

第 16 章 Hindley/Milner 型推論 (Hindley/Milner Type Inference)

この章では、Hindley/Milner スタイル[Mil78] の型推論システムの開発を通して、Scala のデータ型とパターンマッチングを見てみます。型推論器の対象言語は、mini-ML と呼ばれる let 構成子を持つ計算です。mini-ML の抽象構文木は、次のデータ型 Term によって表現されます。

```
abstract class Term {  
  case class Var(x: String) extends Term {  
    override def toString = x  
  }  
  case class Lam(x: String, e: Term) extends Term {  
    override def toString = "(" + x + "." + e + ")"  
  }  
  case class App(f: Term, e: Term) extends Term {  
    override def toString = "(" + f + " " + e + ")"  
  }  
  case class Let(x: String, e: Term, f: Term) extends Term {  
    override def toString = "let " + x + " = " + e + " in " + f  
  }  
}
```

木のコンストラクタが 4 つあります。変数を表す Var、関数抽象を表す Lam、関数適用を表す App、そして let 式を表す Let などです。それぞれのケースクラスでは、Any クラスのメソッド toString をオーバーライドして、項をわかりやすく表示しています。

次に、推論システムによって計算される型を定義します。

```
sealed abstract class Type {  
  case class Tyvar(a: String) extends Type {  
    override def toString = a  
  }  
  case class Arrow(t1: Type, t2: Type) extends Type {  
    override def toString = "(" + t1 + ">" + t2 + ")"  
  }  
  case class Tycon(k: String, ts: List[Type]) extends Type {  
    override def toString =  
      k + (if (ts.isEmpty) "" else ts.mkString("[", ", ", "]"))  
  }  
}
```

型コンストラクタが 3 つあります。型変数を表す Tyvar、関数の型を表す Arrow、そして Boolean や List などのような型コンストラクタを表す Tycon などです。型コンストラクタは構成要素として型パラメータのリストを持ちます。このリストは Boolean のような型定数については空になります。型コンストラクタもまた、型をわかりやすく表示するために toString メソッドを実装しています。

Type が sealed(封印された)クラスである事に注意してください。これは、Type を拡張するサブクラスやデータコンストラクタを、Type が定義されている定義列の外部では作れない事を意味します。これにより、Type は **クローズ** な代数データ型で、代替物が確実に 3 つである事が保証されます。逆に、型 Term は **オープン** な代数型であり、さらなる代替物を定義できます。

型推論器の主な部品は object typeInfer に含まれます。新しい型変数を生成するユーティリティ関数から始めましょう。

```
object typeInfer {  
  private var n: Int = 0  
  def newTyvar(): Type = { n += 1; Tyvar("a" + n) }
```

次に、代入を表すクラスを定義します。代入は、型変数から型への冪等な (何度同じ操作をしても同じ結果となる) 関数です。有限数の型変数を何らかの型にマップし、ほかのすべての型変数はそのまま変えません。代入の意味は、各点でのマッピングから、複数の型から複数の型へのマッピングへと拡張されます。

```
abstract class Subst extends Function1[Type, Type] {  
  def lookup(x: Tyvar): Type  
  
  def apply(t: Type): Type = t match {  
    case tv @ Tyvar(a) => val u = lookup(tv); if (t == u) t else apply(u)  
    case Arrow(t1, t2) => Arrow(apply(t1), apply(t2))  
    case Tycon(k, ts) => Tycon(k, ts map apply)  
  }  
}
```

```

def extend(x: Tyvar, t: Type) = new Subst {
  def lookup(y: Tyvar): Type = if (x == y) t else Subst.this.lookup(y)
}
val emptySubst = new Subst { def lookup(t: Tyvar): Type = t }

```

代入は、`Type => Type` 型の関数で表します。クラス `Subst` を 1 変数の関数型 `Function1[Type, Type]`(*1) を継承させる事で実現できます。この型のインスタンスであるためには、代入 `s` は、引数として `Type` をとり結果として他の `Type` を戻す `apply` メソッドを実装する必要があります。これにより、関数適用 `s(t)` は `s.apply(t)` と解釈されます。

(*1) このクラスは、関数型を直接のスーパークラスではなくミックスインとして継承しています。これは、Scala の現在の実装では `Function1` 型が、他のクラスの直接のスーパークラスとして使用できない、Java の `interface` であるためです。

`lookup` は `Subst` クラスの抽象メソッドです。代入には 2 つの具体的な形態があり、このメソッドの実装方法が異なります。一つは、`emptySubst` 値によって定義されているもの、もう一つはクラス `Subst` の `extend` メソッドによって定義されているものです。

次のデータ型は、型のスキームを記述するもので、型と型の変数名リストからなり、型変数は型スキームにおいて普遍量化されて登場します。たとえば、型スキーム `a b.a b` は、型チェッカーにおいて次のように表現されます。

```
TypeScheme(List(Tyvar("a"), Tyvar("b")), Arrow(Tyvar("a"), Tyvar("b"))),
```

