Agdaによる型推論器の定式化

門脇 香子[1,a)]　浅井 健一[1,b)]

1. Introduction

Agda [5] は Haskell に似た構文を持つ依存型を用いた定理証明支援機構及びプログラミング言語である．プログラミング言語と定理証明支援機構の両面を持っていることから，何かを実装しつつその正当性を証明することに適していると言われている．ここで，依存型は値に依存する型を作ることのできる型のことであり，例えば List n (n は自然数) という型は「長さ n のリスト」という意味を持つ型のことである．これを応用すると「Int 型を持つ項」などを表現することも可能になる．また， Agda は論理体系としてマルティンレフ型理論 [4] に基づいている．

2. 研究背景

依存型を使うと，型の合った項に対するインタプリタや部分評価器をバリアント型を使うことなく実装できる [1]．また，型チェックを行うプログラムで型がつかなかった場合になぜ型がつかなかったのかの理由を添えて返すプログラムが書ける [6] など，依存型の種々の面白い応用が知られるようになってきている．

しかし，ここで示されている型チェックは，束縛変数の型があらかじめユーザによって与えられていることを仮定している．これは，束縛変数の型を含めて型を推論しようとすると，型の unification が必要になるが，型の unification を Agda で表現するのは簡単ではないためである．

一方 McBride は，型変数の数を巧妙に管理することで構造に従った再帰を使って（つまり停止性が明らかな形で）型の unification を表現できることを示した [3]．これは，Agda で型推論を実装する土台が整ったことを意味している．

そこで本稿では Agda の依存型を用いて，停止性が保証された型推論器を構成する．Agda で依存型を用いて型推論器を実装することの利点は，ある型を持つ Term に対して Well-Typed または Well-Scoped な Term だけ考えて書くことができる点である．つまり，Raw な Term をもらってきたら直ちに型推論器にかけて Well-Typed な Term を出力できるということである．この特性はプログラム解析等，さまざまな場面に役立つものであり，例として部分評価器の束縛時解析などで，Static な項と Dynamic な項の判定を計算することができるなどの利点がもたらされると予想される．

3. Syntax and Type Definition

本稿の型推論器で用いる Syntax は以下の通りである．変数の出現位置に関して de Bruijn index を用いている．自由変数の数が n 個の Well-Scoped な項として以下を定義する．

[1] お茶の水女子大学大学院 人間文化創成科学研究科
[a)] kado@pllab.is.ocha.ac.jp
[b)] asai@is.ocha.ac.jp

```
data WellScopedTerm (n : ℕ) : Set where
  Var : Fin n → WellScopedTerm n
  Lam : WellScopedTerm (suc n) →
    WellScopedTerm n
  App : WellScopedTerm n →
    WellScopedTerm n →
      WellScopedTerm n
```

また，型定義は以下の通りである．それぞれ TVar は de Bruijn index に基づいた型変数であり，TInt は整数型であり，⇒ は関数型を表している．

```
data Type (n : ℕ) : Set where
  TVar : (x : Fin n) → Type n
  TInt : Type n
  _⇒_ : (s t : Type n) → Type n
```

4. Unification

本節では，型推論に必要な型の unification について述べる．本節の内容は McBride [3] が示した方法を Agda で実装し直したものである．型の unification は，書き換え可能なセルを使うと簡明に実装することができる [7]．しかし，Agda では書き換えは許されないのに加えて，停止性が保証されている必要がある．

そこで McBride は「型変数が具体化されるたびに，具体化されていない型変数の数がひとつ減る」という点に注目した．n 個の「具体化されていない型変数」を Fin n 型の数字（ここで Fin n は 0 から $n-1$ までの数字からなる有限集合の型を表す．）で表現し，その n を減らすことで停止性が明らかな形の unification を実現した．ひとたび型変数をこのような形で表現できたら，その後の unification は通常通りに進む．具体的には，型の中に型変数が出現するかどうかを調べる関数 check を実装し，それを使って最汎の単一化子 (most general unifier) を求める関数 mgu を実装する．

