Clean for Haskell Programmers

Mart Lubbers
mart@cs.ru.nl

Peter Achten
peter@cs.ru.nl

October 4 2023

This note is meant to give people who are familiar with the functional programming language Haskell[1] a consise overview of Clean[2] language elements and how they differ from Haskell. Many of this is based on work by Achten although that was based on Clean 2.1 and Haskell98 [1]. Obviously, this summary is not exhaustive, a complete specification of the Clean language can be found in the latest language report [9]. The main goal is to support the reader when reading Clean code. Table 1 shows frequently occuring Clean language elements on the left side and their Haskell equivalent on the right side. Other Clean language constructs that also frequently occur in Clean programs, but that do not appear in the table are:

Modules Clean modules have separate definition (headers) and implementation files. The definition module contains the class definitions, instances, function types and type definitions (possibly abstract). Implementation modules contain the function implementations as well. This means that only what is defined in the definition module is exported in Clean. This differs greatly from Haskell, as there is only a module file there. Choosing what is exported in Haskell is done using the **module** `Mod(···)` syntax.

Strictness In Clean, by default, all expressions are evaluated lazily. Types can be annotated with a strictness attribute (`!`), resulting in the values being evaluated to head-normal form before the function is entered. In Haskell, in patterns, strictness can be enforced using `!`[3]. Within functions the strict let (`#!`) can be used to force evaluate an expression, in Haskell **seq** or `$!` is used for this.

Uniqueness typing Types in Clean may be *unique*, which means that they may not be shared [3]. The uniqueness type system allows the compiler to generate efficient code because unique data structures can be destructively updated. Furthermore, uniqueness typing serves as a model for side effects as well. Clean uses the *world-as-value* paradigm where `World` represents the external environment and is always unique. A program with side effects is characterised by a `Start :: *World → *World` start function. In Haskell, interaction with the world is done using the **IO** monad. The **IO** monad could very well be—and actually is—implemented in Clean using a state monad with the `World` as a state. Besides marking types as unique, it is also possible to mark them with uniqueness attributes variables `u:` and define constraints on them. For example, to make sure that an argument of a function is at least as unique as another argument. Finally, using `.` (a dot), it is possible to state that several variables are equally unique. Uniqueness is propagated automatically in function types but must be marked manually in data types. Examples can be seen in listing 1.

```
f :: (Int, *World) → *World // uniqueness is propagated automatically for function types (i.e. *(Int, *World)))
f :: *a → *a                // f works on unique values only
f :: .a → .a                // f works on unique and non-unique values
f :: v:a u:b → u:b, [v<=u]  // f works when a is less unique than b
```

Listing (Clean) 1: Examples of uniqueness annotations in Clean.

Generics Generic functions [8]—otherwise known as polytypic or kind-indexed fuctions—are built into Clean [9, Chp. 7.1][2] whereas in Haskell they are implemented as a library [10, Chp. 6.19.1]. The implementation of generics in Clean is very similar to that of Generic H∀skell [7]. Metadata about the types is available using the **of** syntax, giving the function access to metadata records. This abundance of metadata allows for very complex generic functions that near the expression level of template metaprogramming. Listing 4 shows an example of a generic equality and listing 5 of a generic print function utilising the metadata.

GADTs Generalised algebraic data types (GADTs) are enriched data types that allow the type instantiation of the constructor to be explicitly defined [5, 6]. While GADTs are not natively supported in Clean, they can be simulated using embedding-projection pairs or equivalence types [4, Sec. 2.2]. To illustrate this, listing 2 and 3 show an example GADT implemented in Clean and Haskell[4] respectively.

[1]By Haskell we mean GHC's Haskell
[2]By Clean we mean Clean 3.1's iTask compiler.
[3]Requires BangPatterns to be enabled.
[4]Requires GADTs to be enabled.

Table 1: Syntactical differences between Clean and Haskell.

