22 Type Reconstruction

担当 川崎 禎紀*

2003 年 6 月 17 日

22 Type Reconstruction

これまでの章で見てきた λ 計算の型チェックアルゴリズムは，明示的な型注釈 (type annotation) に依存している．具体的には，λ 抽象では引数の型を注釈として付けてやる必要があった．この章では，より強力な type reconstruction アルゴリズムを構成する．このアルゴリズムは，一部，或いは全ての型注釈が付けられていない term の principal type を求めることが出来る．ML や Haskell の様な言語の中心的存在として，これと同種のアルゴリズムが使われている．

22.1 Type Variables and Substitutions

これまで Nat や Bool のような具体的な base type の代わりに用いてきた uninterpreted base type[1]を，他の型による代入や具体化 (instanciate) の可能な型変数 (type variable) として扱い，term が typable かどうかを考える．

DEFINITION 22.1.1
type substitution(以下，型代入と言ったり代入と言ったりする) とは型変数から型への有限の写像である．σ を型代入としたとき，σ 中の左辺に現れる型変数の集合を $dom(\sigma)$，右辺に現れる型の集合を $range(\sigma)$ と書く．(domain と range に同じ型変数が現れても構わないことに注意)

型[2]への代入の適用は以下の様に定義される．

$$
\begin{array}{lcl}
\sigma(\mathsf{X}) & = & \begin{cases} \mathsf{T} & \text{if } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \\ \mathsf{X} & \text{if } \mathsf{X} \notin dom(\sigma) \end{cases} \\
\sigma(\mathsf{Nat}) & = & \mathsf{Nat} \\
\sigma(\mathsf{Bool}) & = & \mathsf{Bool} \\
\sigma(\mathsf{T}_1 \to \mathsf{T}_2) & = & \sigma\mathsf{T}_1 \to \sigma\mathsf{T}_2
\end{array}
$$

例 1 $\sigma = [\mathsf{X} \mapsto \mathsf{Bool}]$ とすると $\sigma(\mathsf{X} \to \mathsf{X}) = \mathsf{Bool} \to \mathsf{Bool}$ となる．

*kwsk@is.s.u-tokyo.ac.jp
[1]例えば，λx:A.x における A のこと (11.1)
[2]この章では symply typed lambda-calculus(Fig 9-1) with booleans(8-1), numbers(8-2), and an infinite collection of base types(11-1) を扱う

例 2 $\sigma = [\mathsf{X} \mapsto \mathsf{Bool}, \mathsf{Y} \mapsto (\mathsf{X} \to \mathsf{X})]$ とすると，$\sigma\mathsf{Y} = \mathsf{X} \to \mathsf{X}$ となる．代入は同時に適用されるので $\mathsf{Bool} \to \mathsf{Bool}$ とはならない．

型環境 Γ への代入の適用は次の様に定義する．

$$\sigma(\mathsf{x}_1 : \mathsf{T}_1, ..., \mathsf{x}_n : \mathsf{T}_n) = (\mathsf{x}_1 : \sigma\mathsf{T}_1, ..., \mathsf{x}_n : \sigma\mathsf{T}_n)$$

term t への代入の適用は，t 中の全ての型の出現に代入を適用したものとする．
代入の合成は次のように定義する．

$$\sigma \circ \gamma = \begin{bmatrix} \mathsf{X} \mapsto \sigma(\mathsf{T}) & \text{for each } (\mathsf{X} \mapsto \mathsf{T}) \in \gamma \\ \mathsf{X} \mapsto \mathsf{T} & \text{for each } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \text{ with } \mathsf{X} \notin \mathit{dom}(\gamma) \end{bmatrix}$$

型代入の重要な性質として，変数を含む term が well typed ならば，その term の instance も全て well typed であるという性質がある．

