論文紹介: Emerald: A General-Purpose Programming Language

久野*

1993.5.x

1 イントロ

Emerald は強い型を持つ OOPL だが、その設計方針はちょっと変わっている。

- プログラム単位はオブジェクトだけ。
- クラスはなく、object は object-constructer により作られる。
- 型は操作とそのシグニチャの集合。型もオブジェクト。
- 分散もサポート。

2 Emerald のオブジェクト

オブジェクト—自律性を持つ単位。オブジェクトを利用する方法—操作を呼び出すことだけ。

オブジェクトの生成

通常はクラス方式かプロトタイプ方式。Emerald では object-constructor。

```
const anIntergerNode ←
  object IntegerLiteral
    export getValue, setValue, generate
    monitor
      var value: Integer ← val
      operation getValue → [v: Integer]
        v ← value
      end getValue
      operation setValue[v: Integer]
        value ← v
      end setValue
      operation genarate[stream: OutStream]
        stream.printf["LDI %d\n", value]
      end generate
    end monitor
  end IntegerLiteral
```

constructor は実行時にオブジェクトのを作り出すような「式」。役割は:

- オブジェクトの生成。

*RAJ, R. K., Tempero, E., Levy, H. M.: Emerald: Emerald: A General-Puropos Programming Language, Software–Practice and Experience, vol. 21, no. 1, pp. 91-118, 1991

- 内部表現の規定。
- 操作の実現を記述。

複数オブジェクトを作りたければループ内で繰り返しこれを実行する。

```
count ← 5
loop
  exit when count = 0
  count ← count - 1
  nodes(count) ← object IntegerLiteral
                   % 上の通り
                 end IntegerLiteral
end loop
```

クラスのようなものを作りたければそのようにプログラムする。

```
const IntegerNodeCreator ←
  immutable object INC
    export new
    const IntegerNodeType ←
      type INT
        function getValue → [Integer]
        operation setValue[Integer]
        operation generate[OutStream]
      end INT
    operation new[val: Integer] → [aNode: IntegerNodeType]
      aNode ← object IntegerLiteral
                % 上と同じ
              end IntegerLiteral
    end new
  end INC
```

「唯一の」オブジェクト

たとえば ST80 では Boolean のサブクラス True/False は true/false という値だけを作る。が、間違って沢山作ったことがないとは保証できない。「あるオブジェクト」が「1 つだけ」しかできないようにするのは難しい。Emerald では。

```
const idServer ←
  object IDS
    export getNextId
    monitor
      var nextId: Integer ← 0
      operation getNextId → [newId: Integer]
        newId ← nextId
        nextId ← nextId + 1
      end getNextId
    end monitor
  end IDS
```

こういう時はクラスはいらないでしょう? というわけでメタクラスもまつわる混乱もいらない。

3 抽象型

Emerald ではオブジェクトは constructor が作る。型の方は、値を分類する (classification) ことだけに専念する。

型は操作とそのシグニチャの集合。変数や引数には型を宣言。型が合っていなければ入れられない (コンパイル時の型検査による検出)。

型が「合っている」ことを Emerald では conformity と呼ぶ。S conform to T とは ($S \subset T$ と書く)、

1. S は少なくともすべての T の操作を持つ。
2. 対応する操作ごとに、引数や返値の数は等しい。
3. S の各操作の結果型が対応する T の各操作の結果型に conform する。
4. T の各引数の型が対応する S の各引数の型に conform する。

なぜこうなっているかというと。

```
type JunkDeliverer
  operation Deliver → [Any]
end JunkDeliverer
type PizzaDeliverer
  operation Deliver → [Pizza]
end PizzaDeliverer
```

ここで JD を期待している変数に PD を入れるのは OK。逆はだめ。

```
type JunkHeap
  operation Deposit[Any]
end JunkHeap
type Bank
  operation Deposit[Money]
end Bank
```

今度は Bank を期待している変数に JH を入れるのは OK。逆はだめ。

これによって「勝手に」型階層ができる。例えば次の例。

```
const TreeNode ←
  type TN
    operation generate[OutStream]
  end TN
const Expression ←
  type E
    function getType → [Expression]
    operation generate[OutStream]
  end E
const Statement ←
  type S
    operation typeCheck
    operation generate[OutStream]
  end S
```

結局、Emerald ではプログラマは引数の型を指定するのに best-fitting type を指定するよう努力する。そうすればそこに可能なすべての値を渡すことができる。conformity の威力!