<table>
<tr><th>Clean</th><th>Haskell</th></tr>
<tr><td colspan="2">Comments</td></tr>
<tr><td>// single line
/* multi line /* nested */ */</td><td>-- single line
{- multi line {- nested -} }</td></tr>
<tr><td colspan="2">Imports</td></tr>
<tr><td>**import** Mod$_0$
import Mod$_1$ ⇒ **qualified** f, :: t</td><td>**import** Mod$_0$ (f, t)
import qualified Mod$_1$ (f, t)
import Mod$_1$ **hiding** (f, t)</td></tr>
<tr><td colspan="2">Basic types</td></tr>
<tr><td>42 :: Int
True :: Bool
toInteger 42 :: Integer
38.0 :: Real
"Hello" +++ "World" :: String[5]
['Hello'] :: [Char]
?t
(?None, ?Just e)</td><td>42 :: Int
True :: Bool
42 :: Integer
38.0 :: Float -- or Double
"Hello" ++ "World" :: String[6]
"Hello" :: String
Maybe t
(Nothing, Just e)</td></tr>
<tr><td colspan="2">Type definitions</td></tr>
<tr><td>:: T a$_0$ ··· :== t
:: T a$_0$ ··· = C$_0$ f$_0$ f$_1$ ··· | C$_1$ f$_0$ f$_1$ ··· | ···
:: T a$_0$ ··· = { f$_0$ :: t$_0$, f$_1$ :: t$_1$, ··· }
:: T a$_0$ ··· =: t
:: T = E.t: Box t & C t</td><td>**type** T a$_0$ ··· = t
data T a$_0$ ··· = C$_0$ f$_0$ f$_1$ ··· | C$_1$ f$_0$ f$_1$ ··· | ···
data T a$_0$ ··· = T { f$_0$ :: t$_0$, f$_1$ :: t$_1$, ··· }
newtype T a$_0$ ··· = t
data T = **forall** t.C t ⇒ Box t[7]</td></tr>
<tr><td colspan="2">Function types</td></tr>
<tr><td>f$_0$ :: a$_0$ a$_1$ ··· → t | C$_0$ v$_0$ & C$_1$, C2 v$_1$
(+) **infixl** 6 :: Int Int → Int

qid :: (A.a: a → a) → (Bool, Int)
qid id = (id True, id 42)</td><td>f$_0$ :: (C$_0$ v$_0$, C$_1$ v$_1$, C2 v$_1$) ⇒ a$_0$ → a$_1$ ··· → t
infixl 6 +
(+) :: Int → Int → Int
qid[8] :: (**forall** a: a → a) → (Bool, Int)
qid id = (id True, id 42)</td></tr>
<tr><td colspan="2">Type classes</td></tr>
<tr><td>**class** f a :: t
class C a | C$_0$, ··· , Cn a
class C s ~m **where** ···
instance C t | C$_0$ t & C$_1$ t ··· **where** ···</td><td>**class** F a **where** f :: t
class (C$_0$ a, ···, Cn, a) ⇒ C a
class C s m | m → s **where** ···[9]
instance (C$_0$ a, C$_1$ a, ···) ⇒ C t **where** ···</td></tr>
<tr><td colspan="2">As pattern</td></tr>
<tr><td>x=:p[10]</td><td>x@p</td></tr>
<tr><td colspan="2">Lists</td></tr>
<tr><td>[1,2,3]
[x:xs]
[e \\ e<-xs | p e]
[l \\ l<-xs, r<-ys]
[(l, r) \\ l<-xs & r<-ys]</td><td>[1,2,3]
x:xs
[e | e←xs, p e]
[l | l←xs, r←ys]
[(l, r) | (l, r)←zip xs ys] or [(l, r) | l←xs | r←ys][11]</td></tr>
<tr><td colspan="2">Lambda expressions</td></tr>
<tr><td>\a$_0$ a$_1$ ···→e or \···.e or \···=e</td><td>\a$_0$ a$_1$ ···→e</td></tr>
<tr><td colspan="2">Case distinction</td></tr>
<tr><td>**if** p e$_0$ e$_1$
case e **of**
    p$_0$ → e$_0$ // or p$_0$ = e$_0$
    ···

f p$_0$ p$_1$ ···
    | c = t
    | **otherwise** = t // or = t</td><td>**if** p **then** e$_0$ **else** e$_1$
case e **of**
    p$_0$ → e$_0$
    ···

f p$_0$ p$_1$ ···
    | c = t
    | **otherwise** = t</td></tr>
</table>

[5] Strings are unboxed character arrays.

