'98 file as input and produces, to standard output, a textual report of the translation and a Clean program.

Chapter 5

Evaluation

In this chapter we will evaluate our translator using two different approaches. Firstly, by comparing it over a range of programs (each having a particular issue) with the Haskell Frontend for Clean, thereby providing insight into the completeness of each system. And secondly, by running it on a range of "real" Haskell programs and observing not only if the resulting programs run correctly, but how much faster or slower they run, thereby providing insight into the strengths and weaknesses of the translator in general and the motivation for building it.

5.1 A Comparison of the Translator with the Frontend

Recall from the review that Divianszky and Hegedus built a "Haskell Frontend for Clean". In this section we will evaluate this system alongside our own using a range of specially devised test programs. Each test program contains a particular issue that was identified in our design. We run each program through the Frontend and compare its output with that returned by the Hugs interpreter. We also translate each program (using our translator) and compile the resulting program using the Clean 2.1 compiler. The output from this executable is also compared with that from the original Hugs-interpreted program. The Frontend and our translator are said to "pass" a test-program if, after compilation, it returns the same output as the original Hugs-interpreted program.

42 tests in total were carried out and are shown in Appendix B. Here, we will refer to these tests as T1 to T42. 6 of these tests are brought forward for observation in Figures 5.1 to 5.6. Table 5.1 summarises the number of tests passed by each system.

Table 5.1 shows that our translator passes 14 more tests than the Frontend does. However, this result *should not* be taken to mean that our translator performs "better" than the Frontend. This is because the Frontend solves two of the more difficult issues, i.e. automatic class derivation (see Figure 5.4) and to provide a accurate Haskell Prelude, whose solutions have not yet been implemented in our translator. However, one of the more difficult issues that we have implemented, and that the Frontend has not, is that of resolving field-labelled data-types and their associated pattern-matching, construction and updating (see Figure 5.5). A more appropriate conclusion to take from this result is simply that our translator is aware of more issues in Haskell to Clean translation. For example, it seems that Divianszky and Hegedus are unaware of issues such as those with pattern-bindings (see T10, T13, T14, and T42), newtype pattern-matching (see T22), operators in patterns (see T16 and T17), and fixity definitions and constructor operators (see T30, T31, and T32) amongst others.

System	Test Programs Passed
The Haskell Frontend for Clean	23/42
The Translator (Hacle)	37/42

Table 5.1: Comparing the Translator and the Frontend

The small amount of documentation that is available for the Frontend states that irrefutable patterns in the source program will be ignored with a warning. It would therefore seem that Divianszky and Hegedus know that it is an issue, but have not derived a solution for it (see Figure 5.1).

We also notice that although successor patterns are allowed by the Frontend, they have not been resolved correctly (see Figure 5.2). The problem is that the Frontend matches `2` against `n+3`, which is invalid under the semantics of Haskell'98.

5.2 Translating "Real" Programs

In this section we evaluate our translator by translating "real" programs, i.e. programs that actually do something useful (in contrast to our test programs discussed in the previous section). The purpose of this is to help determine if the translator is of any practical use to the Haskell and Clean communities.

We translate 5 programs, of increasing size, in total. The first program solves the shortest superstring problem, which when given k strings, will return the shortest string that contains all k strings as substrings (our solution uses brute-force). The second program solves the cryptarithmetic problem, which when given a input such as `SEND + MORE = MONEY` returns all the assignments of digits to letters so that equality holds (of course, the first letter in each word cannot be assigned the digit `0`). The third and fourth programs make use of a sorting library from the nofib benchmarking suite[1] [Par92] of Haskell programs. All of these Haskell programs were easily translated to Clean using our translator (where the `main` function was translated by hand to an appropriate `Start` function and any non-library functions used were manually inserted).

The fifth program solves the "mate-in-n" problem in Chess, which when given an arbitrary Chess position, whose turn it is, and the number of moves that can be used to force checkmate, will return the sequence(s) of moves that can be used to do it. However, there were several (reasonably simple) manual steps required in the translation of this program:

- Merged all modules into one.
- Translated the `main` function to an appropriate `Start` function by hand, and hard-coded in Shinkman's problem instance shown in Figure 5.7 (instead of having to read it from a file).
- Removed all "deriving" clauses from the source by using the automatic instance derivation tool DrIFT.
- Added non-library functions `sortBy` and `insertBy`.
- Disabled "default class methods" translation in our translator (due to incompatible Prelude).

[1] Which does include the `nfib` program!