THEOREM 22.1.2 [PRESERVATION OF TYPING UNDER TYPE SUBSTITUTION]
σ を型代入とすると，$\Gamma \vdash \mathsf{t} : \mathsf{T}$ のとき $\sigma\Gamma \vdash \sigma\mathsf{t} : \sigma\mathsf{T}$ である．
Proof: $\Gamma \vdash \mathsf{t} : \mathsf{T}$ の導出に関する帰納法により示す．最後の規則で場合分けをする． □

22.2 Two Views of Type Variables

term t と型環境 Γ を，それぞれ型変数を含むことができるとする．t の型付けについて次の二つの観点から考えていく．

1. 任意の σ に対して，$\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ となるか？
2. $\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ となるような，σ が存在するか？

1 は要するに，well typed な t 中の型変数を後でどんな具体的な型で置き換え (instanciate) ても well typed か？ということである．この考え方は parametric polymorphism(22.7) につながる．(より詳しくは 23 章)

2 は要するに，well typed でない t を，適切な代入を適用することで well typed にできるか？ということである．例えば λf:Y. λa:X. f (f a) は typable ではないが，[Y ↦ Nat → Nat, X ↦ Nat] を適用すると λf:Nat → Nat. λa:Nat. f (f a) となり (Nat → Nat) → Nat → Nat という型を持つ．この考え方は type reconstruction(或いは type inference(型推論)) につながる．

DEFINITION 22.2.1 (DECLARATIVE) solution
Γ を型環境，t を term とする．(Γ, t) の *solution* とは $\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ を満たす (σ, T) のことである．

22.3 Constraint-Based Typing

型環境 Γ と term t から，(Γ, t) の solution が満たしていなければならない制約 (型の等式) の集合を求めるアルゴリズムを考える．

DEFINITION 22.3.1
制約集合 (constraint set) C とは等式の集合 $\{\mathsf{S}_i = \mathsf{T}_i{}^{i\in 1..n}\}$ のことである．$\sigma\mathsf{S}$ と $\sigma\mathsf{T}$ が等しい時 σ は $\mathsf{S} = \mathsf{T}$ を unify するという．C の全ての式を σ が unify するとき，σ は C を unify(又は satisfy) するという．

DEFINITION 22.3.2 *constraint typing relation* $\Gamma \vdash \mathsf{t} : \mathsf{T} \mid_\chi C$ (別紙 Figure 22-1)

この関係は,「制約集合 C が満たされている時，型環境 Γ の下で term t は型 T を持つ」と読める．CT-APP にでてくる $FV(\mathsf{T})$ とは T 中の全ての型変数のことである．χ は新規に導入する型変数がその rule の部分導出で使われた型変数と異なることと，異なる部分導出の間で導入された型変数が全て異なることを保障するためだけに用いられている．(以下，必要の無い場合 χ を省略して書く)

この rule から，Γ と t から $\Gamma \vdash \mathsf{t} : \mathsf{T} \mid_\chi C$ を満たす T と C(と χ) を求める手続きが得られる．通常の simply typed lambda-calculus の typing argorithm と異なる点は fail しない，つまり T と C が常に求まる (しかも一意に) ということである．

constraint typing relation の考え方は，まず t が型 S を持つために満たすべき C を求め，次に C を unify する σ を求める，という二段構えなのである．

DEFINITION 22.3.4 (ALGORITHMIC) solution
$\Gamma \vdash \mathsf{t} : \mathsf{S} \mid C$ のとき，(σ, T) が $(\Gamma, \mathsf{t}, \mathsf{S}, \mathrm{C})$ の solution であるとは，σ が C を unify し，$\sigma\mathsf{S} = \mathsf{T}$ であることである．

この ALGORITHMIC な solution と先ほどの DECLARATIVE な solution は等価である．

THEOREM 22.3.5 [SOUNDNESS OF CONSTRAINT TYPING]
$\Gamma \vdash \mathsf{t} : \mathsf{S} \mid C$ のとき，(σ, T) が $(\Gamma, \mathsf{t}, \mathsf{S}, \mathrm{C})$ の solution ならば，(σ, T) は (Γ, t) の solution である．
Proof: $\Gamma \vdash \mathsf{t} : \mathsf{S} \mid C$ の導出に関する帰納法により示す．最後の規則で場合分けをする． □