暗黙的/明示的 conformity

暗黙の conformity だけだと困ることもある。

```
const Stack ←
  object S
    export insert, remove
```

```
    operation insert[Element] → []
      ...
    operation remove → [Element]
      ...
  end S
const Queue ←
  object Q
    export insert, remove
    operation insert[Element] → []
      ...
    operation remove → [Element]
      ...
  end Q
const Stack2 ←
  object S2
    export insert, remove
    operation insert[Element] → []
      ...
    operation remove → [Element]
      ...
  end S2
```

ここで Stack を Stack2 に取り替えるのは許すが Queue に取り替えるのは困る、というのは表現できない。Trellis や Eiffell など明示的に関係を記述する場合はその問題はないが、逆にいつでも後から新版を持ってきて置き換えるようなことはできない。また Emerald でも Queue に queueLength のような操作があればそこに間違って Stack を入れることはない。

なお、Inheritance はないから、コードの共有はない。Emerald は分散指向だからそうなった。なおコード共有のための Jade と呼ばれる拡張も設計したがここでは述べない。

型とオブジェクトの関係

Emerald ではすべてのオブジェクトは複数の型を持つ (conformity のため)。言い替えれば O は

$$\text{typeof } O \subset T$$

なるすべての T を持つ。typeof は最大の型を返すものとする。型とは

```
immulable type AbstractType
  function getSignature → [Signature]
end AbstractType
```

に conform する任意のオブジェクトのこと。Signature 値は type...end type なる式によって作られる。これは built-in で、プログラマが自分で Signature 型に conform する値を作れるということはない。よって型検査は安全。例えば IntegerNode を次のように直すと、

```
const IntegerNode ←
  immutable object INC
    export getSignature, new
    const IntegerNodeType ←
      type INT
        function getValue → [Integer]
        operation setValue[Integer]
        operation generate[OutStream]
      end INT
    function getSignature → [sig: Signature]
      sig ← IntegerNode
    end getSignature
```

```
    operation new[val: Integer] → [aNode: IntegerNodeType]
      aNode ← object IntegerLiteral
                % 上と同じ
              end IntegerLiteral
    end new
  end INC
```

これは立派な型になる。

型と実現の分離

Emerald の道具 — constructor による生成、abstract type による界面の規定、conformity による比較 — によって、型の実現と界面は完全に分離されている。1 つの型に複数の実現をあてはめることが可能。例えばコンパイル済みで実行しているプログラムに、新たなモジュールを付け加えることすら可能。

多相

他相にもいろいろあるが。最低次のものは広く認められている。

- Inclusion Polymorphism
- Parametric Polymorphism
- Ad Hoc Porymorphism

エメラルドで可能な多相:

- 複数の型の引数について動く操作。
- 型を返すような操作。
- 多相値: 1 つの値が場所により複数の型として使われる。

最初のは簡単。

```
const SimplePolyTester ←
  object SPT
    const Printable ←
      type aPrintableType
        function asString → [String]
      end aPrintableType
    operatoin PrintIt[p: Printable]
      stdout.PutString[p.asString||"\n"]
    end PrintIt
    process
      SPT.PrintIt[0]
      SPT.PrintIt[1.1]
      SPT.PrintIt[true]
      SPT.PrintIt["This is not trivial."]
    end process
  end SPT
```

2 番目は、例えば array[T] のようなものを意味する。Emerald では型の値は翻訳時に決まらなければならないという制約を守って書く。

```
const List ←
  immutable object ListCreator
    export of
    function of[eType: AbstractType] → [result: ListCreator]
      where
        ListCreator ←
          immutalbe type LC
            function getSignature → [Signature]
            operation new → [ListType]
          end LC
        ListType ←
          type LT
            operation AddAsNth[eType, Integer]
            operatoin DeleteNth[Integer]
            function GetNth[Integer] → [eType]
            function Length → Integer
          end LT
      end where
      result ←
        immutable object NewListCreator
          export getSignature, new
          function getSignature → [r: Signature]
            r ← ListType
          end getSignature
          operatoin new → [result: ListType]
            result ←
              object theList
                export AddAsNth, DeleteNth, GetNth, Length
                % ...
              end theList
          end new
        end NewListCreator
    end of
  end ListCreator
```

ここで List.of[Integer] という式は「型」を返す。だから

```
var iList: List.of[Integer] ← List.of[Integer].new
```

のようにして汎用のリストを作ることができる。型でありかつ constructor にもなれるのに注意。

3 番目のは、1 つの実体がさまざまな型の変数に入るというだけのこと。特に問題ない。

以下は型とは別の話題なので略。