Test 11 – Irrefutable Patterns Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
t11 :: [Int]
t11 = dup []
  where dup xxs@(~(x:xs))
          | xxs == [] = []
          | otherwise = x:xxs
```

Hacle Output: Hacle Result: `[]`

```
t11 :: [Int]
t11 = dup []
  where dup xxs=:(irr0)
          | xxs == [] = let (x @ xs) = irr0 in []
          | otherwise = let (x @ xs) = irr0 in x @ xxs
```

Figure 5.1: Test: Irrefutable Patterns

Test 12 – Successor Patterns Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: `-1`

```
t12 :: Int
t12 = f 2
  where f (n + 3) = n
```

Hacle Output: Hacle Result: $\perp$

```
t12 :: Int
t12 = f (‘i 2)
  where f (n_succ0) | (n_succ0 >= 3) = let n = n_succ0 - 3 in n
```

Figure 5.2: Test: Successor Patterns

Test 22 – Newtype Pattern Matching Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: $\perp$

```
newtype Natural = N Int
t22 :: Int
t22 = f undefined
  where f (N i) = 1
```

Hacle Output: Hacle Result: `1`

```
:: Natural = N !Int
t22 :: Int
t22 = f undefined
  where f (irr0) = let (N i) = irr0 in (‘i 1)
```

Figure 5.3: Test: Newtype Pattern Matching

Test 26 – Automatic Instance Derivation Hacle: *Fail*, Frontend: *Pass*

Haskell Code: Frontend Result: `[A, B, C]`

```
data Alpha = A | B | C | D deriving Enum
t26 :: [Alpha]
t26 = [A .. C]
```

Hacle Output: Hacle Result: Issue not resolved

Figure 5.4: Test: Automatic Instance Derivation

Test 27 – Field Labelling Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
data T = T {a :: Int}
t27 :: T
t27 = T {a = 1}
```

Hacle Output: Hacle Result: `T 1`

```
:: T = T Int
a x = case x of T y -> y
t27 :: T
t27 = T (‘i 1)
```

Figure 5.5: Test: Field Labelling

Test 42 – Pattern-Binding Type-Signatures Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
t42 :: Int
t42 = x
  where x :: Int
        xs :: [Int]
        (x:xs) = [1, 2, 3]
```

Hacle Output: Hacle Result: `1`

