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 H∀skell.

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

– Quick Reference Guide –

Clean	Haskell98
Basic types	
(True,False) :: (Bool,Bool)	(True,False) :: (Bool,Bool)
42 :: Int	42 :: Int
3.1415926 :: Real	3.1415926 :: Float -- *or* Double
'A' :: Char	'A' :: Char
"Hello" :: String	"Hello" :: String
Type definitions	
:: T $a_1 \ldots a_n$:== t	type T $a_1 \ldots a_n$ = t
:: T $a_1 \ldots a_n$ = C_1 t_1 $\mid$ ... $\mid$ C_n t_n	data T $a_1 \ldots a_n$ = C_1 t_1 $\mid$ ... $\mid$ C_n t_n
:: T $a_1 \ldots a_n$ = { f_1 :: t_1, ..., f_n :: t_n }	data T $a_1 \ldots a_n$ = T { f_1 :: t_1, ..., f_n :: t_n }
Abstract data type definitions	
definition module M	module $M(T)$
:: T $a_1 \ldots a_n$ // *no implementation*	
Function types	
f :: t_1 ... t_{n-1} -> t_n $\mid$ C_1 a_1 & ...& C_m a_m	f :: (C_1 a_1, ..., C_m a_m) => t_1 -> ...-> t_{n-1} -> t_n
Type classes	
class f a :: t	class f a where f :: t
class C a $\mid$ C_1, ..., C_m a	class (C_1 a,..., C_m a) => C a
instance C t $\mid$ C_1, ..., C_m a where ...	instance (C_1 a,...,C_m a) => C t where ...
as-patterns	
x=:p	x@p
λ-expressions	
\ $p_1 \ldots p_n$ -> e *or:* \ $p_1 \ldots p_n$. e *or:* \ $p_1 \ldots p_n$ = e	\ $p_1 \ldots p_n$ -> e
Distinction of cases	
if c t e	if c then t else e
case e of case e of	case e of
p_1 -> e_1 *or:* p_1 = e_1	p_1 -> e_1
... ...	...
f $p_1 \ldots p_n$	f $p_1 \ldots p_n$
$\mid$ c = t	$\mid$ c = t
= e	$\mid$ otherwise = e
List expressions	
[1:[2:[3]]] :: [Int]	1:(2:(3:[])) :: [Int]
[e \\ p_1 <- g_1]	[e $\mid$ p_1 <- g_1]
[e \\ p_1 <- g_1 $\mid$ p]	[e $\mid$ p_1 <- g_1, p]
[e \\ p_1 <- g_1 , p_2 <- g_2]	[e $\mid$ p_1 <- g_1, p_2 <- g_2]
[e \\ p_1 <- g_1 & p_2 <- g_2]	[e $\mid$ (p_1,p_2) <- zipWith (,) g_1 g_2]
Record expressions	
:: R = { f :: t }	data R = R { f :: t }
r = { f = e }	r = R {e}
r.f	f r
r!f	(\v -> (f v,v)) r
{ r & f = e }	r { f = e }
Record patterns	
:: R_1 = { f_1 :: R_2 }	data R_1 = R_1 { f_1 :: R_2 }
:: R_2 = { f_2 :: t }	data R_2 = R_2 { f_2 :: t }
g_1 { f_1 } = e f_1	g_1 (R_1 {f_1=x}) = e x
g_2 { f_1 = { f_2} } = e f_2	g_2 (R_1 {f_1=R_2 {f_2=x}}) = e x
Array expressions	
:: A :== {t}	type A = Array Int t
a = {$v_0,\ldots,v_{n-1}$}	a = array (0,$n-1$) [(0,v_0),...,($n-1$,v_{n-1})]
a = {e \\ p <-: a} *(see list comprehensions)*	a = array (0,length a-1)
	[e $\mid$ (i,a) <- zipWith (,) [0..length a-1] a]
a.[i]	a!i
a![i]	(\v -> (v!i,v)) a
{a & [i]=e}	a//[(i,e)]
Comments	
// single line comment	-- single line comment
/* multi-line, /* nested, */ comment */	{- multi-line, {- nested, -} comment -}
Function definitions	
f p_1	f p_1
# q_1 = e_1	= $e[x := x']$
= e	where $q_1[x := x']$ = e_1 -- *for each* $x \in var(q_1) \cap var(e_1)$