[6] Strings are lists of characters or overloaded if OverloadedStrings is enabled.

[7] Requires ExistentialQuantification to be enabled.

[8] Requires RankNTypes to be enabled.

[9] Requires MultiParamTypeClasses to be enabled.

[10] May also be used as a predicate, e.g. **if** (e$_0$ =: []) e$_1$ e2.

[11] Requires ParallelListComp to be enabled.

[12] Field selection from unique records.

<table>
<tr><th colspan="2">Records</th></tr>
<tr><td>:: R = { f :: t }
r = { f = e }
r.f
r!f[12]
{r & f = e }</td><td>data R = R { f :: t }
r = R {e}
f r
(\v→(f v, v)) r
r { f = e }</td></tr>
<tr><th colspan="2">Record patterns</th></tr>
<tr><td>:: R_0 = { f_0 :: R_1 }
:: R_1 = { f_1 :: t }
g { f_0 } = e f_0
g { f_0 = {f_1} } = e f_1</td><td>data R_0 = R_0 { f_0 :: R_1 }
data R_1 = R_1 { f_1 :: t }
g (R_0 {f_0=x}) = e x or g (R_0 {f_0}) = e f_0[13]
g (R_0 {f_0=R_1 {f_1=x}}) = e x</td></tr>
<tr><th colspan="2">Arrays</th></tr>
<tr><td>:: A :== {t}
a = {v_0, v_1, ⋯}
a = {e \\ p <-: a}
a.[i]
a![i][14]
{ a & [i] = e}</td><td>type A = Array Int t
array (0, n+1) [(0, v_0), (1, v_1), ⋯, (n, ⋯)]
array (0, length a-1) [e | (i, p) ← zip [0..] a]
a!i
(\v→(v!i, v)) a
a//[(i, e)]</td></tr>
<tr><th colspan="2">Dynamics</th></tr>
<tr><td>f :: a → Dynamic | TC a
f e = dynamic e
g :: Dynamic → t
g (e :: t) = e_0
g e = e_1</td><td>f :: Typeable a ⇒ a → Dynamic
f e = toDyn e
g :: Dynamic → t
g d = case fromDynamic d of
Just e → e_0
Nothing → e_1</td></tr>
<tr><th colspan="2">Function definitions</th></tr>
<tr><td>f p_0
q_0 = e_0
= e</td><td>f p_0
= e$[x := x']$
where $q_0[x := x']$ = e_0 — for each $x \in var(q_0) \cap var(e_0)$</td></tr>
</table>

[13]Requires RecordPuns to be enabled.
[14]Field selection from unique arrays.

```
:: BM a b = { ab :: a → b, ba  :: b → a }
bm :: BM a a
bm = {ab=id, ba=id}

:: Expr a
    = E.e: Lit (BM a e) e & toString e
    | E.e: Add (BM a e)    (Expr e) (Expr e) & + e
    | E.e: Eq  (BM a Bool) (Expr e) (Expr e) & == e
lit e = Lit bm e
add l r = Add bm l r
eq l r = Eq bm l r

eval :: (Expr a) → a
eval (Lit bm e)   = bm.ba e
eval (Add bm l r) = bm.ba (eval l + eval r)
eval (Eq  bm l r) = bm.ba (eval l == eval r)

print :: (Expr a) → String
print (Lit _ e)   = toString e
print (Add _ l r) = print l +++ "+" +++ print r
print (Eq  _ l r) = print l +++ "==" +++ print r
```