```
t42 :: Int
t42 = x
  where rhs0 = [(‘i 1), (‘i 2), (‘i 3)]
        xs :: [Int]
        xs = case rhs0 of x @ xs -> xs
        x :: Int
        x = case rhs0 of x @ xs -> x
```

Figure 5.6: Test: Pattern-Binding Type-Signatures

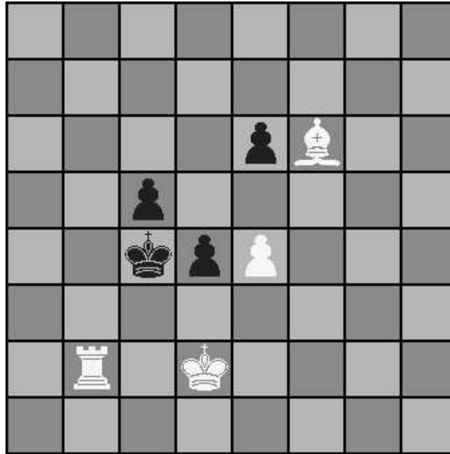

Figure 5.7: Shinkman's White to move and mate-in-4 Problem (solution: `if` 1. Rb1 d3 `then` 2. Ba1 e5, 3. Rb2 Kd4, 4. Rb4# `else` 1. Rb1 e5 `then` 2. Bd8 d3, 3. Bb6 Kd4, 4. Rb4#)

Program	**Problem Instance**	**GHC 5.04.2**	**Clean 2.1**	**Speedup**
Shortest Super-String	9 strings	3.25s	1.01s	3.2
Cryptarithmetic	`WIND` + `RAIN` = `STORM`	1.14s	3.48s	0.3
Heap sort	10^7 integers	4.7s	3.3s	1.4
Merge sort	10^7 integers	4.71s	3.26s	1.4
Mate-in-n	Shinkman's Problem	3.54s	0.83s	4.2

Table 5.2: Timings of a range of Haskell programs (compiled with GHC) compared with those of the translated programs (compiled with Clean 2.1)

Each translated program then compiled and executed to give the same answer as the original Haskell program. The times taken for each program to execute (after being compiled with both GHC 5.04.2 and Clean 2.1) is shown in Table 5.2. Interestingly in 4 of the 5 programs the translated and Clean-compiled program runs 1.4 to 4.2 times faster than the GHC-compiled program. One possible explanation for this is that the Prelude for Clean that we are using defines many of its functions to be strict (and in addition, the Clean compiler searches the program for functions whose arguments can be safely made strict). It is well-known that strictness can have either a positive or a negative effect on efficiency depending on how it is used. For example, Ackerman's function can be much more efficiently calculated if its arguments are evaluated strictly since no extra space is required to store the "unevaluated" expressions. In contrast, the expression `f c` where `f x = 1` and `c` is some massively complex expression is much more efficient when non-strictly evaluated.

5.3 The Strengths and Weaknesses of our Work

Perhaps the greatest weakness of our translator is that it doesn't implement solutions to a few of the more common, and perhaps more difficult, issues. Instead, it implements a large number of "rare" issues. By a "rare" issue, we mean a Haskell construct that is not commonly used by Haskell programmers, e.g. newtypes, irrefutable patterns, pattern bindings, fixity declarations and class defaults .

In hindsight, it may have been more valuable to implement our solutions to the more commonly used constructs such as automatic instance derivation, module system constructs and translating the Prelude, which are used by *almost all* Haskell programs.

However, by using a systematic approach, we have discovered several new issues in Haskell to Clean translation that have not been previously documented. In addition, we have raised and documented a large number of issues, and have formally specified many of our solutions, which is something that previous work has perhaps failed to do (there is little to no documentation about the translations used in the Haskell Frontend for Clean). Our systematic approach to finding issues has proved worthwhile by the 14 extra test-programs passed by our translator but not by the Frontend.

Chapter 6

Further Work and Conclusions

6.1 Further Work

In our design, time didn't permit the formalisation of all our proposed solutions, in particular the module system issues and automatic instance derivation. As a result our translator is by no means finished, as was evident from its evaluation when translating the mate-in-n program. The following tasks remain to be completed:

- Implementation of arbitrary precision integers in Clean – see section 3.4.1.
- Provide more formal solutions to the module-system issues and implement them – see section 3.6.
- Extend the translator to add symbols of imported modules to its symbol table, so that it can resolve issues that extend across modules such as default class methods.
- Translate the primitive parts of the Prelude to Clean. The rest of the Prelude should be translated by our translator.
- Translate other standard Haskell modules in the same way.
- Formalise and implement solutions to the automatic instance derivation issue – see section 3.4.7.

In the design we proposed that Clean's generics could be used to resolve automatic instance derivation issue, although this was never formalised nor implemented. In the evaluation we preprocessed the Haskell mate-in-n program by passing it through the DrIFT tool so to resolve this issue. Due to the success of DrIFT and the ease in which it was used, we now suggest, in hindsight, that this tool could be used in a preprocessing stage of the translator. To illustrate this we show how the first step of this preprocessor might work:

Original Source	After first step of preprocessing
`data Colour = Black | White` `  deriving (Eq, Ord)`	`data Colour = Black | White` `{-! for Colour derive: Eq, Ord !-}`

The output from this step is then suitable for processing by DrIFT which would generate the necessary instance declarations for `Eq` and `Ord` automatically. The resulting code would then be suitable for our

current translator.

There are two other problems that might be suitable for resolution in the preprocessor: firstly, the Clean module system requires the type-signatures for all exported functions, and secondly, overloaded numberic literals will cause a type-error in Clean, and so a type-signature must be given to resolve the ambiguity. If all top-level functions were augmented with type signatures, for instance by passing the source through the Hugs interpreter's ":browse" command, then this would completely resolve the former issue, and go some way to resolving the latter issue in a large number of cases (*all* functions, not just top-level ones, would need augmented to resolve this issue fully)

6.2 Conclusions

In this project we have raised, documented and solved a large number of issues in Haskell to Clean translation, many of which were implemented in our translator "Hacle". Many of these issues have not been raised publicly before, and several were previously unsolved. This was evident in the comparison of Hacle with the Haskell Frontend for Clean, in which Hacle was shown to resolve 37 out of 42 known issues and the Frontend 23 out of 42.

In translating 5 "real" Haskell programs to Clean using Hacle, we found that in 2 of them the resulting Clean-compiled program ran more than 3 times faster than the original GHC-compiled program, and in 2 others, 1.4 times faster. This suggests that not only might the Haskell community be able to use a new Haskell compiler, but that this compiler can potentially generate better code for certain types of program than existing Haskell compilers.

