Agdaによる型推論器の定式化

門脇 香子, 浅井 健一

お茶の水女子大学大学院 人間文化創成科学研究科

kado@pllab.is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要 Agdaはマルティンレフの型理論に基づいた定理証明支援器であり，その一方で依存型をもつ関数型プログラミング言語でもある．Agdaでは依存型(Dependant Types)を用いることができるのも大きな特徴である．また，定理証明支援器とプログラミング言語両方の特徴をもちあわせていることから，何かを実装しつつ証明するのに適している．そこで本稿では，停止性が保証された型推論器をAgdaにより構成する．なかでも，Unificationの実装では， McBrideの手法を採用する．これ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ものである．本稿ではTermの型変数の数に注目しつつ，主にUnificationの部分とApplicationの実装にフォーカスを当てながら解説をしていく．

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．ここで，依存型は値に依存する型のことであり，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．

1.1 研究背景

依存型を使うと，インタプリタや部分評価器を作る際，対象言語の型を項の定義に埋め込むことができる．これを使うと，型付き言語のインタプリタや部分評価器をタグを使うことなく実装することができる [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，このunificationの理論をもとにして，型推論器全体を実装する土台が整ったことを意味している．

1.2 研究概要

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とさらに加えて正当性の証明がついた型推論器を完全にpure functionalに実装した．Agdaで依存型を用いて型推論器を実装することの利点は，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またはwell-scoped

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．本稿の貢献は，McBride が示した unification の機構 [3] をもとにして，停止性と正当性の証明がついた型推論器を実装したところである．

以下，2 節では，型推論器で用いる syntax と型定義を示す．3 節では，McBride による unification の機構とその実装について述べる．4 節では，停止性が保証された型推論器の実装について述べる．5 節では，加えて正当性の証明がついた型推論器の実装について述べる．6 節では，5 節で必要となる正当性の証明がついた unification の実装について述べる．7 節で本稿のまとめを行う．なお，ここで示した型推論器の実装は以下の URL で公開している．

`https://github.com/KDXU/InferAgda`

2 Syntax と 型定義

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．

```
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 は型変数であり，TInt は整数型であり，⇒ は関数型を表している．今回の実装の場合は，ラムダ項（well-scoped term）が TInt 型の定数を含まないので，TInt を用いずに arrow 型だけで表現することもできるが，今回は形式的に TInt 型を付け加えている．

```
data Type (n : ℕ) : Set where
  TVar : (x : Fin n) → Type n - 型変数 (de Bruijn index)
  TInt : Type n - base case の型
  _⇒_ : (s t : Type n) → Type n - arrow 型
```

3 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．

3.1 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． 場合分けをすると以下のようになる．

$$thin\ x\ y = \begin{cases} y & (y < x) \\ suc\ y & (otherwise) \end{cases}$$

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．場合分けをすると以下のようになる．

$$thick\ x\ (suc\ y) = \begin{cases} just\ (suc\ y) & (y < x) \\ nothing & (y = x) \\ just\ y & (otherwise) \end{cases}$$

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 - x = y だった
thick {n} zero (suc y) = just y - 濃縮する
thick {zero} (suc ()) zero
thick {suc n} (suc x) zero = just zero - x 未満なのでそのまま
thick {zero} (suc ()) (suc y)
thick {suc n} (suc x) (suc y) with thick {n} x y
... | just x' = just (suc x')
... | nothing = nothing - x = y だった
```

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 にするような変数一つに対する代入を表す．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
```

3.2 check

check x t は x 番の型変数が型 t の中に現れるかを thick 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．

3.3 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 $m \geq n$ であるような自然数）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 - 何もしない代入
  _asnoc_/_ : {m : ℕ} {n : ℕ} → (σ : AList m n) → (t' : Type m) →
    (x : Fin (suc m)) → AList (suc m) n - x を t' にした上で、さらにσを行う代入
```

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 σ 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 を単一化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) - x = y だった。代入の必要なし
... | just y' = (m , anil asnoc (TVar y') / x) - TVar y' for x を返す
```

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 を単一化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 - x が t に現れていた
... | just t' = just (m , anil asnoc t' / x) - t' for x を返す
```