4.1 変数を薄める・濃縮する

この節では，型変数の表現について述べる．n 個の型変数を Fin n 型で表している状態で，その x 番目の位置に新たな型変数が挿入されると全体として型変数は $n+1$ 個になる．逆に n 個の型変数がある状態で，その x 番目の位置の型変数が（具体化されることで）削除されると型変数は $n-1$ 個になる．このように特定の場所に型変数を挿入したり削除したりする操作を McBride は thin, thick という関数で表現している．（直感的には thin は型変数が挿入されて全体が薄められる感じ，逆に thick は型変数が削除されて全体が濃縮される感じである．）thin x y は変数 y を x の位置で薄めるものである．つまり，x 未満の変数はそのまま返され，x 以上の変数は $+1$ されて返される．また，thin の結果が x になることはない．定義は以下のようになる．

```
thin : {n : ℕ} → Fin (suc n) → Fin n → Fin (suc n)
thin {n} zero y = suc y
thin {suc n} (suc x) zero = zero
thin {suc n} (suc x) (suc y) = suc (thin x y)
```

また，thin の partial inverse である thick についても説明をする．thick x y は変数 y を x の位置で濃縮するものである．x 未満の変数はそのまま返され，x より大きい変数は -1 される．x と y が同じ場合は濃縮できないので nothing が返る．定義は以下のようになる．

```
thick : {n : ℕ} → (x y : Fin (suc n))
  → Maybe (Fin n)
thick {n} zero zero = nothing
thick {n} zero (suc y) = just y
thick {zero} (suc ()) zero
thick {suc n} (suc x) zero = just zero
thick {zero} (suc ()) (suc y)
thick {suc n} (suc x) (suc y)
      with thick {n} x y
... | just x' = just (suc x')
... | nothing = nothing
```

本論文では，これらのうち thick のみを使う．thin は thick の性質などを証明するときに役に立つ [3].

thick を使うと、例えば次のような関数を実装することができる．t for x は、型変数 x を t にするような unifier を表す．x に t を代入したら型変数 x は不要となるため，x 以外の変数については

x で thick された型変数が返る．

```
_for_ : {n : ℕ} → (t : Type n)
    → (x : Fin (suc n)) → Fin (suc n) → Type n
_for_ t x y with thick x y
... | just y' = TVar y'
... | nothing = t
```

4.2 check

check x t は x 番の型変数が型 t の中に現れるかを thick 関数を用いてチェックする関数である．現れていたら nothing を返す．一方，現れていなかったら，型 t で使われている型変数から x を取り除いて，ひとつ少ない型変数を使って型 t を表現できるはずである．check x t は，その型を結果として（あるいは x が t に現れなかったという証拠として）返す．具体的なコードは以下のようになる．

```
check : {n : ℕ} → Fin (suc n)
    → Type (suc n) → Maybe (Type n)
check x (TVar y) with thick x y
... | just y' = just (TVar y')
... | nothing = nothing
    - x が現れた（x = y だった）
check x TInt = just TInt
check x (s ⇒ t) with check x s | check x t
... | just s' | just t' = just (s' ⇒ t')
... | just s' | nothing = nothing
... | nothing | just t' = nothing
... | nothing | nothing = nothing
```

型 t が型変数だった場合，thick を使ってそれが x と等しいかどうかを確認する．等しい場合は型 t に x が現れたので nothing を返す．等しくなければ，thick した結果を証拠として返す．型 t が整数型だったら x は現れていないのでそのまま返す．型 t が関数型だったら，左右両方の型について再帰し，どちらにも x が現れていないかを確認する．以上である型変数についての出現の有無を調べることができるようになった．

4.3 Most general unifier

代入を表すデータ構造として、AList を定義する。AList m n（m, n は自然数）は「m 個の型変数を持つ型」を「n 個の型変数を持つ型」に変換するような代入の型である。

```
data AList : ℕ → ℕ → Set where
  anil : {m : ℕ} → AList m m
  _asnoc_/_ : {m : ℕ} {n : ℕ} →
    ( σ : AList m n) → (t' : Type m) →
      (x : Fin (suc m)) → AList (suc m) n
```