However, there remains much work to be done, as documented in section 6.1, before our translator is complete, and so claim to provide a new fully-compliant Haskell'98 compiler. Although some issues will be more difficult to solve than others, we see no reason why this wouldn't be possible with a reasonable amount of effort.

Of course, the Clean community can also benefit from our work. Once the translator is complete, a range of widely-used Haskell'98 libraries can be ported to Clean and so used also by Clean developers.

One particular way that we were able to reduce the complexity of our translator was to use a hybrid syntax that is able to represent both the input and output languages of the translation. Without this approach a considerable amount of time would have been wasted in translating constructs which didn't require translation and in developing two abstract syntax data types when one was sufficient.

Hopefully we have provided useful reading material to anyone who is interested in the development of programming language translators in the wider context of computer science. In particular, in the area of specifying translations, our comparison of Haskell and Txl for parsing, pretty-printing and program translation might explain some ideas which were found useful in this project: one being that the value of specifying translations concisely and elegantly should not be underestimated. This helps us to convince ourselves that our translator conforms to its specification.

Striving for simplicity in the face of complexity is a difficult but necessary challenge: Haskell, Haskell with Generics, Txl, and a hybrid syntax were all of interest in the simplification of the complex translator developed here.

Bibliography

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, January 1998.

[Chi02] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for Tracing. International Workshop on the Implementation of Functional Languages, September 2002. URL: `http://www.cs.kent.ac.uk/people/staff/oc/`.

[Cor03] James R. Cordy, Ian H. Carmichael, and Russell Halliday. *The TXL Programming Language Version 10.3*, March 2003.

[EWD69] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[Div03] Peter Divianszky and Hajnalka Hegedus. Haskell - Clean Compiler, 2003. URL: `http://aszt.inf.elte.hu/~fun_ver/2003/software/HsCleanAll2.0.2.zip`.

[Gil95] Andy Gill and Simon Marlow. Happy: the parser generator for Haskell. January 1995. URL: `http://www.haskell.org/happy/`.

[Heg01b] Hajnalka Hegedus. Transforming Haskell Structures to Clean. Technical report. URL: `http://aszt.inf.elte.hu/~grid/dg/dg-rep-1-arch-h2clean.ps`.

[Heg01a] Hajnalka Hegedus. Haskell to Clean Front End. Master's thesis, ELTE, Budapest, Hungary, 2001.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

[Hugs98] Mark P. Jones and John Peterson. Hugs 98. URL: `http://www.haskell.org/hugs/`.

[Pey97] Simon Peyton Jones. Haskell pretty-printer library, 1997. URL: `http://research.microsoft.com/~simonpj/downloads/pretty-printer/pretty.html`.

[Hs98] Simon Peyton Jones and John Hughes et al. *Haskell 98: a non-strict, purely functional language*, 1999. URL: `http://www.haskell.org/definition/`.

[GHC] Simon Peyton Jones and Simon Marlow et al. The Glasgow Haskell Compiler (GHC). URL: `http://www.haskell.org/ghc/`.

[Pey03] Simon Peyton Jones and Ralf Lammel. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37. ACM Press, 2003.

[Mar02] Simon Marlow. Haddock, A Haskell Documentation Tool. URL: `http://www.haskell.org/haddock/`.

[Par92] Will Partain, Simon Peyton Jones, and Simon Marlow. The NoFib benchmark suite. URL: `http://www.haskell.org/ghc/nofib.html`.

[Cl02] M. J. Plasmeijer and M.C.J.D. van Eekelen. *Language Report of Concurrent Clean version 2.1*, November 2002. URL: `http://www.cs.kun.nl/~clean/`.

[Pla93] Rinus Plasmeijer and Marko Van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[nhc98] Niklas Rojemo and Malcolm Wallace et al. The nhc98 Compiler. URL: `http://www.haskell.org/nhc98/`.

[Sim03] Janos Gyorgy Simon. *Prelude for Clean*. ERASMUS of the European Community and the Hungarian National Science Research Grant (Project number: OTKA T037742), 2003. URL: `http://aszt.inf.elte.hu/~fun_ver/2003/software/Prelude.pdf`.

[Tur85] D. A. Turner. Miranda: A non-strict language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, 1985.

[Wad93] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

Appendix A

Lexical Issues in Haskell to Clean Translation

Character Set Haskell-98 allows the use of the Unicode character set to encode the program, whereas Clean only supports ASCII. However, few Haskll-98 compilers support Unicode input.

General Indentifiers General identifiers in Haskell and Clean are almost identical. There are only two differences:

- The prime character in Haskell is represented using an apostrophe whereas it is represented using a backtick in Clean.
- Under no circumstances can a Clean identifier begin with an underscore (except when it appears on its own: see section 1.4), whereas a Haskell identifier may begin with one. There is a convection specified in the report that variables begining with an underscore should be unused variables.

Namespaces Both Haskell and Clean have restrictions on what kind of identifiers begin with an upper case letter and a lower case letter. Those kinds which can begin with an upper case letter are marked UC, those which can begin with lower case letter are marked LC. Also, those that can be written using symbols (i.e. not letters) are marked S. Some symbol identifiers must begin with a colon (marked SC), and some must not begin with a colon (marked SNC).

Haskell essentially has the following seperate namespaces:

- Function names (LC, SNC), variable names (LC, SNC), and field names (LC, SNC)
- Constructor names (UC, SC)
- Type variable names (LC)
- Type constructor names (UC) and type class names (UC)
- Module names (UC)

Clean essentially has the following seperate namespaces:

- Function names (LC, UC, S), constructor names (UC, S), variables (LC)
- Field names (LC)
- Type constructor names (UC, S)
- Type variable names (LC)
- Type classes (LC, UC, S)
- Module names (LC, UC, S)

From the above tables, a variable name can take the form SNC, however in Clean it may only take the form LC. This means symbol-style variable names will have to be translated appropriately.

In the remaining cases each Haskell namespace and naming restriction is a subspace of the corresponding Clean namespace and naming restriction.

Reserved Identifiers, Keywords and Symbols Both Haskell and Clean make use of the reserved identifier _ to represent anonymous variables in pattern matching. I am unaware of any differences in the use of _ between the two semantics.

Clean 2.1 keywords that are not reserved in Haskell are:

```
definition system implementation from with dynamic special export CONS
FIELD derive code
```

In Haskell and Clean, function and constructor names can comprise of the following symbols:

```
! # $ % & * + / < = > ? @ \ ^ | - ~ :
```

The `.` symbol is the only symbol allowed in Haskell but not in Clean.

The following operators cannot be used in Clean for function or constructor names, but can be used in Haskell.

```
/* */ // :== =: \\ <-: . ! & # #!
```

Comments Block and line comments, which may be nested to an aribtrary depth, are present in both Haskell and Clean.

Qualified names Haskell allows use of the use of qualified names to explicitly refer to different namespaces. I am unaware of an equivalent feature in Clean. This problem will be appropriately dealt with in the modules section.

Literals Integer literals between the two languages are similar, except for the following differences:

- Octal literals in Haskell start `0o` or `0O` e.g. `10` is represented as `0o12`. In Clean they start `0` e.g. `10` is represented as `012`.
- Hex literals in Haskell may start `0x` or `0X`, however in Clean they may only start `0x`.

Real-valued literals are also very similar, apart from the following difference:

- In both languages, real-valued literals may be specified in exponential form e.g. `1.1e-2`. In Haskell an upper and lower case `e` is allowed, whereas it must be lower case in Clean.

String and character literals are similar, apart from the following differences:

- Clean is missing `'\a'` (abort), `'\v'` (vertical tab) and `'\&'` (null character) special escape codes.
- Haskell allows string gaps where a string can be written over multiple lines. Clean doesn't appear to support an equivalent feature.
- Haskell allows ASCII control characters to be written using a set of three letter mnemonics or using the hat character (e.g. `∧X`), whereas Clean doesn't support either.

Layout Rule There are a few differences in the layout rules between Haskell and Clean:

- Tabs represent 4 spaces in Clean and 8 spaces in Haskell.
- Haskell supports mixed layout where layout mode can be both on and off in different parts of the same source file. Clean only supports one or the other for the entire source file (i.e. no mixing allowed).

Other issues not covered in the Design Some examples are given below of som syntactic issues that were not mentioned explicity in the main report (but some examples did illustrate them without explicitly commenting on it):

Haskell'98	Clean 2.1
`if e0 then e1 else e2`	`if e0 e1 e2`
`f :: (Eq a, Eq b) => a -> b -> Int`	`f :: a b -> Int | Eq a & Eq b`

Further notes The Haskell-98 report states that real-valued literals may take the form `1e-1` instead of the form `1.0e-1`. However, neither Hugs or GHC support this. This tends to suggest a mistake in the Haskell-98 report.

I am convinced there is a mistake in the Clean 2.1 report, which by my understanding, states that `'?'` is not a valid character literal (amongst many others).

Appendix B

Comparing Hacle with the Frontend

Test 1 – Unary Minus Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `11`

```
t1 :: Int
t1 = abs (-11)
  where abs x | x >= 0 = x
              | x < 0 = -x
```

Hacle Output: Hacle Result: `11`

```
t1 :: Int
t1 = abs ((‘negate ((‘i 11))))
  where abs x
          | x >= (‘i 0) = x
          | x < (‘i 0) = (‘negate (x))
```

Test 2 – List Syntax Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `[1, 2, 3, 4]`

```
t2 :: [Int]
t2 = id 1 : [2, 3, 4]
```

Hacle Output: Hacle Result: `[1, 2, 3, 4]`

```
t2 :: [Int]
t2 = id (‘i 1) @ [(‘i 2), (‘i 3), (‘i 4)]
```

Test 3 – Sections Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `(0.2, 2)`

```
t3 :: (Float, Float)
t3 = (reciprocal 5, half 4)
  where reciprocal = (1 /)
        half = (/ 2)
```

Hacle Output: Hacle Result: `(0.2, 2)`

```
t3 :: (Real, Real)
t3 = (reciprocal (‘i 5), half (‘i 4))
  where reciprocal = (((/)) (‘i 1))
        half = (((\ f x y -> f y x) (/)) (‘i 2))
```

Test 4 – Do Notation Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: 30

```
data M a = M a
instance Monad M where
  return x = M x
  m >> k = m >>= \ _ -> k
  m >>= k = ...
  fail s = undefined
t4 :: Int
t4 = x
  where (M x) = do x <- return 10
                   y <- return 20
                   return (x + y)
```

Hacle Output: Hacle Result: 30

```
// A small number of changes were made to the instance declaration to
// due incompatibility with the Prelude we use.

:: M a = M a
instance Monad M where
        return x = M x
        (>>=) m k = ...
        fail s = undefined
t4 :: Int
t4 = x
  where rhs0 = let ok x
                 = let ok y = return (x + y)
                       ok _ = undefined
                     in return (`i 20) >>= ok
               ok _ = undefined
             in return (`i 10) >>= ok
        x = case rhs0 of M x -> x
```

Test 5 – Arithmetic Sequences Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `[Mon, Tue]`

```
data Weekday = Mon | Tue | Wed | Thu | Fri
instance Enum Weekday where
    enumFrom a = [Mon, Tue]
    ...
t5 :: [Weekday]
t5 = [Mon ..]
```

Hacle Output: Hacle Result: Couldn't test due to Prelude