型スキームのクラス定義には、`extends` 節がありません。これは、型スキームはクラス `AnyRef` を直接拡張しているということです。型スキームを構築できる方法がひとつしかなくても、ケースクラス表現を選びました。この型のインスタンスを部分に分解する便利な方法が必要だったからです。

```

case class TypeScheme(tyvars: List[Tyvar], tpe: Type) {
  def newInstance: Type = {
    (emptySubst /: tyvars) ((s, tv) => s.extend(tv, newTyvar())) (tpe)
  }
}

```

型スキームオブジェクトには、`newInstance` メソッドがあります。これは、普遍量化された型変数を新しい変数に変名した後、そのスキームに含まれる型を返します。このメソッドの実装では、拡張オペレーションによって型スキームの型変数を `(/:` で) 畳み込みます。拡張オペレーションは、与えられた代入 `s` について、与えられた型変数 `tv` を新しい型変数に変名することで、`s` を拡張します。得られた代入は、型スキームのすべての型変数を新しい名前に変名します。次にこの代入は、型スキームの型部分に適用されます。

型推論器に必要な最後の型は、環境を表す `Env` です。変数名と型スキームを関連づけます。これは、`typeInfer` モジュール内の型エイリアス `Env` によって表されます。

```
type Env = List[(String, TypeScheme)]
```

「環境」には 2 つのオペレーションがあります。`lookup` 関数は、与えられた名前に関連づけられた型スキームを返します。名前がその環境に記録されていない場合は `null` を返します。

```

def lookup(env: Env, x: String): TypeScheme = env match {
  case List() => null
  case (y, t) :: env1 => if (x == y) t else lookup(env1, x)
}

```

`gen` 関数は与えられた型を、その環境になく、かつ、型で自由であるすべての型変数を量化することで、型スキームに変えます。

```

def gen(env: Env, t: Type): TypeScheme =
  TypeScheme(tyvars(t) diff tyvars(env), t)

```

ある型の自由な型変数の集合は、単に、その型に現れているすべての型変数です。ここでは、次のように構築される型変数のリストとして表現されます。

```

def tyvars(t: Type): List[Tyvar] = t match {
  case tv @ Tyvar(a) =>
    List(tv)

```

```

case Arrow(t1, t2) =>
  tyvars(t1) union tyvars(t2)
case Tycon(k, ts) =>
  (List[Tyvar]() /: ts) ((tvs, t) => tvs union tyvars(t))
}

```

最初のパターンにある文法 $tv @ \dots$ は、それに続くパターンに束縛された、変数を導入します。また、三つ目の節の式にある、明示的な型パラメータ $[Tyvar]$ は、ローカルな型推論を働かせるために必要です。型スキームの自由な型変数の集合は、量化された型変数を除いた、その型コンポーネントの自由な型変数の集合です。

```

def tyvars(ts: TypeScheme): List[Tyvar] =
  tyvars(ts.tpe) diff ts.tyvars

```

結局、環境の自由な型変数の集合は、その環境に記録されたすべての型スキームの自由な型変数の集合です。

```

def tyvars(env: Env): List[Tyvar] =
  (List[Tyvar]() /: env) ((tvs, nt) => tvs union tyvars(nt._2))

```

Hindley/Milner型チェックの中心的操作は単一化であり、それは2つの与えられた型を等しくする代入計算です(そのような代入は**単一化代入**と呼ばれます)。関数 mgu は、事前の代入 s 下の2つの与えられた型 t と u の最も大きな単一化代入を計算します。それは、 s を拡張する最も大きな代入 s' を返し、 $s'(t)$ と $s'(u)$ を同じ型にします。

```

def mgu(t: Type, u: Type, s: Subst): Subst = (s(t), s(u)) match {
  case (Tyvar(a), Tyvar(b)) if (a == b) =>
  case (Tyvar(a), _) if !(tyvars(u) contains a) =>
    s.extend(Tyvar(a), u)
  case (_, Tyvar(a)) =>
    mgu(u, t, s)
  case (Arrow(t1, t2), Arrow(u1, u2)) =>
    mgu(t1, u1, mgu(t2, u2, s))
  case (Tycon(k1, ts), Tycon(k2, us)) if (k1 == k2) =>
    (s /: (ts zip us)) ((s, tu) => mgu(tu._1, tu._2, s))
  case _ =>
    throw new TypeError("cannot unify " + s(t) + " with " + s(u))
}

```

もし単一化代入がなければ mgu 関数は `TypeError` 例外を送出します。このことは、2つの型が対応する場所で異なる型コンストラクタを持つとき、あるいは型変数が型変数自身からなる型と単一化されるときに起こり得ます。そのような例外を、あらかじめ定義された `Exception` クラスから継承する、ケースクラスのインスタンスとしてモデル化してみます。

```

case class TypeError(s: String) extends Exception(s) {}