anil は、何も変化させない代入、σ asnoc t / x は、まず型変数 x に t を代入してから $sigma$ を行うような代入である。

次に，flexFlex と flexRigid という関数を実装する．flexFlex は型変数 x と y を unify する代入を返すものであり，次のように定義される．

```
flexFlex : {m : ℕ} → (x y : Fin m) →
  Σ [ n ∈ ℕ ] AList m n
flexFlex {zero} () y
flexFlex {suc m} x y with thick x y
... | nothing = (suc m , anil)
... | just y' = (m , anil asnoc (TVar y') / x)
```

x と y が同じかどうかを thick で調べ，同じだったら空の代入を，違ったら x を y にする代入を返している．型変数の数を把握しておくため，いずれの場合も代入後の型変数の数を合わせて返している．

flexRigid は型変数 x と型 t を unify する代入を返すものである．

```
flexRigid : {m : ℕ} → (x : Fin m) →
  (t : Type m) → Maybe (Σ [ n ∈ ℕ ] AList m n)
flexRigid {zero} () t
flexRigid {suc m} x t with check x t
... | nothing = nothing
... | just t' = just (m , anil asnoc t' / x)
```

基本的には x を t にする代入を返すが，この代入は x が t の中に現れていたら行うことができない．それを check を使って調べている．

以上を使って mgu を求める関数を実装する．

```
mgu : {m : ℕ} → (s t : Type m) →
  Maybe (Σ [ n ∈ ℕ ] AList m n)
mgu {m} s t = amgu {m} s t ( m , anil )
```

mgu s t は，型 s と型 t を同じにするような代入を返す．ここで s と t は型変数を m 個持つような型である．mgu は結果として $(n,$ 代入$)$ というペアを返す．ここで n は unification を行った後の型変数の数であり，代入は m 個の型変数を n 個まで減らすようなもの（$m-n$ 個の型変数に対して代入を行うようなもの）である．この関数はアキュムレータを使った amgu を使って定義される．

```
amgu : {m : ℕ} → (s t : Type m) →
    (acc : Σ [ n ∈ ℕ ] AList m n) →
       Maybe (Σ [ n ∈ ℕ ] AList m n)
amgu TInt     TInt      acc      = just acc
amgu TInt     (t ⇒ t₁)  acc      = nothing
amgu (s ⇒ s₁) TInt      acc      = nothing
amgu (s ⇒ s₁) (t ⇒ t₁)  acc      with amgu s t acc
...    | just acc'
           = amgu s₁ t₁ acc'
...    | nothing = nothing
amgu (TVar x) (TVar y) (s , anil) = just (flexFlex x y)
amgu (TVar x) t        (s , anil) = flexRigid x t
amgu t        (TVar x) (s , anil) = flexRigid x t
amgu {suc m} s t (n , σ asnoc r / z)
    with amgu {m} ((r for z) ◁ s) ((r for z) ◁ t) (n , σ)
... | just (n' , σ') = just (n' , σ' asnoc r / z)
... | nothing = nothing
```

これは型 s と t の構造に従って素直に unification を行う関数である．片方，または両方が型変数だった場合は flexRigid, flexFlex を使って unification を行う．最後のケースに出てくる $(r$ for $z) \triangleleft s$ は，型 s の中の型変数全てについて $(r$ for $z)$ を施した型を示す．

最後に，後述の型推論器のため mgu を unify という名前で再定義しておく．これで unify $t1$ $t2$ は型変数が m 個であるような型 $t1$ と $t2$ を unify する代入を返す関数として実装ができた．

```
unify : {m : ℕ} → Type m → Type m →
    Maybe (Σ [ n ∈ ℕ ] AList m n)
unify {m} t1 t2 =
    mgu {m} t1 t2
```

5. Type Inference

本稿の型推論では，algorithm W [2] の単相の部分を使用する．具体的には，型環境と Well-Scoped な項 s をもらってきたら，s の型と必要な代入を Maybe モナドに包んで返す関数 InferW を実装する．

```
inferW : {m n : ℕ} →    (Γ : Cxt {m} n) →
    (s : WellScopedTerm n) →
       Maybe (Σ [ m' ∈ ℕ ]
          AList (m + count s) m' × Type m')
```