```
:: Weekday = Mon | Tue | Wed | Thu | Fri
instance Enum Weekday where
        enumFrom a = [Mon, Tue]
        ...
t5 :: [Weekday]
t5 = (‘enumFrom (Mon))
```

Test 6 – Arithmetic Sequences Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `[(1,1),(1,2),(2,1),(2,2)]`

```
t6 :: [(Int, Int)]
t6 = cross [1, 2] [1, 2]
  where cross xs ys = [(x, y) | let xs2 = xs, x <- xs2, y <- ys]
```

Hacle Output: Hacle Result: `[(1,1),(1,2),(2,1),(2,2)]`

```
t6 :: [(Int, Int)]
t6 = cross [(‘i 1), (‘i 2)] [(‘i 1), (‘i 2)]
  where cross xs ys
          = let xs2 = xs in
              let ok x
                    = let ok y = [(x, y)]
                          ok _ = []
                        in concatMap ok ys
                  ok _ = []
                in concatMap ok xs2
```

Test 7 – Expression Type-Signatures Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `10`

```
t7 :: Int
t7 = (fromInt v) :: Int
  where v = 10
```

Hacle Output: Hacle Result: `10`

```
t7 :: Int
t7 = let ‘ :: Int
         ‘ = (fromInt v)
       in ‘
  where v = (‘i 10)
```

Test 8 – Backticked Notation Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `15`

```
t8 :: Int
t8 = 2 ‘power‘ 2 ‘power‘ 2 - 1
  where power :: Int -> Int -> Int
        power x 0 = 1
        power x n = x * power x (n - 1)
        infixr 9 ‘power‘
```

Hacle Output: Hacle Result: `15`

```
t8 :: Int
t8 = (‘i 2) ‘power‘ (‘i 2) ‘power‘ (‘i 2) - (‘i 1)
  where power :: Int Int -> Int
        power x 0 = (‘i 1)
        power x n = x * power x (n - (‘i 1))
        (‘power‘) infixr 9
        (‘power‘) = power
```

Test 9 – General As-Patterns Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `[1,1,2,3]`

```
t9 :: [Int]
t9 = dup [1, 2, 3]
  where dup xxs@(x:xs) = x:xxs
```

Hacle Output: Hacle Result: `[1,1,2,3]`

```
t9 :: [Int]
t9 = dup [(‘i 1), (‘i 2), (‘i 3)]
  where dup xxs=:(x @ xs) = x @ xxs
```

Test 10 – As-Patterns in Pattern-Bindings Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `[1,1,2,3]`

```
t10 :: [Int]
t10 = x : xxs
  where xxs@(x:xs) = [1, 2, 3]
```

Hacle Output: Hacle Result: `[1,1,2,3]`

```
t10 :: [Int]
t10 = x @ xxs
  where rhs0 = [(‘i 1), (‘i 2), (‘i 3)]
        xxs = case rhs0 of
                (x @ xs) -> xxs
        xs = case rhs0 of
                (x @ xs) -> xs
        x = case rhs0 of
                (x @ xs) -> x
        xxs = rhs0
```

Test 11 – Irrefutable Patterns Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
t11 :: [Int]
t11 = dup []
  where dup xxs@(~(x:xs))
          | xxs == [] = []
          | otherwise = x:xxs
```

Hacle Output: Hacle Result: `[]`

```
t11 :: [Int]
t11 = dup []
  where dup xxs=:(irr0)
          | xxs == [] = let (x @ xs) = irr0 in []
          | otherwise = let (x @ xs) = irr0 in x @ xxs
```

Test 12 – Successor Patterns Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: `-1`

```
t12 :: Int
t12 = f 2
  where f (n + 3) = n
```

Hacle Output: Hacle Result: $\perp$

```
t12 :: Int
t12 = f (‘i 2)
  where f (n_succ0) | (n_succ0 >= 3) = let n = n_succ0 - 3 in n
```

Test 13 – Top-Level Pattern-Bindings Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
(p1, p2, p3) = (1, 2, 3)
t13 :: Int
t13 = p1
```

Hacle Output: Hacle Result: `1`

```
rhs0 :: (Int, Int, Int) // Type given due to internal overloading
rhs0 = ((‘i 1), (‘i 2), (‘i 3))
p1 = case rhs0 of (p1, p2, p3) -> p1
p2 = case rhs0 of (p1, p2, p3) -> p2
p3 = case rhs0 of (p1, p2, p3) -> p3
t13 :: Int
t13 = p1
```

Test 14 – 0-Arity Constructors in Pat-Bindings Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Runtime Error

```
t14 :: Int
t14 = x
  where [x] = tail [1, 2]
        (1:y:ys) = [1, 2, 3]
```

Hacle Output: Hacle Result: 2

```
t14 :: Int
t14 = x
  where rhs0 = tail [(‘i 1), (‘i 2)]
        x = case rhs0 of [x] -> x
        rhs1 = [(‘i 1), (‘i 2), (‘i 3)]
        ys = case rhs1 of 1 @ y @ ys -> ys
        y = case rhs1 of 1 @ y @ ys -> y
```

Test 15 – Guarded Pattern-Bindings Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: 1

