Gentle Introduction to DOT calculus

ScalaMatsuri 2019

Kota Mizushima (DWANGO Co.,Ltd.)

Who am I?

- Twitter ID: @kmizu
- GitHub: kmizu
- Love: Scala/Rust/Nemerle/...
- Parsing algorithm enthuasiast

Scalaと構文解析アルゴリズムが大好きなプログラマです

Purpose

- Introduce the notaion of **core calculus**
  - abstract syntax, typing rule, operational semantics
- Introduce DOT calculus informally
- Show properties of DOT

- 核計算の概念、DOT計算の紹介などが目的です

Scala

- Designed by Martin Odersky in 2003
  - A FP researcher
- SCAlable LAnguage
- Latest: Scala 2.13.0
- Many industrial usages

- Martin Oderskyによって2003年に開発されました
- 多くの実用例があります

Dotty

- https://dotty.epfl.ch/
- Dotty will be Scala 3.0
- Many improvements
- Dotty is based on DOT calculus

- DottyはScala 3.0になる予定です
- DOT計算をベースにした言語です

DOT Calculus

- Core calculus of Scala 3.0
- DOT is abbreviation of *D*ependent *O*bject *T*ypes
- Core calculus:
  - Essense of a programming language
    - Abstract Syntax, Typing Rules, Semantics
  - Used to prove the behavior of the language

- DOT計算はScala 3.0の核言語です
- 核言語とはプログラミング言語の本質です

Why Core Calculus?

- Too difficult to model real programming languages
- Need good **subset** of programming languages

- 現実のプログラミング言語をモデル化することは難しすぎます
- 良いサブセットが必要です

History of Core Calculus in Scala

- vObj calculus (2003)
- Featherweight Scala (2006)
- DOT calculus (2012-)

- vObj calculus（2003年）
- Featherweight Scala（2006年）

Note

- DOT has several versions
  - since several DOT papers exists
- This presentation is based on
  - Nada Amin's dot repository
  - the paper of DOT appeared in OOPSLA'16
- Other papers may explain DOT in different manner

- Nada AminさんのリポジトリとOOPSLA'16の論文をベースにします
- 他の論文では違う形で説明されているかもしれません

What is Needed for Core Calculus ?

- Abstract Syntax
  - drop concrete syntax
- Typing rule
  - if the language has type system
- (Operational) semantics
  - like a naive implementation of an interpreter

- 抽象構文、型付け規則、（操作的）意味論が必要です
- 操作的意味論は、ナイーヴなインタプリタ実装のようなものです

Abstract Syntax

- Concrete syntax has extra information
  - spaces, tabs, commas, parenthesis, etc.
- An example of pseudo EBNF:
  - very simple calculation

```
E ::= P ("+" S P)*
P ::= "(" S E ")" S | I S
I ::= "0" | "1"
S ::= ("\t" | " " | "\r" | "\n)*
```

- 具象構文は余計な情報を含んでいます

Abstract Syntax

- Drop such extra information from concrete syntax
- Important to define *value*
  - *value* is a term which cannot be evaluated
- An example of abstract syntax in EBNF-like notation
  - very simple calculation
  - V is value

```
E ::= E + E | V
V ::= 0 | 1  // value
```

- 抽象構文は具象構文から余計な情報を除いたものです
- *値*を定義するのが重要です

Abstract Syntax - Intuition

```scala
sealed trait E
sealed trait V extends E
case object Zero extends V
case object One extends V
case class Plus(t1: E, t2: E) extends E
```

Typing Rules

- Rules for typing terms
- Γ(gamma) is called as **typing environments**
  - mappings from a variable to a type
- ⊢ is called as **typing judgements**
- An example:
  - very simple calculation

```
-----------------------------
Γ⊢ 0 : Int  Γ⊢ 1 : Int ...

Γ⊢ t1 : Int Γ⊢ t2 : Int
----------------------------
Γ⊢ t1 + t2 : Int
```