ここで count s は WellScoped な項 s の中に含まれる Lam と App のノード数を数える関数である．これらのノード全てに対して，それぞれ新しい型変数が与えられ，型推論が進む．

```
count : {n : ℕ} → (s : WellScopedTerm n) → ℕ
count (Var x) = zero
count (Lam s) = suc (count s)
count (App s1 s2) =
    count s1 + suc (count s2)
```

s は最大で n 個の自由変数を持つ項，Γ はその n この自由変数の型を与える型環境，型環境中の各型は最大で m 個の型変数を含むものであり，この状態で inferW を呼ぶと，途中で count s 個の型変数が新たに割り当てられて，最終的に s に型がつかないなら nothing，s に型がつくなら just (m', σ, τ) が返ってくる．ここで τ は s の型で最大 m' 個の型変数を含む．また σ は，もとの $m+$ count s 個の型変数を m' 個まで落とす代入である．ここで返って来た結果は $\sigma\,\Gamma\ \sigma\Gamma \vdash e : \tau$ を満たすが，その証明は本稿では行わない．

5.1 Variable

変数の場合は，単に型環境の中の型を返す．代入は不要で，型変数の数にも変化はない．

```
inferW {m} Γ (Var x)
    rewrite +-right-identity m =
                    just (m , anil , lookup x Γ)
```

5.2 Lambda

次に，Lambda の場合である．

```
inferW {m} Γ (Lam s)
  with inferW
  (TVar (fromℕ m) :: liftCxt 1 Γ) s
    - TVar (fromℕ m) が引数の型
... | nothing = nothing
  - s に型がつかなかった
... | just (m' , σ , t) rewrite +-suc m (count s) =
  just (m' , σ , substType σ
    (liftType2 (count s) (TVar (fromℕ m))) ⇒ t)
      - TVar (fromℕ m) ⇒ t が Lam s の型
```

Lambda の場合は，まず body 部分の型推論を行う．その際，引数の型として型変数 m を用いる．これまで型環境 Γ には 0 から $m-1$ までの型変数しか使われていないはずなので，m は新しい型変数となる．ここで，liftCxt 1 Γ は Γ 中の型変数の数を 1 増やす関数である．body 部分に型がつかなかったら，全体としても型がつかない．

body 部分に型が付いたら「引数の型⇒ body の型」が全体の型である．ただし，その際，count s 個の型変数を新たに割り振っているので，その分だけ使っている型変数の数を liftType2 を使って持ち上げている．また，引数の型変数は body を型推論中に具体化されているかも知れないので，代入 σ を施している．

5.3 Application

最後に関数適用における実装である．App $s1$ $s2$ とう式の型推論は，普通通りまず $s1$ と $s2$ の型推論を行い，次に $s1$ の型が「$s2$ の型⇒ β 」になっていることを確認する形で行う．より詳しくは以下のようになる．

(1) はじめに，再帰呼び出し inferW $\{m\}\,\Gamma\,s1$ で $\sigma_1\Gamma \vdash s1 : t1$ なる σ_1, $t1$ が得られる．

(2) 次に $s2$ についての再帰をするが，この時点では環境 $\sigma_1\Gamma$ の元で $s1$ の型が inferW $\{m_1\}$ Γ $s2$ となるので，$\sigma_2\,(\sigma_1\,\Gamma) \vdash s2 : t2$ なる σ_2, $t2$ が得られる．ここで得られた σ_2 を 1. で得られた型判定に適用すると，$\sigma_2\,(\sigma_1\,\Gamma) \vdash s1 : \sigma_2(t1)$ となる．

(3) 次に，$\sigma_2(t1) = t2 \to \beta$（ここで β は新しい型変数）となる unify σ_3 を求める．そうすると，$\sigma_3(\sigma_2(t1)) = \sigma_3(t2 \to \beta)$ となり，σ_3 を 2. で得られているふたつの型判定に適用すると，

$$\sigma_3\ (\sigma_2\ (\sigma_1\,\Gamma)) \vdash s1 : \sigma_3\ (\sigma_2\ (t1))$$
$$\sigma_3(\sigma_2(\sigma_1\,\Gamma)) \vdash s2 : \sigma_3(t2)$$

が得られる．ここから関数適用の型規則を使うと，$\sigma_3(\sigma_2(\sigma_1\,\Gamma)) \vdash$ App $s1$ $s2 : \sigma_3(\beta)$ が得られる．