DEFINITION 22.3.6
χ に含まれる変数については未定義で，それ以外については σ と同じ代入を $\sigma \setminus \chi$ で表す．

THEOREM 22.3.7 [COMPLETENESS OF CONSTRAINT TYPING]
$\Gamma \vdash \mathsf{t} : \mathsf{S} \mid_\chi C$ のとき，(σ, T) が (Γ, t) の solution で，かつ $dom(\sigma) \cap \chi = \varnothing$ ならば，$(\Gamma, \mathsf{t}, \mathsf{S}, \mathrm{C})$ の solution (σ', T) で $\sigma' \setminus \chi = \sigma$ を満たすものが存在する．
Proof: $\Gamma \vdash \mathsf{t} : \mathsf{S} \mid_\chi C$ の導出に関する帰納法により示す．最後の規則で場合分けをする． □

22.4 Unification

C を unify する σ を求めるために unification アルゴリズムを導入する (別紙 Figure 22-2)．

DEFINITION 22.4.1
$\sigma' = \gamma \circ \sigma$ となる γ が存在するとき，σ は σ' より *less specific*(又は *more general*) であると言い，

$\sigma \sqsubseteq \sigma'$ と書く．

DEFINITION 22.4.2
C を unify する全ての σ' に対して $\sigma \sqsubseteq \sigma'$ が成り立つ時，σ が C の *principal unifier*(又は *most general unifier*(mgu)) であると言う．

THEOREM 22.4.5

1. *unify*(C) は，全ての C に対し fail するか代入を返して停止する．
2. *unify*$(C) = \sigma$ ならば，σ は C を unify する．
3. δ が C を unify するならば，*unify*$(C) = \sigma$ で $\sigma \sqsubseteq \delta$

Proof:

1. C の degree (m, n) を，C 中の異なる型変数の種類を m，C の型のサイズ[3]を n として定義する．unify がすぐに停止 (成功または fail) する場合以外は，C の degree を辞書式順序で小さくするように unify の再帰呼び出しが行われる．よって停止する．
2. *unify*(C) の計算の再帰呼び出しの回数に関する帰納法で示せる．σ が $[\mathsf{X} \mapsto \mathsf{T}]D$ を unify するならば，$\sigma \circ [\mathsf{X} \mapsto \mathsf{T}]$ は $\{\mathsf{X} = \mathsf{T}\} \cup D$ を unify する．
3. *unify*(C) の計算の再帰呼び出しの回数に関する帰納法で示せる．

□

22.5 Principal Types

DEFINITION 22.5.1
$(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の solution (σ, T) が *principal solution* であるとは，全ての $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の solution (σ', T') に対して $\sigma \sqsubseteq \sigma'$ となることである．また，(σ, T) が principal solution であるとき，T を Γ のもとでの t の *principal type* と言う．

THEOREM 22.5.2 [PRINCIPAL TYPES]
$(\Gamma, \mathsf{t}, \mathsf{S}, C)$ が solution を持つとき，principal solution を持つ．また，unification アルゴリズムを使うことで $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ が solution を持つかどうかを判定し，solution を持つならば principal solution を得ることが出来る．
Proof: $(\Gamma, \mathsf{t}, \mathsf{S}, C)$ の solution の定義と，unification アルゴリズムの性質から明らか． □

以上より，(Γ, t) が solution を持つかどうか決定可能であることが言えた．

いままで見てきた type reconstruction アルゴリズムの考え方を発展させて，制約を全て求めてからそれを解くのではなく，各ステップで制約を求めるのとそれを解くのを交互に行うことができる．こうすることで，プログラムの型エラーを正確に指摘することが出来る．

[3] S=T の場合，S=$\mathsf{S}_1 \to \mathsf{S}_2$ かつ T=$\mathsf{T}_1 \to \mathsf{T}_2$ の場合それぞれで n が小さくなるように適当にサイズを定義する．base type と型変数と $\to$ の出現個数の総和など．