- Γは**型環境**と呼ばれます
- ⊢は**型判断**と呼ばれます

Typing Rule - Intuition

```
def ⊢(t, Γ) = t match {
  case Zero => Int
  case 1 => Int
  case (t1 `+` t2`) =>
    (⊢(t1, Γ), ⊢(t2, Γ)) match {
      case  (Int, Int) => Int
      case _ => TypeError
    }
}
```

Operational Semantics

- Rules for evaluating terms
- It can be seen as a naive interpreter

```
t1 → `t1`  t2 → `t2`
-------------------------
t1 `+` t2 → `t1` + `t2`
```

- ナイーヴなインタプリタのようなものと見ることができます

Operational Semantics - Intuition

```
def →(t) = t match {
  case (t: V) => t
  case (t1 `+` t2) =>
    val (`t1`, `t2`) =  (→(t1), →(t2))
    `t1` + `t2`
}
```

DOT: *D*ependent *O*bject *T*ypes

Characteristics of DOT's Type System

- Union and intersection types
  - e: Int | String, e: Number & Serializable
- Type members
  - e: Any { type M }
- Path-dependent types
  - e: v.x (v is constant)
- Combination of subtyping and these types

- 交差型（union type）、合併型（intersection type）など
- それらとサブタイピングが組み合わさっています

Syntax of DOT

- Citation from https://github.com/namin/dot

DOT:

Syntax

$t ::=$	**terms**:
$\quad x$	variable
$\quad \{z \Rightarrow \overline{d}\}$	object
$\quad t.m(t)$	method invocation
$d ::=$	**initialization**:
$\quad L = T$	type member
$\quad m(x : T) = t$	method member
$v ::=$	**values**:
$\quad \{z \Rightarrow \overline{d}\}$	object
$p ::=$	**paths**:
$\quad x$	variable
$\quad v$	value

$S, T, U ::=$	**types**:
$\quad \top$	top
$\quad \bot$	bottom
$\quad T \wedge T$	intersection
$\quad T \vee T$	union
$\quad L : S..U$	type member
$\quad m(x : S) : U$	method member
$\quad p.L$	selection
$\quad \{z \Rightarrow T\}$	recursive self
$\Gamma ::=$	**contexts**:
$\quad \emptyset \mid \Gamma, x : T$	variable bindings

Figure 1: DOT: Syntax

Subtyping in DOT (1)

- Citation from https://github.com/namin/dot

DOT:

Subtyping $\boxed{\Gamma \vdash S <: U}$

Lattice structure

$$\Gamma \vdash \bot <: T \quad \text{(Bot)}$$

$$\Gamma \vdash T <: \top \quad \text{(Top)}$$

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \quad \text{(And11)}$$

$$\frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \quad \text{(Or21)}$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \quad \text{(And12)}$$

$$\frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \quad \text{(Or22)}$$

$$\frac{\Gamma \vdash T <: T_1 \ , \ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \quad \text{(And2)}$$

$$\frac{\Gamma \vdash T_1 <: T \ , \ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T} \quad \text{(Or1)}$$

Type and method members

$$\frac{\Gamma \vdash S_2 <: S_1 \ , \ U_1 <: U_2}{\Gamma \vdash L : S_1..U_1 <: L : S_2..U_2} \quad \text{(Typ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_2 <: S_1 \\ \Gamma, x : S_2 \vdash U_1 <: U_2\end{array}}{\Gamma \vdash m(x : S_1) : U_1 <: m(x : S_2) : U_2} \quad \text{(Fun)}$$

Subtyping in DOT (2)

- Citation from https://github.com/namin/dot

Type selections

$$\frac{\Gamma_{[x]} \vdash x :_! (L : T..\top)}{\Gamma \vdash T <: x.L} \quad (\textsc{Sel2}) \qquad \frac{[z \mapsto \overline{d}]\overline{d} \ni L = T}{\Gamma \vdash T <: \{z \Rightarrow \overline{d}\}.L} \quad (\textsc{SSel2})$$

$$\frac{\Gamma_{[x]} \vdash x :_! (L : \bot..T)}{\Gamma \vdash x.L <: T} \quad (\textsc{Sel1}) \qquad \frac{[z \mapsto \overline{d}]\overline{d} \ni L = T}{\Gamma \vdash \{z \Rightarrow \overline{d}\}.L <: T} \quad (\textsc{SSel1})$$