```

型チェックの主な仕事は関数 tp によって実装されます。この関数はパラメータとして環境 env 、項 e 、プロトタイプ t 、事前の代入 s をとります。関数 tp は s を拡張して代入 s' をもたらし、 $s'(env) \vdash e:s'(t)$ を Hindler/Milner 型システム[Mil178] の推論規則に従って推論可能な型判定に変えます。もしそのような代入が存在しなければ `TypeError` 例外が送出されます。

```

def tp(env: Env, e: Term, t: Type, s: Subst): Subst = {
  current = e
  e match {
  case Var(x) =>
    val u = lookup(env, x)
    if (u == null) throw new TypeError("undefined: " + x)
    else mgu(u.newInstance, t, s)

  case Lam(x, e1) =>
    val a, b = newTyvar()
    val s1 = mgu(t, Arrow(a, b), s)
    val env1 = {x, TypeScheme(List(), a)} :: env
    tp(env1, e1, b, s1)

  case App(e1, e2) =>
    val a = newTyvar()
    val s1 = tp(env, e1, Arrow(a, t), s)

```

```

    tp(env, e2, a, s1)

    case Let(x, e1, e2) =>
      val a = newTyvar()
      val s1 = tp(env, e1, a, s)
      tp({x, gen(env, s1(a))} :: env, e2, t, s1)
  }
}
var current: Term = null

```

エラー診断を支援するため、tp 関数は直前に解析されたサブ項を現変数に保存します。したがって、もし型チェックが `TypeError` 例外でアボートされれば、この変数が問題を引き起こしたサブ項を含みます。

型推論モジュールの最後の関数 `typeOf` は、tp の簡略化版です。与えられた環境 `env` 内の与えられた項 `e` の型を計算します。新しい型変数 `a` を生成し、型付け代入を計算して `env e:a` を推論可能な型判定にし、代入を `a` に適用した結果を返します。

```

def typeOf(env: Env, e: Term): Type = {
  val a = newTyvar()
  tp(env, e, a, emptySubst)(a)
}
} // end typeInfer

```

型推論を適用するにあたり、よく使われる定数まわりの環境を事前に定義しておくとう便利です。事前定義されたモジュールでは、`boolean`、数字、リスト等の型について、それらのプリミティブな操作とともに、関係するものを含む環境 `env` を定義しています。不動点操作 `fix` も定義しており、再帰を表現するのに使えます。

```

object predefined {
  val booleanType = Tycon("Boolean", List())
  val intType = Tycon("Int", List())
  def listType(t: Type) = Tycon("List", List(t))

  private def gen(t: Type): typeInfer.TypeScheme = typeInfer.gen(List(), t)
  private val a = typeInfer.newTyvar()
  val env = List(
    {"true", gen(booleanType)},
    {"false", gen(booleanType)},
    {"if", gen(Arrow(booleanType, Arrow(a, Arrow(a, a))))},
    {"zero", gen(intType)},
    {"succ", gen(Arrow(intType, intType))},
    {"nil", gen(listType(a))},
    {"cons", gen(Arrow(a, Arrow(listType(a), listType(a))))},
    {"isEmpty", gen(Arrow(listType(a), booleanType))},
    {"head", gen(Arrow(listType(a), a))},
    {"tail", gen(Arrow(listType(a), listType(a)))},
    {"fix", gen(Arrow(Arrow(a, a), a))}
  )
}

```

型推論の使い方の例を見てみましょう。あらかじめ定義された環境 `Predefined.env` 内で計算される、与えられた項の型を返す関数 `showType` を定義しましょう。

```

object testInfer {
  def showType(e: Term): String =
    try {
      typeInfer.typeOf(predefined.env, e).toString
    } catch {
      case typeInfer.TypeError(msg) =>
        "\n cannot type: " + typeInfer.current +
        "\n reason: " + msg
    }
}

```

アプリケーションを起動すると

```
> testInfer.showType(Lam("x", App(App(Var("cons"), Var("x")), Var("nil"))))
```

次のように応答するでしょう。

```
> (a6->List[a6])
```

演習 16.0.1 Mini-ML 型推論器を、letrec 構成子で再帰関数を許容するように拡張しなさい。構文：

```
letrec ident "=" term in term .
```

letrec の形は、定義された識別子が定義している式に見えていることを除き、let と同じです。letrec を使えば、リスト用の length 関数を次のように定義できます。

```
letrec length = \xs.  
  if (isEmpty xs)  
    zero  
    (succ (length (tail xs)))  
in ...
```

[前ページ](#) [16章](#) [目次](#) [次ページ](#)

名前:	<input type="text"/>
コメント:	<input type="text"/>