Listing 2: Expression GADT in Clean.

```
data Expr a where
    Lit :: Show a ⇒ a → Expr a
    Add :: Num a  ⇒ Expr a → Expr a → Expr a
    Eq  :: Eq e   ⇒ Expr e → Expr e → Expr Bool

eval :: Expr a → a
eval (Lit e)   = e
eval (Add l r) = eval l + eval r
eval (Eq  l r) = eval l == eval r

print :: Expr a → String
print (Lit e)   = show e
print (Add l r) = print l ++ "+"  ++ print r
print (Eq  l r) = print l ++ "==" ++ print r
```

Listing 3: Expression GADT in Haskell.

```
generic gEq a :: a a → Bool

gEq{|Int|}           x            y            = x == y
gEq{|Bool|}          x            y            = x == y
gEq{|Real|}          x            y            = x == y
gEq{|Char|}          x            y            = x == y
gEq{|UNIT|}          x            y            = True
gEq{|OBJECT|} f      (OBJECT x)   (OBJECT y)   = f x y
gEq{|CONS|}   f      (CONS x)     (CONS y)     = f x y
gEq{|RECORD|} f      (RECORD x)   (RECORD y)   = f x y
gEq{|FIELD|}  f      (FIELD x)    (FIELD y)    = f x y
gEq{|PAIR|}   fl fr (PAIR lx rx) (PAIR ly ry) = fl lx ly && fr rx ry
gEq{|EITHER|} fl _  (LEFT x)     (LEFT y)     = fl x y
gEq{|EITHER|} _  fr (RIGHT x)    (RIGHT y)    = fr x y
gEq{|EITHER|} _  _  _            _            = False

:: T = C1 Int ([Char], ?Bool) | C2
derive gEq [], T, (,), ?

Start = (gEq{|*|} C2 (C1 42 ([], ?Just True)), gEq{|*→*|} (<) [1,2,3] [2,3,4])
// (False, True)
```

Listing 4: Generic equality function in Clean..

```
generic gPrint a :: a [String] → [String]

gPrint{|Int|}          x         acc = [toString x:acc]
gPrint{|Bool|}         x         acc = [toString x:acc]
gPrint{|Real|}         x         acc = [toString x:acc]
gPrint{|Char|}         x         acc = [toString x:acc]
gPrint{|UNIT|}         x         acc = acc
gPrint{|PAIR|}   fl fr (PAIR l r) acc = fl l [" ":fr r acc]
gPrint{|EITHER|} fl _  (LEFT x)   acc = fl x acc
gPrint{|EITHER|} _  fr (RIGHT x)  acc = fr x acc

gPrint{|OBJECT|}       f     (OBJECT x) acc = f x acc
gPrint{|CONS of gcd|}  f     (CONS x)   acc = ["(", gcd.gcd_name, " ":f x [")":acc]]
gPrint{|RECORD of grd|} f    (RECORD x) acc = ["{", grd.grd_name, " | ":f x ["}":acc]]
gPrint{|FIELD of gfd|} f     (FIELD x)  acc = [pre, gfd.gfd_name, "=":f x acc]
where
    pre = if (gfd.gfd_index == 0) "" ", "

:: T = {f1 :: Int, f2 :: (Real, [?Int])}
derive gPrint (,), [], ?, T

Start = gPrint{|*|} {f1=42, f2=(3.14, [?None])} []
// {T | f1=42 , f2=(_Tuple2 3.14 (_Cons (_!None ) (_Nil )))}
```

Listing 5: Generic print function in Clean.

References

[1] Peter Achten. Clean for Haskell98 Programmers, July 2007.

[2] Artem Alimarine. *Generic Functional Programming.* PhD, Radboud University, Nijmegen, 2005.

[3] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical structures in computer science*, 6(6):579–612, 1996.

[4] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM, 2002.

[5] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[6] Ralf Hinze. Fun With Phantom Types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 245–262. Bloomsbury Publishing, Palgrave, 2003.

[7] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and Theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, pages 1–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[8] Johan Jeuring and Patrik Jansson. Polytypic programming. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, pages 68–114, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[9] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. Clean Language Report version 3.1. Technical report, Institute for Computing and Information Sciences, Nijmegen, December 2021.

[10] GHC Team. GHC User's Guide Documentation, 2021.