Recursive self types

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2}{\Gamma \vdash \{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}} \quad (\textsc{Bind}) \qquad \frac{\Gamma, z : T_1 \vdash T_1 <: T_2 \quad z \notin fv(T_2)}{\Gamma \vdash \{z \Rightarrow T_1\} <: T_2} \quad (\textsc{Bind1})$$

Properties

$$\Gamma \vdash T <: T \quad (\textsc{Refl}) \qquad \frac{\Gamma \vdash T_1 <: T_2 \ , \ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \quad (\textsc{Trans})$$

Typing Rules of DOT

- Citation from https://github.com/namin/dot

Type assignment $\boxed{\Gamma \vdash t :_{(!)} T}$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x :_{(!)} T} \quad (\textsc{Var}) \qquad \frac{\Gamma \vdash t :_{(!)} T_1 \;,\; T_1 <: T_2}{\Gamma \vdash t :_{(!)} T_2} \quad (\textsc{Sub})$$

$$\frac{\Gamma \vdash p : [z \mapsto p]T}{\Gamma \vdash p : \{z \Rightarrow T\}} \quad (\textsc{Pack}) \qquad \frac{\Gamma \vdash p :_{(!)} \{z \Rightarrow T\}}{\Gamma \vdash p :_{(!)} [z \mapsto p]T} \quad (\textsc{Unpack})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : (m(x : T_1) : T_2) \;,\; t_2 : T_1 \\ x \notin fv(T_2)\end{array}}{\Gamma \vdash t.m(t_2) : T_2} \quad (\textsc{TApp}) \qquad \frac{\Gamma \vdash t : (m(x : T_1) : T_2) \;,\; p : T_1}{\Gamma \vdash t.m(p) : [x \mapsto p]T_2} \quad (\textsc{TAppDep})$$

$$\frac{\begin{array}{c}\text{(labels disjoint)} \\ \Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash d_i : T_i \qquad \forall i, 1 \leq i \leq n\end{array}}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : [x \mapsto \{x \Rightarrow d_1 \ldots d_n\}](T_1 \wedge \ldots \wedge T_n)} \quad (\textsc{TObj})$$

Member initialization $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma \vdash T <: T}{\Gamma \vdash (L = T) : (L : T..T)} \quad (\textsc{DTyp}) \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (m(x) = t) : (m(x : T_1) : T_2)} \quad (\textsc{DFun})$$

Oerational Semantics of DOT

- Citation from https://github.com/namin/dot

DOT:

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{[z \mapsto \overline{d}]\overline{d} \ni m(x : T_{11}) = t_{12}}{\{z \Rightarrow \overline{d}\}.m(v_2) \longrightarrow [x \mapsto v_2]t_{12}} \quad \text{(E-App)}$$

$$\frac{t_1 \longrightarrow t_1{}'}{t_1.m(t_2) \longrightarrow t_1{}'.m(t_2)} \quad \text{(E-App1)} \qquad \frac{t_2 \longrightarrow t_2{}'}{v_1.m(t_2) \longrightarrow v_1.m(t_2{}')} \quad \text{(E-App2)}$$

Figure 2: DOT: Small-Step Operational Semantics

Properties of DOT

- DOT type system is sound
  - soundness is very important property
    - typed terms don't do undefined behavior
  - a mechanized proof in Coq exists:
    - https://github.com/namin/dot
- Typing in DOT is undecidable
  - typing algorithm may not terminate
  - DOT can encode F<:, typing is also undecidable
  - typing in many programming languages is undecidable
    - Scala, Java, C++, TypeScript, etc.

- DOTの型システムは健全です
- DOTの型付けは決定不能です

Conclusion

- Core calculus is essence of programming languages
- DOT is core calculus of Scala 3
- Type system of DOT is sound
- Typing in DOT is undecidable (!)