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 most general unifier を求める関数 mgu 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 を行う関数であり，また m についての構造的帰納法を用いている．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 $t1$ $t2$ を型変数が m 個であるような型 $t1$ と $t2$ を単一化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
```

4 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 infer を実装する．以下の infer Γ s は Γ のもとで s を型推論する関数である．

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

ここで count s は well-scoped 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．その数が、新たに導入される型変数の数になり，型推論が進む．具体的には，もとの型変数の数が m だったとき、推論結果として (m', 代入, 型) を返す。(ここで m' は返って来た型の中に含まれる型変数の数) ここで，あらかじめ s の中の Lam, App ノードに型変数をひとつ割り振ったとすると，型変数の合計はもともと m + count s となり，返ってくる代入は，型変数の数を m + count s から m' に落とすものになる．

```
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)
```

infer の引数 s は最大で n 個の自由変数を持つ項， Γ はその n 個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る．この状態で infer 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 count $s + m$ 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 $\sigma\Gamma \vdash s : \tau$ を満たす．また，補助関数として liftCxt, liftAList, liftType などの「型変数の数を持ち上げる」関数を利用している．これらの関数の型は以下となる．

```
- liftCxt m' Γ :  Γ の中の型変数の数を m' だけ増やす
liftCxt : (m' : ℕ) → {m n : ℕ} → Cxt {m} n → Cxt {m' + m} n
- liftAList n lst :  lst の中の型変数の数を n だけ増やす
liftAList : {m m' : ℕ} → (n : ℕ) → AList m m' → AList (n + m) (n + m')
- liftType m' t :  t の中の型変数の数を m' だけ増やす
liftType : (m' : ℕ) → {m : ℕ} → Type m → Type (m' + m)
```

これらはいずれも簡単に定義することができる．

また，substCxt や substType など Cxt や Type に対する代入も補助関数として用いている．

```
substType : {m m' : ℕ} → AList m m' → Type m → Type m'
- substCxt σ Γ :  Γ に σ を適用した型環境を返す
substCxt : {m m' n : ℕ} → AList m m' → Cxt {m} n → Cxt {m'} n
```

4.1 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．

```
infer {m} Γ (Var x) = just (m , anil , lookup x Γ)
```

4.2 Lambda

次に， lambda 抽象の場合について解説する．

```
infer {m} Γ (Lam s)
          with infer (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) =
    just (m' , σ' , tx ⇒ t) - TVar (fromℕ m) ⇒ t が Lam s の型
        where
            σ' : AList (suc (count s + m)) m'
            σ' = subst (λ m → AList m m') (+-suc (count s) m) σ
            tx : Type m'
            tx = substType σ (liftType (count s) (TVar (fromℕ m)))
```

ここで，+-suc の型定義は次のようになる．

```
+-suc : ∀ m n → m + suc n ≡ suc (m + n)
```

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 s に型が付き，$(m'\ ,\ \sigma\ ,\ t)$ が返ってきたら，これは $\sigma\,(m :: \Gamma) \vdash\ s : t$ となることを示している．従ってここから $\sigma\,\Gamma \vdash (\mathsf{Lam}\ s) : \sigma\ m \Rightarrow t$ を導けるので全体の型はほぼ $(m'\ ,\ \sigma\ ,\ \sigma\ m \Rightarrow t)$ となる．ただし，s の型推論中に count s 個の型変数を新たに割り振っているので，その分だけ型変数を増やしているのが上記の tx である．また，σ は AList (count s + suc m) m' という型を持っているが，全体の結果として欲しい代入の型は，+-suc の型からも AList (suc (count s) + m) m' となる．そこで subst を使って型変換を行い σ' を結果に返している．

4.3 Application

最後に関数適用における実装である．App $s1$ $s2$ という式の型推論は，元のアルゴリズムの通りまず $s1$ 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 infer $\{m\}\,\Gamma\,s1$ 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 $s2$ の型が infer $\{m_1\}$ $\sigma_1\Gamma$ $s2$ 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 unifier 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 App $(m_2, \sigma_3$+!+$(\sigma_2$+!+$\sigma_1), \sigma_3\,(\beta))$ となる．ここで，+!+ はふたつの代入を結合する関数である．

以上をコードにしたのが以下である．

```
infer {m} Γ (App s1 s2)
        with infer 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 infer (substCxt σ 1 (liftCxt (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 (liftType (count s2) t1))) - t1
           (liftType 1 t2 ⇒ TVar (fromℕ m2)) - t2 ⇒ TVar (fromℕ m2)
... | nothing = nothing - unify できなかった
... | just (m3 , σ 3) =
```