```
t15 :: Int
t15 = x
  where (x:xs) | True = [1, 2, 3]
               | otherwise = y
          where y = [4, 5, 6]
```

Hacle Output: Hacle Result: 1

```
t15 :: Int
t15 = x
  where rhs0
          | True = [(‘i 1), (‘i 2), (‘i 3)]
          | otherwise = y
          where y = [(‘i 4), (‘i 5), (‘i 6)]
        xs = case rhs0 of x @ xs -> xs
```

Test 16 – Prefixed Operators in Patterns Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
t16 :: Int
t16 = head [1, 2, 3]
  where head ((:) x xs) = x
```

Hacle Output: Hacle Result: 1

```
t16 :: Int
t16 = head [(‘i 1), (‘i 2), (‘i 3)]
  where head (x @ xs) = x
```

Test 17 – Backticked Constructors in Patterns Hacle: *Fail*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
data MyList a = MyNil | MyCons a (MyList a)
t17 :: Int
t17 = head (MyCons 1 (MyCons 2 MyNil))
  where head (x ‘MyCons‘ xs) = x
```

Hacle Output: Hacle Result: Implementation forgotten!

```
:: MyList a = MyNil | MyCons a (MyList a)
t17 :: Int
t17 = head (MyCons (‘i 1) (MyCons (‘i 2) MyNil))
  where head (x ‘MyCons‘ xs) = x
```

Test 18 – Arbitrary Precision Integers Hacle: *Fail*, Frontend: *Fail*

Haskell Code: Frontend Result: 2080661269

```
t18 :: Integer
t18 = 123456789123456789123456789
```

Hacle Output: Hacle Result: Not implemented.

```
t18 :: Int
t18 = (‘i 123456789123456789123456789)
```

Test 19 – Curried Type-Signatures Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: (1,2)

```
t19 :: (Int, Int)
t19 = mkPair 1 2
  where mkPair :: a -> b -> (a, b)
        mkPair x = \y -> (x, y)
```

Hacle Output: Hacle Result: (1,2)

```
t19 :: (Int, Int)
t19 = mkPair (‘i 1) (‘i 2)
  where mkPair :: a -> b -> (a, b)
        mkPair x = \ y -> (x, y)
```

Test 20 – Curried Type-Signatures and Classes Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `(1,2)`

```
class C a where
  foo :: a -> b -> (a, b)
instance C Int where
  foo x y = (x, y)
t20 :: (Int, Int)
t20 = foo 1 2
```

Hacle Output: Hacle Result: `(1,2)`

```
class C a where
      foo :: (a -> b -> (a, b))
instance C Int where
      foo = ‘ where ‘ x y = (x, y)
t20 :: (Int, Int)
t20 = foo (‘i 1) (‘i 2)
```

Test 21 – Curried Type-Signatures Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `MyCons 2 MyNil`

```
data MyList a = MyNil | MyCons a (MyList a)
t21 :: MyList Int
t21 = myMap (+1) (MyCons 1 MyNil)
  where myMap :: (a -> b) -> MyList a -> MyList b
        myMap f MyNil = MyNil
        myMap f (MyCons x xs) = MyCons (f x) (myMap f xs)
```

Hacle Output: Hacle Result: `MyCons 2 MyNil`

```
:: MyList a = MyNil | MyCons a (MyList a)
t21 :: MyList Int
t21 = myMap (((\ f x y -> f y x) (+)) (‘i 1)) (MyCons (‘i 1) MyNil)
  where myMap :: (a -> b) (MyList a) -> MyList b
        myMap f (MyNil) = MyNil
        myMap f (MyCons x xs) = MyCons (f x) (myMap f xs)
```

Test 22 – Newtype Pattern Matching Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: $\perp$

```
newtype Natural = N Int
t22 :: Int
t22 = f undefined
  where f (N i) = 1
```

Hacle Output: Hacle Result: `1`

```
:: Natural = N !Int
t22 :: Int
t22 = f undefined
  where f (irr0) = let (N i) = irr0 in (‘i 1)
```

Test 23 – Multi-Variable Type-Signatures Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: 1

```
  t23 :: Int
  t23 = x
    where x, y :: Int
          x = 1
          y = 2
```

Hacle Output: Hacle Result: 1

```
t23 :: Int
t23 = x
  where x :: Int
        x = (‘i 1)
        y :: Int
        y = (‘i 2)
```

Test 24 – Defualt Class Methods Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: 1

```
  class MyClass a where
    myFun1 :: a -> a
    myFun1 x = x
    myFun2 :: a -> a -> (a, a)
  instance MyClass Int where
    myFun2 x y = (x, y)
  t24 :: Int
  t24 = myFun1 1
```

Hacle Output: Hacle Result: 1

```
  class MyClass a where
        myFun1 :: (a -> a)
        myFun2 :: (a -> a -> (a, a))
  instance MyClass Int where
        myFun2 = ‘ where ‘ x y = (x, y)
        myFun1 = ‘ where ‘ x = x
  t24 :: Int
  t24 = myFun1 (‘i 1)
```

Test 25 – Overloaded Literals Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: 2.3

