はじめての型推論

ドキュメントバージョン 1.0.5

元のドキュメントはここにあります: TypeInferencerリポジトリ: https://github.com/kekyo/TypeInferencer

このドキュメントは、以下の読者を想定しています:

- `TypeInferencer` を試したい。
- 型推論や型推論器に興味を持たれた方。
- 型推論や型推論器の初学者。

以下は解説していません:

- 型推論器の内部の詳細な解説（別途、関連論文や型理論に関する解説書を参照してください。また、 `TypeInferencer` は出来るだけ平易な実装となるようにしています。）

なお、私自身も精通しているわけではないので、誤っている箇所があるかもしれません。このドキュメントは、私の理解を整理する目的でもあります。

（日本語で記述してから、機械翻訳で英語に翻訳して手直ししています。そのため、英語版で表現に不適な箇所があるかもしれません。明らかにおかしな文法や、わかりにく言い回しなどは、PRなどで助けてもらえると非常に嬉しいです。）

1. 型推論の概要

`TypeInferencer` は、コンピューター言語処理系で使われる「型推論」を、平易かつコンパクトに実装したものです。一般的に、型推論の処理は、言語処理系の内部に実装されています。
我々が最もなじみのある `F#` 、 `OCaml` 、 `Haskell` 、 `Java` など、静的型システムを持つ言語処理系です。あるいは、動的型システムでも、実行の最適化の為に何らかの型推論を行っているかもしれません。

型推論とは、式の型を明示することなく、処理系によって型を自動的に計算して示すことです。
例えば:

```
// [*] F#の式 - (1)
let f x = x + 1

// [*] 推論した結果 - (2)
let f (x:int) : (int -> int) = x:int +:(int -> int -> int) 1:int
```

- 注意: F#の疑似コードと、 `TypeInferencer` を試すコードとを区別出来るように、目印として疑似コードには `[*]` を示します。

通常、人間がコードを記述する時には (1) のように型を明示しません。そこで、型推論器を使用することによって、(2) のように、式の項それぞれにどのような型が適用されうるかを、自動的に計算・導出できます。

型推論には様々な手法が考案されていますが、 `TypeInferencer` は、型推論の導入で代表的な `Algorithm W` と `Algorithm M` という手法を実装していて、使い分ける事が出来ます。

- `Algorithm W` は、式の構文木を上(root)から、下(leaf)に向かって、型を確定していきます。
- `Algorithm M` は逆に、式の構文木の下(leaf)から、上(root)に向かって、型を確定していきます。

（構文木とは、抽象構文木(Abstract syntax tree、略してAST)の事です。後で示します。）

どちらも、推論した結果の意味に違いは出ません。はじめに `Algorithm W` が示され、のちに `Algorithm M` が示されました。何故、異なる方法が研究されたのかというと、 `Algorithm M` のほうが、型の問題を発見した箇所を局所化できるからです。

`TypeInferencer` では、型推論中に問題が発生しても、具体的な発生個所を細かく明示しませんが、多くの処理系やIDEは、誤った箇所の指摘を、出来るだけ局所的なコードの位置で表示したいと思うかもしれません。そのような場合に、 `Algorithm M` は、より望ましい結果を出力できます。

2. TypeInferencerが出来ること

`TypeInferencer` は、型推論器です。通常、コンピューター言語処理系には、以下の要素が含まれます:

1. `Lexer` と `Parser`：ソースコードはテキストベースで記述します。これをコンピューターが認識しやすい構造に再解釈するのが最初のステップです。 `Lexer` と `Parser` はこの役割を担い、ソースコードから構文木を生成します。

2. `Type checker`：型検査器です。構文木の各ノードに適切な型付けを行います。あるいは、適切な型付けが行われているかどうかを検査します。

3. `Reducer` 又は評価器: 構文木の各ノードを評価します。評価することは、結果的に式を実行することに繋がります。一般的には「インタプリタ」に相当します。

4. `Code generator`：構文木から、独立したコードを生成します。ターゲットで直に動作するコードを生成します。一般的には「コンパイラ」に相当します。