```
  just (m3 , σ 3 +!+ (σ 2' +!+ σ 1') , substType σ 3 (TVar (fromℕ m2)))
  where
    σ 1' : AList (count s1 + suc (count s2) + m) (suc (count s2 + m1))
    σ 1' rewrite +-comm (count s1) (suc (count s2)) | +-assoc (count s2) (count s1) m
      = liftAList (suc (count s2)) σ 1
    σ 2' : AList (suc (count s2 + m1)) (suc m2)
    σ 2' = liftAList 1 σ 2
```

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 count $s1 + m$ 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 count $s2 + m1$ 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 $1 + m2$ 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 infer 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．

5 Well-Typed な型推論

前節の infer を使うと項の型を推論できるようになったが，infer は推論結果として代入と型を返すのみで，それらが本当に正しい推論結果であることは保証していない．本節では，推論結果の正当性が保証された型推論について述べる．本節で述べる型推論では，推論結果は単に代入と型のみを返すのではなく，同時に well-typed な項を返す．これが「推論結果が正しい」ことの証明となる．全体として，型推論は well-scoped な項を well-typed な項へと変換する関数となる．well-typed な項の定義は以下の通りである．

```
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'
```

これは型環境 Γ において自由変数の数が n 個，型が t であるような well-typed な項を表している．これは即ち term が書けた時点で well-typed であることが保証されるもので， simply-typed の型規則をそのまま埋め込んでいる．

型推論器の定義部分の実装は以下の通りである．

```
infer2 : {m n : ℕ} → (Γ : Cxt {m} n) → (s : WellScopedTerm n) →
  Maybe (Σ [ m' ∈ ℕ ]
    Σ [ σ ∈ AList (count s + m) m' ]
    Σ [ τ ∈ Type m' ]
    WellTypedTerm (substCxt σ (liftCxt (count s) Γ)) τ)
```

これは，型環境と well-scoped な項 s をもらってきたら，型変数の数 m' と s の型 τ と必要な代入 σ と型付けされた well-typed な項を Maybe モナドに包んで返す関数である．型推論のアルゴリズムは，well-scoped term の場合と同じく，algorithm W [2] の単相の部分を使用する．

5.1 Variable

はじめに， Term が変数の場合である．

```
infer2 {m} Γ (Var x)    = just (m , anil , lookup x Γ , VarX)
  where
    VarX : WellTypedTerm (substCxt anil (liftCxt 0 Γ)) (lookup x Γ)
    VarX rewrite substCxtAnil Γ | liftCxtZero Γ = Var x
    - Var x :  WellTypedTerm Γ (lookup x Γ)
```

well-scoped term の場合と同じく，単に型環境の中の型を返す．代入は不要で型変数の数に変化はない．well-typed な項は Var x 自身となるが， Var x の型 (上記コメントに示した) と必要な型 (上記 VarX の型) がわずかに異なるので，それらを以下の補助関数を使って結んでいる．

```
- substCxt anil Γ は Γ と同じ
substCxtAnil : {m n : ℕ} → (Γ : Cxt {m} n) → substCxt anil Γ ≡ Γ
- liftCxtZero :  Cxt Γ を 0 持ち上げたものは Γ と同じ
liftCxtZero : {m n : ℕ} → (Γ : Cxt {m} n) → liftCxt 0 Γ ≡ Γ
```

5.2 Lambda

つぎに term がラムダ式 Lam s の形になっている場合である．こちらも well-scoped term の場合と同様に，まず body 部分の型推論を行い，body 部分に型が付いた場合のみ型付けされ,「引数の型⇒ body の型」が全体の型となる．コードは以下のようになる．

