CLEAN – UNIQUENESS TYPING

José-Ignacio Serna

May 19th 2014

Contents

Clean

- Language
- Features
- Sparkle
- Platform

Contents

Clean

- Language
- Features
- Sparkle
- Platform

Uniqueness Typing

- Intuition
- Definition

Contents

Clean

- Language
- Features
- Sparkle
- Platform

Uniqueness Typing

- Intuition
- Definition

Why?

- Efficient Space Management
- Interfacing with Non-functional Operations

Clean Language

Clean is a practical applicable general-purpose lazy pure functional programming language suited for the development of reald world applications.[1]

Haskell	Clean	Remarks
(a -> b) -> [a] -> [b]	(a -> b) [a] -> [b]	higher-order function
f . g	f o g	function composition
-5	~5	unary minus
[x | x <- [1..10] , isOdd x]	[x \\ x <- [1..10] | isOdd x]	list comprehension
x:xs	[x:xs]	cons operator

http://en.wikipedia.org/wiki/Clean_(programming_language)

[1] Rinus Plasmeijer, Marko van Eekelen, John van Groningen [2011]. Language report Version 2.2.

Features

- Strictness analyzer

```
[ 1,3..9 ]    // a lazy list
[! 1,3..9 ]   // a head strict list
[! 1,3..9 !]  // a strict list (head and spine)
[# 1,3..9 ]   // a head strict list, unboxed
[# 1,3..9 !]  // a strict list (head and spine), unboxed
[| 1,3..9 ]   // an overloaded list
```

Features

- Strictness analyzer

```
[ 1,3..9 ]    // a lazy list
[! 1,3..9 ]   // a head strict list
[! 1,3..9 !]  // a strict list (head and spine)
[# 1,3..9 ]   // a head strict list, unboxed
[# 1,3..9 !]  // a strict list (head and spine), unboxed
[| 1,3..9 ]   // an overloaded list
```

- Generic programming

Features

- Strictness analyzer

```
[ 1,3..9 ]    // a lazy list
[! 1,3..9 ]   // a head strict list
[! 1,3..9 !]  // a strict list (head and spine)
[# 1,3..9 ]   // a head strict list, unboxed
[# 1,3..9 !]  // a strict list (head and spine), unboxed
[| 1,3..9 ]   // an overloaded list
```

- Generic programming
- I/O library

Features

- Strictness analyzer

```
[ 1,3..9 ]    // a lazy list
[! 1,3..9 ]   // a head strict list
[! 1,3..9 !]  // a strict list (head and spine)
[# 1,3..9 ]   // a head strict list, unboxed
[# 1,3..9 !]  // a strict list (head and spine), unboxed
[| 1,3..9 ]   // an overloaded list
```

- Generic programming
- I/O library
- Dynamics

Sparkle

- **Proof assistant** written and specialized in **Clean** that uses tactics and a hint mechanism

Sparkle

- **Proof assistant** written and specialized in **Clean** that uses tactics and a hint mechanism
- Makes use of a subset of the language: **Core-Clean**

Sparkle

- **Proof assistant** written and specialized in **Clean** that uses tactics and a hint mechanism
- Makes use of a subset of the language: **Core-Clean**
- No support for pattern matching. Patterns have to be transformed to case distinctions

Sparkle

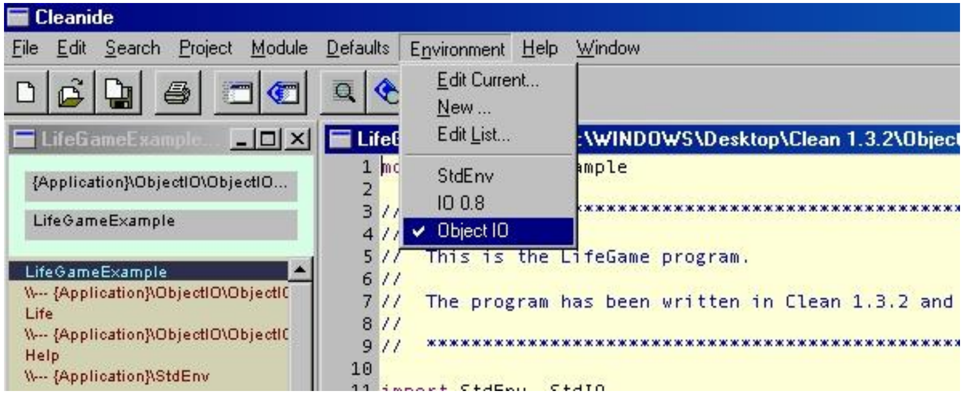
- **Proof assistant** written and specialized in **Clean** that uses tactics and a hint mechanism
- Makes use of a subset of the language: **Core-Clean**
- No support for pattern matching. Patterns have to be transformed to case distinctions
- **42** tactics, each is assigned a score between 1 and 100

Sparkle

- **Proof assistant** written and specialized in **Clean** that uses tactics and a hint mechanism
- Makes use of a subset of the language: **Core-Clean**
- No support for pattern matching. Patterns have to be transformed to case distinctions
- **42** tactics, each is assigned a score between 1 and 100

- Absurd
- AbsurdEquality
- Apply
- Assume
- Case
- ChooseCase
- Compare
- Exact
- Generalize
- Induction
- Injective
- Introduce
- MoveQuantors
- Reduce
- Reflexive
- Rewrite
- Split
- Symmetric
- Transitive
- Undo
- …

