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 real world applications.¹

<table>
<thead>
<tr>
<th>Haskell</th>
<th>Clean</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(a -&gt; b) -&gt; [a] -&gt; [b]</code></td>
<td><code>(a -&gt; b) [a] -&gt; [b]</code></td>
<td>higher-order function</td>
</tr>
<tr>
<td><code>f . g</code></td>
<td><code>f o g</code></td>
<td>function composition</td>
</tr>
<tr>
<td><code>-5</code></td>
<td><code>~5</code></td>
<td>unary minus</td>
</tr>
<tr>
<td>`[ x</td>
<td>x &lt;- [1..10] , isOdd x]`</td>
<td>`[ x \ x &lt;- [1..10]</td>
</tr>
<tr>
<td><code>x:xs</code></td>
<td><code>[x:xs]</code></td>
<td>cons operator</td>
</tr>
</tbody>
</table>


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

- Strictness analyzer

```haskell
[ 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

\[
\begin{align*}
[ & 1,3..9 ] & \text{ // a lazy list} \\
[! & 1,3..9 ] & \text{ // a head strict list} \\
[! & 1,3..9 !] & \text{ // a strict list (head and spine)} \\
[# & 1,3..9 ] & \text{ // a head strict list, unboxed} \\
[# & 1,3..9 !] & \text{ // a strict list (head and spine), unboxed} \\
[| & 1,3..9 ] & \text{ // an overloaded list}
\end{align*}
\]

- 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
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. 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 \( \sigma \) is a conventional type and \( A \) is a uniqueness attribute. The underlying conventional type \( \sigma \) 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 $\sigma$ is a conventional type and $A$ is a uniqueness attribute. The underlying conventional type $\sigma$ is denoted $|S|$. (Also a more convenient notation is using superscripts).

$$a^v \rightarrow b^w, [v < w]$$
Uniqueness Typing: Definition

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

$$a^v \rightarrow b^w, [v < w]$$

fwritec :: Char *File -> *File
Why?

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

Why?

Adding uniqueness information provides a solution to two problems in implementations of functional languages.\(^3\)

- Efficient space management

---

Why?

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

- Efficient space management

- Interfacing with non functional operations

Efficient Space Management

• Memory cells of $m$ could be reused

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

in
  m
```
Efficient Space Management

• Memory cells of \( m \) could be reused

```haskell
let
    \( l = [1..10] \)
    \( m = \text{map } (*2) \, l \)
in
    \( m \)
```

• Memory cells of \( m \) can not be reused

```haskell
let
    \( l = [1..10] \)
    \( m = \text{map } (*2) \, l \)
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 \rightarrow (Char, *File)

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