The Clean programming language

Group 25
, Jingui Li
, Daren Tuzi

The Clean programming language

Overview

The Clean programming language first appeared in 1987 and is still being further developed. It was first designed by a research group from Radboud University Nijmegen, the Netherlands. Clean is dual licenced, it is available under the terms of the GNU LGPL and also under a proprietary license.

Clean is a general purpose functional programming language, it is widely used in scientific research all over the world. Featuring lazy evaluation, unique type inferencing system, and graph rewriting semantics, Clean is useful for writing relatively bug-free programs in fewer lines of code. Clean is said to have some advantages over similar functional programming languages. For example, efficient running-time of compiled code. It is appreciably faster than Haskell for many applications. Also, Clean comes with easy optimization of code. It is easier to optimize than Haskell.

Features

Before we talk about individually the features of Clean, we first take a look at an example in Clean to get a conceptual understanding of the language.

1. *fac :: Int -> Int // int as input and int as output*
2. *fac 0 = 1*
3. *fac n = n * fac (n-1)*
4. *Start :: Int*
5. *Start = fac 6*

The first line is a function declaration which says the function takes an integer as parameter and return an integer. Line 2 and 3 are corresponding rules of the

function. From line 1 to 3, those are lines which define the function called *fac*. Each program in Clean computes the value of the expression Start. In Start, we evaluate *fac* 6. Since there is definition for the function, the value will be 6*5…*1. Now we have a rough idea on how function definition and evaluation works in Clean, we continue talking about features of Clean.

Strong typing

Clean is a strongly typed language based on an extension of the well-known Milner / Hindley / Mycroft type inferencing/checking scheme, it includes common higher order types, polymorphic types, abstract types, algebraic types, synonym types and existentially quantified types. Due to it's strong typing and the obligation to initialize all objects being created, run-time error can only occur in a very limited number of cases.

Predefined and user defined types

Clean offers various of predefined types: integers, reals, Booleans, characters, strings, lists, tuples, records, arrays and files.
These types are commonly seen in many modern programming languages, but it has it's unique types. In Clean, type 'Real' is used for 64 bit double precision floating point values. We list some an example in Clean to see some peculiarities of the language.

1. *[1,3..9]* *// [1,3,5,7,9]*
2. *[1..9]* *// [1,2,3,4,5,6,7,8,9]*
3. *[1,2,3,4,5]* *// [1,2,3,4,5 and so on…*

As we can see from the above examples, the syntax of using list is quite straightforward. In Clean, there is 'dot-dot' ('..') syntax facilitating the definition of successive values.

A tuple is an algebraic data type predefined for reasons of programming convenience and efficiency. Tuples have as advantage that they allow bundling a finite number of objects of arbitrary type into a new object without being forced to

define a new algebraic type for such a new object. This is in particular handy for functions that return several values.
In Clean, programmer can define algebraic types, record types and synonyms.

Modularity and sophisticated I/O

Module structure is used to control the scope of definitions. The basic idea is that definitions only have a meaning in the implementation module they are defined in unless they are exported by the corresponding definition module. Having the exported definitions collected in a separate definition module has as advantage that one in addition obtains a self-contained interface document one can reach out to others. The definition module is a document that defines which functions and data types can be used by others without revealing uninteresting implementation details. Furthermore, the module structure enables separate compilation that heavily reduces compilation time. If the definition module stays the same, a change in an implementation module only will cause the recompilation of that implementation module. When the definition module is changed as well, only those implementation modules that are affected by this change need to be recompiled.

Clean also offers sophisticated I/O library with which GUI application can be built easily. The portability of Clean program makes it possible to use the same source code without any modification.

Laziness and high order function

As is common for many functional programming languages, Clean is a lazy programming language. Expression is evaluated only if it is actually used in program. There is also explicit graph rewriting semantics in Clean. As an aside, graph rewriting concerns with the technique of creating a new graph out of an original graph algorithmically. High order function does more than one thing, it can either take function as arguments or return function as result. Since functions

can be arguments and results of other functions in Clean, Clean programs are easy to understand, modify and reuse .

Classes and overloading

Clean has a very powerful notion of classes. It is important to understand that this concept of classes is different from the classes you might know from object oriented languages. In Clean a class is a family of functions with the same name. The difference between these family members is the type processed.
As a very simple example consider the class of increment functions.

```
1. class inc t :: t -> t
```

This says that the class inc has type variable t. There is only a single manipulation function in this class, which is also named inc. The type of this increment function is t -> t. Instances of this class for integers and reals are defined by:

```
1. instance inc Int where
2.    inc i = i+1
3.  instance inc Real where
4.   inc r = r+1.0
```

Even the record Item, an element of an order, can be incremented:

```
1. instance inc Item where
2. inc i = {i & quantity = inc i.quantity}
```

The increment of item i, is that item with the field quantity set to the increment of that field from the argument record.
Also basic operations like +, == and < are defined as a type class, in fact we have used this to define the functions square and product shown above. This implies that it is possible to define these basic operators for your own data types. For instance we can define the comparison of products (the record Product form above) as the comparison of their names:

```
1. instance < Product where
2. (<) p1 p2 = p1.name < p2.name
```

Using list comprehensions we can easily define the famous *quick sort* algorithm for lists of elements of the class <. The *quick sort* algorithm states that an empty list is always sorted. A non empty list is split in a fraction of elements smaller than the

first element and a fraction of elements larger than or equal to the first element. These sub-lists are sorted separately and the resulting lists are glued together by the

```
1.append operator ++:
2.qsort :: [t] -> [t] | < t
3.qsort []    = []
4.qsort [e:r] = qsort [ x \\ x <- r | x<e ]
5.            ++ [e] ++
6.            qsort [ x \\ x <- r | x>=e ]
```

The operator >= is derived from the operator <:

```
1. class >= a where
2.  (>=) infix 4 :: a a -> Bool | < a
3. (>=) x y :== not (x<y)
```

This implies that >= is defined for a class as soon as the operator < is defined. This function qsort is able to sort lists of integers as well as list of products, and any other member of the class <. To be member of the same class, data types need not have any relation. It is sufficient that the appropriate functions are defined for that data type. As you will understand now, the classes in Clean are more general than the notion of classes in most object oriented languages. It is easy to emulate the notion of classes from those object oriented languages using Clean's concept of classes.

Concluding remarks

Although Clean has so many advantages and nice features, it is not widely used in industry. Clean looks like Haskell but way easier than Haskell, you can learn all there is to Clean in a few hours. For a long time, Clean was closed-source, offering only commercial licenses. A few years ago, Clean was released as open source under LGPL. If you want to have some commercial application, you would not want to use Clean. Perhaps this is one of the reasons why Clean is not so popular.

References

[1]. Clean - The Functional Programming Language
http://compsci.ca/v3/viewtopic.php?t=15459
[2]. Clean
http://clean.cs.ru.nl/Clean
[3]. Functional programming in CLEAN
http://www.inf.ufsc.br/~jbosco/cleanBookI.pdf
[4]. Functional programming matters
http://fpmatters.blogspot.fi/2009/04/clean-programming-language-insanely.html