Clean Platform

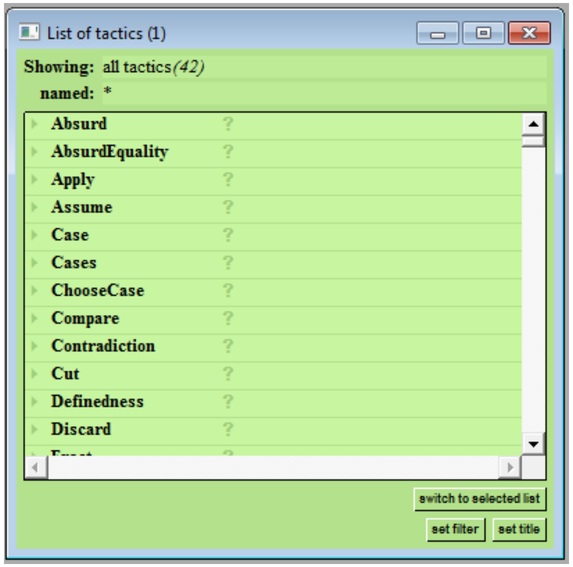

Cleanide
File Edit Search Project Module Defaults Environment Help Window
Edit Current...
New...
Edit List...
StdEnv
IO 0.8
✓ Object IO
LifeGameExample
LifeG
:\WINDOWS\Desktop\Clean 1.3.2\Object

{Application}\ObjectIO\ObjectIO...

LifeGameExample

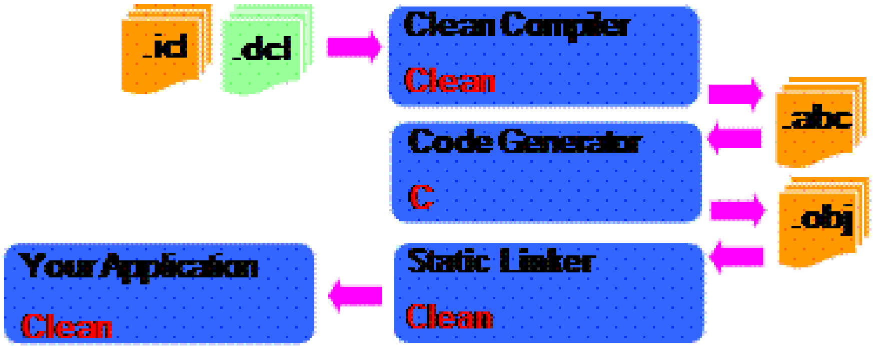
LifeGameExample
--- {Application}\ObjectIO\ObjectIO
Life
--- {Application}\ObjectIO\ObjectIO
Help
--- {Application}\StdEnv

```
mo              ample

//              ************************************
//
//  This is the LifeGame program.
//
//  The program has been written in Clean 1.3.2 and
//
//  ************************************************

import StdEnv, StdIO
```



Showing: all tactics *(42)*
named: *

- Absurd ?
- AbsurdEquality ?
- Apply ?
- Assume ?
- Case ?
- Cases ?
- ChooseCase ?
- Compare ?
- Contradiction ?
- Cut ?
- Definedness ?
- Discard ?



Uniqueness Typing: Intuition

“The type of a value is given a ‘unique’ attribute if that value is used at most once. On such ‘unique’ values update operations may be safely implemented in-place since their uniqueness guarantees that their value is no longer required by the program.” [2]

[2] Dana G. Harrington [2001]. A type system for destructive updates in declarative programming languages.

Uniqueness Typing: Definition

A uniqueness type is a pair $S = \langle\sigma, A\rangle$, where σ is a conventional type and A is a uniqueness attribute. The underlying conventional type σ is denoted $|S|$. (Also a more convenient notation is using superscripts).

Uniqueness Typing: Definition

A uniqueness type is a pair $S = \langle\sigma, A\rangle$, where σ is a conventional type and A is a uniqueness attribute. The underlying conventional type σ is denoted $|S|$. (Also a more convenient notation is using superscripts).

$$a^{v} \longrightarrow b^{w}, [v < w]$$

Uniqueness Typing: Definition

A uniqueness type is a pair $S = \langle \sigma, A \rangle$, where σ is a conventional type and A is a uniqueness attribute. The underlying conventional type σ is denoted $|S|$. (Also a more convenient notation is using superscripts).

$$a^{v} \longrightarrow b^{w}, [v < w]$$

```
fwritec :: Char *File -> *File
```

Why?

Adding uniqueness information provides a solution to two problems in implementations of functional languages.[3]

[3] Erik Barendsen and Sjaak Smesters [1993]. Conventional and Uniqueness Typing in Graph Rewrite Systems.

Why?

Adding uniqueness information provides a solution to two problems in implementations of functional languages.[3]

- Efficient space management

[3] Erik Barendsen and Sjaak Smesters [1993]. Conventional and Uniqueness Typing in Graph Rewrite Systems.

Why?

Adding uniqueness information provides a solution to two problems in implementations of functional languages.[3]

- Efficient space management
- Interfacing with non functional operations

[3] Erik Barendsen and Sjaak Smesters [1993]. Conventional and Uniqueness Typing in Graph Rewrite Systems.

Efficient Space Management

- Memory cells of `m` could be reused

```
let
  l = [1..10]
  m = map (*2) l
in
  m
```

Efficient Space Management

- Memory cells of `m` could be reused

```
let
  l = [1..10]
  m = map (*2) l
in
  m
```

- Memory cells of `m` can not be reused

```
let
  l = [1..10]
  m = map (*2) l
in
  (l,m)
```

Interfacing with Non-functional Operations

```
// C example
int foo( FILE *file ) {
  int a = fgetc(file );  // Read a character from ’file’
  int b = fgetc(file );
  return a + b;
}
```

Interfacing with Non-functional Operations

```
// Clean example
fgetc :: *File → (Char, *File)


foo :: *File → (Char, *File)
foo file0 = let (a, file1) = fgetc file0
                (b, file2) = fgetc file1
            in  (a + b, file2)
```