22.6 Implicit Type Annotations

言語に type reconstruction を取り入れることで，λ 抽象の type annotation を完全に取り除くことが出来る．簡単には，syntax sugar として parser の段階で fresh な型変数を付加してやることでこれを実現できる．

より良い方法として文法に annotation の無い λ 抽象を追加して，それに対応した次のような constraint typing relation を導入する方法がある．

$$\frac{\mathsf{X} \notin \chi \quad \Gamma, \mathsf{x}{:}\mathsf{X} \vdash \mathsf{t}_1 \ : \ \mathsf{T} \mid_{\chi} C}{\Gamma \vdash \lambda \mathsf{x}.\mathsf{t}_1 \ : \ \mathsf{X} \to \mathsf{T} \mid_{\chi \cup \{\mathsf{x}\}} C} \text{ (CT-ABSINF)}$$

この様にする事で，abstraction を複製したときに，その複製それぞれについて異なる型変数を付加することが出来る．

22.7 Let-Polymorphism

polymorphism という言葉は，プログラムのある部分を，異なる場所で異なる型として使うことが出来る言語機能の事を指す．type reconstruction アルゴリズムは let-porymorphism として知られるシンプルな polymorphism を実現する．これは list や array などのデータ構造の generic library を作る基礎となっている．

例えば，

```
let doubleNat = λ f:Nat → Nat. λ a:Nat. f(f(a)) in
let doubleBool = λ f:Bool → Bool. λ a:Bool. f(f(a)) in
let a = doubleNat succ 1 in
let b = doubleBool not true in
...
```

というようなプログラムがあったとき，型注釈以外は同じなのだから

```
let double = λ f:X → X. λ a:X. f(f(a)) in
let a = double succ 1 in
let b = double not true in
...
```

とまとめて書いて，double が使われる場面ごとにそれぞれ適切に型付けしてもらえれば嬉しい，という感じである．しかし，今までの方法ではそれぞれの場面で X という同じ型変数が使われてしまうので，すべての場面で double は同じ型を持たなければならない (という制約がでてくる)．

これを解決するために，まず型付け規則を変更する．

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \quad \Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let\ x{=}t_1\ in\ t_2} \ : \ \mathsf{T}_2} \text{ (T-LET)}$$

上のように let の束縛の右辺 t_1 の型を求めて，その型で拡張した型環境で let の body t_2 の型を求める代わりに，次のように，t_2 中の xの出現を t_1 で置き換えるようにする．

$$\frac{\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}_1]\mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let\ x{=}t_1\ in\ t_2} \ : \ \mathsf{T}_2} \text{ (T-LETPOLY)}$$

constraint-typing rule も同様に変更する．

$$\frac{\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}_1]\mathsf{t}_2 : \mathsf{T}_2 \mid_\chi C}{\Gamma \vdash \mathsf{let\ x{=}t_1\ in\ t_2\ :\ T_2} \mid_\chi C} \text{ (CT-LETPOLY)}$$

この変更は実質的に，型を求める前に次の評価を行うことに相当する．

$$\mathsf{let\ x{=}v_1 in\ t_2} \longrightarrow [\mathsf{x} \mapsto \mathsf{v}_1]\mathsf{t}_2 \text{ (E-LETV)}$$

次に，22.6 で扱った implicitly annotated lambda-abstraction(CT-ABSINF) を導入すればよい．プログラムは次のようになる．

```
let double = λ f. λ a. f(f(a)) in
let a = double succ 1 in
let b = doublel not true in
...
```

しかし，実際にはこれでもまだ問題は残っている．例えば，

```
let x = 1 2 (* apply 2 to 1 ?? *) in 3
```

というようなプログラムが型チェックを通ってしまう．let で束縛される変数が let の body で出現しないというのが原因である．これを防ぐために型付け規則を次のように変更して t_1 の型チェックを行うようにする．

$$\frac{\Gamma \vdash [\mathsf{x} \mapsto \mathsf{t}_1]\mathsf{t}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{let\ x{=}t_1\ in\ t_2\ :\ T_2}} \text{ (T-LETPOLY)}$$

