The following full text is an author's version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/34992

Please be advised that this information was generated on 2019-10-20 and may be subject to change.
Clean for Haskell98 Programmers

– A Quick Reference Guide –

Peter Achten, Radboud University Nijmegen, The Netherlands, P.Achten@cs.ru.nl

July 13, 2007

This note is meant to give people who are familiar with the functional programming language Haskell98 a concise overview of Clean language elements. The goal is to support the reader when reading Clean code. In the table on the other side of this page frequently occurring Clean language elements are summarized, and their Haskell98 counterpart is given next to it.

Obviously, this summary is not exhaustive. Notable Clean language elements that also occur frequently in Clean programs, but that do not appear in this summary are:

Strictness By default, Clean evaluates expressions lazily. Types can be annotated with strictness attributes (!) to indicate that they occur within a strict context. This is similar to Haskell98. Within a function definition, #! can be used to enforce evaluation of expressions. Haskell98 programmers will likely use the seq function.

The uniqueness type system Briefly, types and type variables can be annotated with a uniqueness attribute. This attribute can be *, which indicates that the type must occur within a unique context, and u:, which is a uniqueness attribute variable, which can be instantiated with * or not. The uniqueness type system allows Clean to use the world-as-value paradigm for side-effective programming: an interactive program is of type *World -> *World, where World represent the external environment of the program of which there can be only one, hence its uniqueness attribute. In Haskell98, such a program would have type IO (). The uniqueness type system also allows the Clean compiler to generate efficient code because uniquely attributed data structures can be destructively updated.

Generic programming With generic programming, the programmer defines a small set of instances of a generic function scheme that are used by the Clean compiler to derive for arbitrary data types the corresponding instance. Generic programming in Clean resembles generic programming in Generic Haskell.

Dynamic types Dynamic types allow the programmer to serialize and de-serialize arbitrary expressions (including functions), resulting in a new value of type Dynamic. Dynamic values can be stored on disk.

I hope you enjoy this note and that it will aid you in reading Clean programs.
Clean for Haskell98 Programmers

Basic types

(True,False) :: (Bool,Bool)
42 :: Int
3.1415926 :: Real
'A' :: Char
"Hello" :: String

Type definitions

:: T a1...an := t
:: T a1...an = C1 t1 | ... | Cn tn
:: T a1...an = { f1 :: t11...tn } data T a1...an = T { f1 :: t1, ..., fn :: tn }

Abstract data type definitions

definition module M
:: T ai .. .an / / no implementation

Function types

f :: t1 ... t0-1 -> t0 I Ci ai & ...& Cm am I f •• (Ci ai o •••? Cm am) -> t1 ...~> tn

Type classes

class f a :: t

class C a I Ci , ... o Cm a

instance (Ci a ,...,C m a) -> C a

as-patterns

x-:p

A-expressions

\pi ...\Pi n \rightarrow e or: \pi ...\Pi n \cdot e or: \pi ...\Pi n -> e

Distinction of cases

if c t e
case e of
  p1 -> e1 or: ...
  f p1...pn
  | c = t
  | otherwise = e

 List expressions

[1:[2:[3]]] :: [Int]
[e \ pi <- gi ]
[e \ pi <- gi , p ]
[e \ pi <- gi & p2 <- g2 ]

data R = { f :: t } data R = R { f :: t } r

Record expressions

:: R = { f :: t }
r = { f = e }

Record patterns

:: R1 = { f1 :: R1 }
:: R2 = { f2 :: t }
g1 { f1 } = e f1
g2 { f1 - f2 } = e f2

data R1 = R1 { f1 :: R2 } data R2 = R2 { f2 :: t }
g1 (R2 (f1=x)) = e x
g2 (R1 (f1,R2 (f2=x))) = e x

Array expressions

:: A ::= [t] a = [v0,...,vn-1] (see list comprehensions)
a = [v \ p <-: a ] (see list comprehensions)

Array types

:: A = Array Int t

a = array (0,n-1) [(0,v0),...,((n-1),vn-1)]
a = array (0,length a-1)

a!i

Comments

// single line comment
/* multi-line, /* nested, */ comment */

Function definitions

f p1
  # q1 = e1
  = e

f p1
  = e[x := x']

where q1[x := x'] = e1 -- for each x \in \var(q1) \cap \var(e1)