```
  t25 :: Float
  t25 = x + y
    where x = 1
          y = 1.3
```

Hacle Output: Hacle Result: 2.3

```
  t25 :: Real
  t25 = x + y
    where x = (‘i 1)
          y = 1.3
```

Test 26 – Automatic Instance Derivation Hacle: *Fail*, Frontend: *Pass*

Haskell Code: Frontend Result: `[A, B, C]`

```
data Alpha = A | B | C | D deriving Enum
t26 :: [Alpha]
t26 = [A .. C]
```

Hacle Output: Hacle Result: Issue not resolved

Test 27 – Field Labelling Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
data T = T {a :: Int}
t27 :: T
t27 = T {a = 1}
```

Hacle Output: Hacle Result: `T 1`

```
:: T = T Int
a x = case x of T y -> y
t27 :: T
t27 = T (‘i 1)
```

Test 28 – Empty Classes Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
class (Eq a, Enum a) => EqEnum a
instance EqEnum Int
```

Hacle Output: Hacle Result: Instance declaration removed

```
class (Eq a, Enum a) => EqEnum a
```

Test 29 – Order of Declarations Hacle: *Pass*, Frontend: *Pass*

Haskell Code: Frontend Result: `10`

```
t29 = 10
helloWorld = "hello world"
t29 :: Int
```

Hacle Output: Hacle Result: `10`

```
t29 :: Int
t29 = (‘i 10)
helloWorld = [’hello world’]
```

Test 30 – Infix Defintions Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
infixl 9 +*
(+*) :: Int -> Int -> Int
a +* b = a
t30 :: Int
t30 = 1 +* 2
```

Hacle Output: Hacle Result: 1

```
(+*) infixl 9 :: Int Int -> Int
(+*) a b = a
t30 :: Int
t30 = (‘i 1) +* (‘i 2)
```

Test 31 – Infix Defintions Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
infixr 9 +-
(f +- g) x = f (g x)
t31 = ((\x -> x + 1) +- (\x -> x * 2)) 2
```

Hacle Output: Hacle Result: 5

```
(+-) infixr 9
(+-) f g x = f (g x)
t31 = ((\ x -> x + (‘i 1)) +- (\ x -> x * (‘i 2))) (‘i 2)
```

Test 32 – Constructor Operator Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
infixr 5 :+
data MyList2 a = MyNil2 | a :+ (MyList2 a)
t32 :: MyList2 Int
t32 = 1 :+ 2 :+ MyNil2
```

Hacle Output: Hacle Result: `(:+ 1 (:+ 2 MyNil2))`

```
:: MyList2 a = MyNil2 | (:+) infixr 5 a MyList2 a)
t32 :: MyList2 Int
t32 = (‘i 1) :+ (‘i 2) :+ MyNil2
```

Tests 33-41 – Syntactic Issues Hacle: *Passed 7/9* Frontend: *Passed 4/9*

Haskell Code:

```
f' :: Int   -- t33: Frontend passed
f' = 1
_id :: Int  -- t34: Frontend passed
_id = 1
t35 :: Int
t35 = (*)   -- Frontend failed
  where (*) = 1
with :: Int -- t36: Frontend passed
with = 1
t37 :: (Int, Int, Int) -- Frontend failed
t37 = (0x11, 0X11, 0x10)
t38 :: Float -- Frontend failed
t38 = 1.1e-1
t39 :: (Int, Int) -- Frontend failed
t39 = (ord '\v', ord '\a')
t40 :: String
t40 = "hello \ -- Frontend passed
      \world"
t41 :: Int -- Frontend failed
t41 = ord '\NUL'
```

Hacle Output:

```
f` :: Int  // Hacle passed
f` = (`i 1)
u_id :: Int // Hacle passed
u_id = (`i 1)
t35 :: Int
t35 = (*) // Hacle passed
where (*) infixl 9
      (*) = (`i 1)
with_ :: Int // Hacle passed
with_ = (`i 1)
t37 :: (Int, Int, Int) // Hacle passed
t37 = ((`i 17), (`i 17), (`i 16))
t38 :: Real // Hacle passed
t38 = 0.11
t39 :: (Int, Int) // Hacle failed
t39 = (ord '\v', ord '\a')
t40 :: CharList // Hacle passed
t40 = ['hello world'
t41 :: Int // Failed
t41 = ord '\NUL' // Hacle failed
```

Test 42 – Pattern-Binding Type-Signatures Hacle: *Pass*, Frontend: *Fail*

Haskell Code: Frontend Result: Compile Error

```
t42 :: Int
t42 = x
  where x :: Int
        xs :: [Int]
        (x:xs) = [1, 2, 3]
```

Hacle Output: Hacle Result: 1

```
t42 :: Int
t42 = x
  where rhs0 = [(‘i 1), (‘i 2), (‘i 3)]
        xs :: [Int]
        xs = case rhs0 of x @ xs -> xs
        x :: Int
       x = case rhs0 of x @ xs -> x
```