CT-LETPOLY も同様に変更する．

反対に，let で束縛される変数が let の body で多く出現する場合を考える．let で束縛した変数 x の出現の数だけ束縛の右辺 t_1 の型チェックが行われることになるのだが，束縛の右辺にも let が出現し得るので，最悪の場合，型チェックにかかる時間は term のサイズに対して指数時間となってしまう．そこで，実際の言語では次のように Γ の下での $\mathsf{let\ x{=}t_1\ in\ t_2}$ の型チェックを行う．

1. constraint typing を用いて $\Gamma \vdash \mathsf{t}_1 : \mathsf{S}_1 \mid C_1$ となる型 S_1 と制約集合 C_1 を求める．
2. unification を用いて C_1 の mgu σ を求め，σ を S_1 に適用して，t_1 の principal type T_1 を得る．
3. T_1 中の型変数で Γ に現れないものを $\mathsf{X}_1 ... \mathsf{X}_n$ とすると，$\forall \mathsf{X}_1 ... \mathsf{X}_n.\mathsf{T}_1$ を t_1 の (principal) type scheme とする．
4. 型環境に $\mathsf{x}{:}\forall \mathsf{X}_1 ... \mathsf{X}_n.\mathsf{T}_1$ を追加して拡張したもので，body である t_2 の型チェックを行う．
5. t_2 の型チェック時に，x が出現した場合，まず fresh な型変数 $\mathsf{Y}_1 ... \mathsf{Y}_n$ を用意して，そこでの x の型として type scheme を instanciate した $[\mathsf{X}_1 \mapsto \mathsf{Y}_1, ..., \mathsf{X}_n \mapsto \mathsf{Y}_n]\mathsf{T}_1$ を用いる．

このアルゴリズムは効率的で，現実には大体の場合にサイズに関する線形時間で終了する．しかしながら，型のサイズが指数爆発するような例では指数時間かかってしまう．

さらに，let-polymorphism のある言語で reference のような副作用を扱う場合は注意が必要である．

```
let r = ref (λx. x) in
(r:=(λx:Nat. succ x) ; (!r) true)  (* apply true to succ !? *)
```

の様なプログラムが前述のアルゴリズムだと型チェックを通ってしまう．これは r の type scheme が $\forall X.\mathsf{Ref}(X \to X)$ となり，最初の r の出現では $\mathsf{Ref}(\mathsf{Nat} \to \mathsf{Nat})$ に，二度目の r の出現では $\mathsf{Ref}(\mathsf{Bool} \to \mathsf{Bool})$ に instanciate されて型チェックを通ってしまうためである．

この問題は，評価規則と型付け規則が一致していないことが原因である．要するに，この章で変更した型付け規則では，let が出現した場合，即座に束縛の右辺を body に代入しているが，評価規則では，束縛の右辺が value に簡約された後に body に代入を行っているということである．

これを修正するために，

$$\mathsf{let}\ x{=}t_1\ \mathsf{in}\ t_2 \longrightarrow [x \mapsto t_1]t_2 \quad \text{(E-LET)}$$

とするのはあまり意味がない．先ほどのプログラムは

```
(ref (λx. x)):=(λx:Nat. succ x) ;
(!(ref (λx. x))) true
```

となり確かに安全であるが，直感と反する semantics となってしまう．

そうではなく，型付け規則の方を評価規則に合うように変更するのがより良い解決策である．これは value restriction と呼ばれる制約で，束縛の右辺が (syntax における)value であるときに限り，その型の自由変数を polymorphic に扱う ($\forall$ をつける)．

これにより，先ほどの r の型は $\mathsf{Ref}(X \to X)$ となり，最初の r で $X = \mathsf{Nat}$ という制約が生じるので，次の r で fail する．

また，value restriction によって表現力が制限されるが，現実的にはあまり問題の無い事が知られている．

22.8 Notes

略.