(4) よって，App $s1$ $s2$ が返すものは just $(m_2, \sigma_3{+}(\sigma_2{+}\sigma_1), \sigma_3(\beta))$ となる．

以上をコードにしたのが以下である．

```
inferW {m} Γ (App s1 s2)
      with inferW Γ s1
... | nothing = nothing
  - s1 に型がつかなかった
... | just (m1 , σ 1 , t1)
  - s1 の型が t1
      with inferW (substCxt σ 1
      (liftCxt2 (count s1) Γ)) s2
... | nothing = nothing
  - s2 に型がつかなかった
... | just (m2 , σ 2 , t2)
  - s2 の型が t2。
  - m2 を App s1 s2 の返り値の型に割り当てる
      with unify (liftType 1 (substType σ 2
        (liftType2 (count s2) t1)))
        (liftType 1 t2 ⇒ TVar (fromℕ m2))
... | nothing = nothing
  - unify できなかった
... | just (m3 , σ 3)
      rewrite sym (+-assoc m
        (count s1) (suc (count s2)))
      with liftAList2 1 σ 2
... | σ 2' rewrite sym (+-suc m1 (count s2))
      with liftAList (suc (count s2)) σ 1
... | σ 1' =
      just (m3 , σ 3 +!+ (σ 2' +!+ σ 1') ,
      substType σ 3 (TVar (fromℕ m2)))
```

ここで，+!+ はふたつの代入を結合する関数である．

このコードでは，上記に加えてさらに型変数の

数を細かく保持している．型変数の数は，型推論が始まる時点では m である．$s1$ を型推論する時点でそれは m+count $s1$ まで増えるが，型推論中に型変数の具体化が起こり $m1$ となる．次に，$s2$ を型推論する時点で $m1$+count $s2$ まで増えるがやはり型推論中に型変数の具体化が起こり $m2$ となる．さらに，β を割り当てるので $m2+1$ 個に増え，そこから最後の unification で $m3$ 個に減ることになる．このように inferW では型変数の移り変わりを全てとらえることで型推論を行っている．

6. まとめ

本研究では，Agda を用いて Simple-Typed Lambda Calculus の unification の機構を McBride の手法 [3] により克服し，型推論器の実装を行った．今後は推論が正しいことのを証明を含めた実装を行いたい．具体的には，WellScopedTerm をもらったら WellTypedTerm（きちんと型付けされた項）を返す実装である．WellTypedTerm の定義は以下のようになる．

```
data WellTypedTerm {m n : ℕ} (Γ : Cxt n) :
    Type m → Set where
    Var : (x : Fin n) → WellTypedTerm Γ (lookup x Γ)
    Lam : (t : Type m) → {t' : Type m} →
        WellTypedTerm (t :: Γ) t' →
            WellTypedTerm Γ (t ⇒ t')
    App : {t t' : Type m} →
        WellTypedTerm Γ (t ⇒ t') →
            WellTypedTerm Γ t → WellTypedTerm Γ t'
```

また，Syntax の拡張としては，多相の Let 文を含めた形についても実装と証明を行っていきたい．

参考文献

[1] Asai, K., L. Fennell, P. Thiemann, and Y. Zhang “A Type Theoretic Specification of Partial Evaluation,” *Proceedings of the 2014 Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pp. 57–68 (September 2014).

[2] Damas, L. and R. Milner “Principal type-schemes for functional programs,” *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pp. 207–212 (January 1982).

[3] McBride, C. “First-order unification by structural recursion,” *Journal of Functional Programming*, Vol. 13, No. 6, pp. 1061–1075, Cambridge University Press (November 2003).

[4] Nordström, B., K. Petersson, and J. M. Smith *Programming in Martin-Löf's Type Theory*, Oxford University Press (1990).

[5] Norell, U. “Dependently typed programming in Agda,” In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming (LNCS 5832)*, pp. 230–266 (2009).

[6] Norell, U. “Interactive Programming with Dependent Types,” *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, invited talk, pp. 1–2 (September 2013).

[7] 住井 英二郎「MinCaml コンパイラ」コンピュータソフトウェア Vol. 25, No. 2, pp. 28–38 (2008).