全ての言語処理系が、すべての機能を実装しているとは限りません。例えば、動的型システムを元とする処理系では、2と4のステップを省略するかもしれません。

`TypeInferencer` は、2の `Type checker` の実装の一部と言えます。そのため、入力として構文木を用意する必要があります。これは、以下のように書く事が出来ます:

```
// NuGetパッケージのロード
#r "nuget: TypeInferencer"

// 使用する名前空間とモジュール
open TypeInferencer

// （ここまでは定型、以降の例では省略）

// ----------------------

// 以下のコード片と同等の、構文木を組み立てる
// `let f = fun x -> x 1 in f`
let expr =
    ELet("f",                               // let f =
        EAbs("x",                           // fun x ->
            EApp(EVar "x", ELit(LInt 1))    // x 1
        ),
        EVar "f"                            // in f
    )

// （この後、TypeInferencerを使う）
```

もっと自然な形で式を書けないものか? と思われた方も居るかもしれません。つまり、コメントに書いたような式を、直接文字列で:

```
// ソースコードを文字列で書き、
let code = "let f = fun x -> x 1 in f"

// 構文木に変換できないか?
let expr = parse code
```

のように書ければ、より便利でしょう。そのような変換を行うのが、先に示した `Lexer` と `Parser` の役割です。従って、 `TypeInferencer` には、このような機能は含まれていません。

しかし、がっかりする必要はありません。実務として、そのような課題に取り組む必要があるなら、それは既に、皆さんの手の中にあります。F#コンパイラサービス です。

これは、過不足なく、完全なF#の機能を網羅しています。そこには、 `Lexer` 、 `Parser` 、 `Type checker` 、 `Code generator` の公開APIが全てが含まれていて、個別に使用したり、組み合わせて使用する事が出来ます。あなたのアプリケーションに組み込んで使用する事も可能でしょう。

しかし、F#の言語仕様は複雑であり、その構造を示すための構文木もまた膨大で複雑です。多くの人は、F#の構文木を見ても、何故そのようなノード定義が必要なのかを理解するのは容易ではなく、扱うのも困難でしょう。

その点、 `TypeInferencer` は、型推論の機能のみに絞っていて、構文木の種類も最小限に絞っています。つまり:

- 構文木とは何か。
- 型推論の具体的な方法。
- 型推論を実行すると何が起きるのか、又は特定の構文木がどのように推論されるのか。

について、調べやすくなっている筈です。

特に、F# 6.0以降、デバッグ情報が大幅に改善されたので、デバッガを使って内部実装をトレースすると、理解の助けになると思います。

（その場合は、NuGetパッケージではなく、このリポジトリをクローンしてテストコードからデバッグ実行すると良いでしょう）

恐らくは、 `TypeInferencer` のような素朴な型システムの構造を理解できれば、F#コンパイラサービスのような、より複雑で実践的な言語処理系を使用したメタプログラミングを実践できるでしょう。

3. TypeInferencerの構文木

構文木のノードは以下のように再帰的に定義されています:

```
// 構文木の各ノードを示す型
type public Exp =
    | EVar of name:string
    | ELit of literal:Lit
    | EApp of func:Exp * arg:Exp
    | EAbs of name:string * expr:Exp
    | ELet of name:string * expr:Exp * body:Exp
    | EFix of func:string * name:string * expr:Exp
```

ノードは全部で6種類あります。それぞれについて、以下に説明します。

4. ELit （リテラル値）

まずは、最も理解しやすくて簡単な、リテラル値の型を推論してみましょう。コード中のリテラル値は、 `ELit` と言うノードで表現します。以下は、 `123` という整数値を定義して、それを推論したものです:

```
// 直値: 123
let expr = ELit (LInt 123)

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

- `TopDown` は、 `BottomUp` でも問題ありません。 `Algorithm W` を使うか、 `Algorithm M` を使うかの違いです。 `TypeInferencer` においては、結果の意味に違いは生まれません。
- `TypeEnv` は「型環境」と呼びますが、今は空リストから生成するコードとしておきます。
- `show` 関数は、得られた型を、人間が読みやすい文字列に変換します。リテラル値の型は、 `Int` のように、プレフィックス `T` が無い結果が得られます。完全な定義を得たい場合は、 `printfn "%A" actual` のようにして下さい。

`infer` 関数にこの構文木を適用すると、 `TInt` という値が帰ってきます。これはつまり、 `123` という値は、整数型であると推論したのです。

同様に:

```
// 直値: true
let expr = ELit (LBool true)

// actual = TBool
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

この場合は、 `TBool` が返されます。文字通り、真偽型であると推論しました。

`ELit` の引数に指定できる `Lit` 型の定義は、以下の通りです:

```
// ELitの引数指定に使う型
type public Lit =
    | LInt of value:int32
    | LBool of value:bool
```

他にも様々な種類の値を試したくなるかもしれませんが、その前に `TInt` や `TBool` の定義も見て下さい:

```
// 構文木の型を示す型（TVarとTFunについては後述）
type public Type =
    | TInt
    | TBool
    | TVar of name:string
    | TFun of parameterType:Type * resultType:Type
```

`TInt` や `TBool` は、 `Type` 型で定義された値です。（慣れるまでは何が型で何が値なのかで、混乱するかもしれません。また、 `Type` 型は、.NETの `System.Type` 型ではなく、独自に定義したものです。）

実は、 `TypeInferencer` が認識できる型は、この4種類しかありません。つまり、リテラル値については、整数型と真偽型しかないのです。そのため、他の型（例えば、浮動小数点型や文字列型など）に対応させるには、このバリエーションを増やすか、もう少し内部実装を工夫する必要があります。

他にも疑問点があります。構文木のノードを、わざわざ `LInt` と `LBool` で区別して定義しているのだから、それらの型が `TInt` と `TBool` に帰着するのは当たり前ではないか？ という点です。

実際、内部実装でも、このようにハードコーディングで結果を返しています。

`TypeInferencer` は、型定義を増やしたり、柔軟性を導入したりしませんでした。リテラル値の型を増やす改造は比較的容易ですが、先ほど示したように、値・型・型の値、など、慣れていないと混乱するためです。

今の `TypeInferencer` の内部構造は、F#が実装上の整合性を（文字通りF#の型システムで）担保してくれるおかげで、理解を助けてくれます。また、実装の元となった論文を、出来るだけ忠実に再現する事が目的であったためでもあります。

5. EAbsとEVar（匿名関数と自由変数）

匿名関数とは、F#で言うところの、`fun`を使った式の事です:

```
// [*] 匿名関数を定義して、fと言う名前で束縛する
let f = fun x -> x + 1
```

このような式を`EAbs`で定義します（`let`は含んでいないことに注意）:

```
// 匿名関数を定義
// `fun x -> 123`
let expr = EAbs("x", ELit(LInt 123))

// actual = a0 -> TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

この構文木の評価結果は、`a0 -> TInt`で、この型は`TFun`で定義されます。これは、「不明な型`a0`の値を受け取り、`TInt`型を返す関数型」という意味です。

`EAbs`の本体式は`LInt`なので、固定的に`TInt`を返すという事は分かります。一方で、パラメータ変数の`x`は何も情報が無く、かつ、本体式内で使用もされていないため、推論する材料がありません。結局、`x`の型は不明なまま推論が終了し、パラメータ引数部が`a0`型となっているのです。

不明な型`a0`は、プレースホルダとして機能していて、推論中に型が判明すれば、その型に収束します。このプレースホルダの事を、「型変数」（又は自由型変数）と呼び、`TVar`で表されます。

関数型`TFun`と、型変数`TVar`は、以下のように手動でも定義できます:

```
// a0 -> TInt
let typ = TFun(TVar "a0", TInt)

// 完全に一致することを確認する - (1)
System.Diagnostics.Debug.Assert((typ = actual))
```

型の内容を検査するには、パターンマッチ式を利用できます:

```
// 完全に一致することを確認する - (2)
System.Diagnostics.Debug.Assert(
    match actual with
    | TFun(TVar name, TInt) when name = "a0" -> true
    | _ -> false
)
```

`TypeInferencer`は、プレースホルダに、機械的に生成された番号を付与します。この番号は、構文木の構造によって変わる可能性がある事に注意してください。従って、手動で定義するよりも、推論された他の`TVar`と一致しているかどうかを確認するために使用することをお勧めします。

ここで、パラメータ引数の変数`x`を参照可能にする、`EVar`についても説明します（`TVar`ではありません）:

```
// 匿名関数を定義し、本体式でパラメータの変数を使う
// `fun x -> x`
let funcexpr = EAbs("x", EVar "x")

// actual = a0 -> a0
let actual = infer TopDown (TypeEnv []) funcexpr
printfn "%s" (show actual)
```

参照したい変数を `EVar` で指定する事が出来ます。このような変数指定を「自由変数」と呼びます。（パラメータ変数は、束縛されているとみなす事が出来るので、「束縛変数」と呼ぶ場合もあります）

`EVar` と `TVar` は似ていますが、それぞれ示す対象が、具象値か型か、という違いがあります。

結果の `a0 -> a0` は、「全体としては関数型で、引数と結果の型は両方とも不明なものの、両者は一致する」という事を示しています。

匿名関数を定義だけしても、あまり面白くないでしょう。次の関数適用が必要です。

6. EApp（関数適用）

関数適用とは、以下のように関数を定義して、その関数を引数に適用する事です:

```
// [*] F#で匿名関数を定義
let f = fun x -> x
// let f x = x      // F#ではletでも定義可能

// [*] 引数に関数を適用(結果は123)
f 123
```

`EApp` は、このような、関数適用を定義するノードです:

```
// 匿名関数を定義
// `fun x -> x`
let funcexpr = EAbs("x", EVar "x")

// 123に匿名関数を適用
// `(fun x -> x) 123`
let expr = EApp(funcexpr, ELit(LInt 123))

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

これこそが推論と呼べる例にたどり着きました。この匿名関数は、パラメータ引数 `x` がそのまま返されているため、 `x` の型と同じ型が返される、と推論できるはずです。従って、関数適用で与えられた `123` の型 `TInt` が、匿名関数から返される、と推論出来て、結果として `TInt` が得られます。同様に:

```
// 匿名関数をtrueに適用する
// `(fun x -> x) true`
let expr = EApp(funcexpr, ELit(LBool true))

// actual = TBool
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

同じ匿名関数の定義でありながら、与えられた引数の型によって推論された結果も変わり、確かに推論されていることが確認できます。

では、もう少し、異なる例も見てみましょう:

```
// 123に関数trueを適用 (?)
// `true 123`
let expr = EApp(ELit(LBool true), ELit(LInt 123))

// uncaught UnificationException
let actual = infer TopDown (TypeEnv []) expr
```

`true 123` という式は、見るからに落ち着きません。一見して、これは駄目だろうと分かりますが、どうして駄目なのかを考えてみます:

- `true` はリテラル値です。その型は真偽型、つまり `TBool` です。
- `123` に `true` を適用するという事は、 `true` は関数であると仮定しています。関数の型については `EAbs` で説明した通り、関数型、つまり `TFun` です。

想定される関数型がどういうものかを考えると、「整数型を引数として受け取り、結果の型が不明な関数型」と考えられます。つまり、 `TInt -> a0` のようになるはずです。

そして、真偽型と、関数型 `TInt -> a0` では、どうやっても整合出来ません。 `TypeInferencer` の「同一化」という処理中でこの問題が発見され、例外がスローされたのです。

7. Coffee break

匿名関数の定義で、面白い例を示します:

```
// `fun x = x x`
let expr = EAbs("x", EApp(EVar "x", EVar "x"))

// uncaught OccurrenceException
let actual = infer TopDown (TypeEnv []) expr
```

この式は例外が発生します。実は、この構文木を推論することは出来ません。理由を考えてみて下さい。今夜は眠れないかもしれません…

8. ELet （束縛）

これまで、F#のコード例で散々示してきた `let` に相当するノードです。念のために、おさらいしておきます:

```
// [*] 束縛する
let x = 123

// [*] 束縛した変数を参照
printfn "%d" x
```

最近、F#を使い始めた方は、もしかしたら気が付いていないかもしれません。この記法は「軽量構文」と呼ばれていて、実は以下のように書け、意味的にはこちらが正確です（「冗長構文」と呼びます）:

```
// [*] 束縛する
let x = 123
in
  // [*] 束縛した変数を参照(明示的なスコープ内)
  printfn "%d" x
```

`in` は、束縛のスコープを決めていて、変数 `x` はこのスコープ内でのみ使用できます。 `ELet` はこれを忠実に定義します:

```
// `let x = 123 in x`
let expr =
    ELet("x",
        ELit(LInt 123),
        EVar "x"   // スコープ本体の式
    )

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

この構文木は、結局 `123` を返すので、推論結果は `TInt` です。

もちろん、匿名関数を束縛することもできます:

```
// `let f = fun x -> x in f 123`
let expr =
    ELet("f",
        EAbs("x", EVar "x"),
        EApp(EVar "f", ELit(LInt 123))
    )

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

もうこの時点で、人間が型の推論を正確に行うのは難しくなってきたのではないでしょうか。
もっと興味深い例もあります:

```
// `let f = fun x -> fun y -> x y in f (fun z -> z) 123`
let expr =
    ELet("f",
        EAbs("x", EAbs("y", EApp(EVar "x", EVar "y"))),
        EApp(EApp(EVar "f", EAbs("z", EVar "z")), ELit(LInt 123))
    )

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

関数の型や、その関数を適用した結果の型など、正しく追跡されていることが分かると思います。この例では、 `x` が関数型として評価される、という所が面白いと思います。

このように、複雑な構文木を眺めていると、ある事に気が付きます。以下のコード例を、良く眺めてみてください:

```
// [*] 束縛式
let x = true in 123

// [*] 匿名関数と関数適用の組
(fun x -> 123) true
```

私は、この事を、暗黙に理解していたはずですが、実際に推論された結果を見て驚きました。つまり、束縛式は、匿名関数と関数適用の組で代用できます。これを確かめてみます:

```
// `let x = true in 123`
let expr1 = ELet("x", ELit(LBool true), ELit(LInt 123))

// `(fun x -> 123) true`
let expr2 = EApp(EAbs("x", ELit(LInt 123)), ELit(LBool true))

// actual1 = actual2 = TInt
let actual1 = infer TopDown (TypeEnv []) expr1
let actual2 = infer TopDown (TypeEnv []) expr2
printfn "%s = %s" (show actual1) (show actual2)
```

余談: Javascriptでは、変数のスコープがかなり緩いため、誤って値を再代入する危険があります。そのような場合に、ここで示したような、匿名関数と関数適用を使用して、疑似的なスコープを作るテクニックがあります。

なお、ここでは示しませんが、もちろん `ELet` のネストも可能です。これは、皆さんが自分で確かめてみてください。

9. EFix （不動点）

一般的に、再帰的な関数は、 `let` を使うだけでは定義できません:

```
// [*] 再帰関数を定義する
let f x = f x    // fは定義されていない
in
  f 123
```

F#では、 `let rec` を使うことで、再帰関数を定義できます:

```
// [*] 正しい定義
let rec f x: int = f x   // 結果の型が推論出来ないため、仮にintとする(型注釈)
in
  f 123
```

- 注意: もちろん、この式は無限ループに陥ります。

`let` で実現できない理由として、束縛する式内に現れる自由変数のスコープが、 `let` 束縛式の外にあるためです。 `ELet` で示した通り、束縛変数を参照できるスコープは、 `let ... in ...` の `in` の後です。F#の `let rec` はこの制限を緩和します。

`TypeInferencer` では、 `EFix` を使うことで、再帰関数を定義可能にします:

```
// `fix f x = f x`
let expr =
    EFix("f",
        "x",
        EApp(EVar "f", EVar "x"))

// actual = a0 -> a1
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

実際には、これで再帰関数（そしてその型は `a0 -> a1` ）が定義されたにすぎません。 `let rec` と同じ式にするには、以下のように束縛を行います:

```
// `let f = fix fi x = fi x in f 123`
let expr =
    ELet("f",
        EFix("fi",   // 分かりやすくするために`fi`と名前を変えています(任意)
            "x",
            EApp(EVar "fi", EVar "x")),
        EApp(EVar "f", ELit(LInt 123)))

// actual = a0
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

F#のコード例で結果の型が推論できなかったように、 `TypeInferencer` も、推論することは出来ませんでした。結果の値を返すようにすれば、その値の型を足掛かりに推論する事が出来ます。

なお、`EFix`の名前は、「不動点」から来ています。不動点の概念は直感的に理解しづらいので、様々な文献を参照すると良いと思います。（Wikipediaの例）

10. 制限と発展的話題

以上が、`TypeInferencer`が持つ構文木と推論の解説です。本文中にも一部示しましたが、実用的な言語処理系と比較して、以下のような制限とその解決方法があります:

- リテラル値の種類が、整数型と真偽型しかありません。
  - `ELit`が、ホスト環境の直値（.NETであれば`obj`から継承されるすべてのインスタンス）を、直接扱うように変更するのは、それほど難しくありません。

- 実用的な計算を行うためのノード種が足りません。
  - 例えば、パターンマッチ式や`if ... then ... else`に相当する式が無いため、現実的な式を定義できません。`EFix`の解説で、停止条件を実装する事が出来ないため、現実的な再帰関数の例を示せませんでした。

  - `EIf`のようなノードを定義することは可能で、十分に研究されているので、各種論文を追ってみると良いでしょう。

  - 改めて述べるまでも無いかもしれませんが、ループを構成する`for`や`while`に類似する構文木は、再帰関数で構成できるため、`EFix`を応用すれば定義できます。もちろん、これらに直接対応するノード種を定義してみるのも良いと思います。その場合は、関数が値を返すかどうか、と言う、関数型言語の根幹にかかわる問題に向き合う必要があります。

  - ノードの種類を増やさずに工夫する、と言う方法もあります。「ラムダ計算」は、極めて原始的ではありますが、関数の定義と関数適用さえあれば、全ての計算が可能である、という理論です。`TypeInferencer`には、既に`EAbs`も`EApp`もあるので、ラムダ計算に従った構文木を構築できます。

- 構文木の各ノードに、型注釈を適用できません。
  - `EFix`のF#の例で示したように、一部の項に手動で型を注釈したほうが、うまく推論出来たり、私たちの理解の助けになることがあります。構文木の部分的な型注釈についても、既存の研究成果を参照すると良いでしょう。

- 型の合成が出来ません。
  - F#のような関数型言語は、タプル型や代数データ型が定義可能です。これにより、もっと柔軟で複雑な定義が出来ますが、`TypeInferencer`にはそのようなノード定義と推論機能がありません。

  - 代数データ型を扱うには、レコードの構造をどのように扱うかや、部分的なフィールドの整合を認めるかどうかという、複雑な問題があります。

- 部分型付けが出来ません。
  - 部分型付けとは、OOPにおける型の継承関係や実装関係により、より抽象度の高い型への整合を認める、というものです。

  - 既に見てきたように、`TypeInferencer`では、型が不明（`TVar`）・`TInt`・`TBool`・`TFun`しかないため、互換性のある別の型、という型を定義できません。これらは、不明な型に対して、互換性があるかどうかという「制約条件」を計算することで実現出来ます。

- 推論結果は、常に式全体に対しての型のみ返します。
  - IDEで、文脈位置に対応した型をサジェストするためには、部分的な構文木のノードに対応した推論結果を使う必要がありますが、`TypeInferencer`は、推論の途中で得ら

れた型を捨てています。

- 各ノードの推論結果を保存するようにすれば、この問題を解決できるでしょう。

これらについて、`TypeInferencer`の実装から、改造して付け足していくのも良いでしょう。上記を包括的に網羅する書籍として、「型システム入門」（和訳版）が有名ですが、難易度はかなり高いため、初心者には向きません。まずは、論理学や記号論理学に関する書籍で基礎を確認してから進むと良いと思います。