```
infer2 {m} {n} Γ (Lam s) with infer2 (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 , w) = just (m' , σ' , tx ⇒ t , LamW)
  where
    σ' : AList (suc (count s + m)) m'
    σ' = subst (λ m → AList m m') (+-suc (count s) m) σ
    tx : Type m'
    tx = substType σ (liftType (count s) (TVar (fromℕ m)))
    LamW : WellTypedTerm (substCxt σ' (liftCxt (count (Lam s)) Γ)) (tx ⇒ t)
    LamW = Lam tx w'
      where
        w' : WellTypedTerm (tx :: (substCxt σ' (liftCxt (count (Lam s)) Γ))) t
        w' = subst (λ l → WellTypedTerm (tx :: l) t) eq w
          where
            eq : substCxt σ (liftCxt (count s) (liftCxt 1 Γ)) ≡ substCxt σ' (liftCxt (count (Lam s)) Γ)
            - eq = ...
```

このコードは well-scoped な場合と比べて，再帰呼び出しの結果として well-typed な項を受け取っていることと，最後に well-typed な LamW を返している点のみが異なっている．body 部分 s を型推論した結果，well-typed な w が得られた場合，最終的に返したい well-typed な項はほぼ Lam tx w となる．しかし，再帰呼び出しで得られた w の型は

WellTypedTerm (substCxt σ (liftCxt (count s)(TVar (fromℕ m) :: liftCxt 1 Γ))) t

であるのに対して，実際に必要なものは

WellTypedTerm (tx :: substCxt σ (liftCxt (count (Lam s)) Γ)) t

である．w の型は展開すると

WellTypedTerm (tx :: substCxt σ (liftCxt (count s) (liftCxt 1 Γ))) t

となるので，w の型の型変数の数を「1 加えてから count s 加えている」部分を「一度に count (Lam s) つまり count $(1 + s)$ を加える」ように変更すれば良い．これを行っているのが w' の定義である．この中に出てくる eq が（コードは省略しているが）上のふたつが等しいことを示している．

5.3 Application

最後に term が関数適用 App *s1 s2* の形になっている場合である．こちらも well-scoped term の時と同様に，App *s1 s2* において，$s1$ の型推論によって返された代入 σ_1 を Γ に加え環境 $\sigma_1\,\Gamma$ の元での $s2$ の型推論が成功した場合にのみ，$s2$ の型⇒ β が s1 と等しくなっているかどうかを unify 関数によって判定する．コードは以下のようになる．

```
infer2 {m} {n} Γ (App s1 s2) with infer2 Γ s1
... | nothing = nothing - s1 に型がつかなかった
... | just (m1 , σ 1 , t1 , w1)

    with infer2 (substCxt σ 1 (liftCxt (count s1) Γ)) s2
... | nothing = nothing - s2 に型がつかなかった
... | just (m2 , σ 2 , t2 , w2) - m2 を App s1 s2 の返り値の型に割り当てる

    with unify2 (liftType 1 (substType σ 2 (liftType (count s2) t1))) (liftType 1 t2 ⇒ TVar (fromℕ m2))
... | nothing = nothing - unify できなかった
... | just (m3 , σ 3 , eq') =
    just (m3 , σ 3 +!+ (σ 2' +!+ σ 1') , substType σ 3 (TVar (fromℕ m2)) , AppW1W2)
    where
      σ 1' : AList (count s1 + suc (count s2) + m) (suc (count s2 + m1))
      σ 1' rewrite +-comm (count s1) (suc (count s2)) | +-assoc (count s2) (count s1) m
        = liftAList (suc (count s2)) σ 1
      σ 2' : AList (suc (count s2 + m1)) (suc m2)
      σ 2' = liftAList 1 σ 2
      AppW1W2 : WellTypedTerm (substCxt (σ 3 +!+ (σ 2' +!+ σ 1')) (liftCxt (count (App s1 s2)) Γ))
        (substType σ 3 (TVar (fromℕ m2)))
      AppW1W2 = App w1' w2'
        where
    w1' : WellTypedTerm (substCxt (σ 3 +!+ (σ 2' +!+ σ 1')) (liftCxt (count (App s1 s2)) Γ))
      (substType σ 3 (liftType 1 t2 ⇒ TVar (fromℕ m2)))
    - w1 = ...
    w2' : WellTypedTerm (substCxt (σ 3 +!+ (σ 2' +!+ σ 1')) (liftCxt (count (App s1 s2)) Γ))
      (substType σ 3 (liftType 1 t2))
    - w2' = ...
```

このコードも前節と同様，well-scoped な場合と比べて，再帰呼び出しの結果を $w1, w2$ に受け取っていることと，最後に well-typed な AppW1W2 を返している点のみが異なっている．この AppW1W2 は，ほぼ App *w1 w2* となる．しかし前節と同様，型の調整を行う必要がある．より簡単な $w2$ のケースから説明する．

$w2$ の型は

```
WellTypedTerm (substCxt σ 2 (liftCxt (count s2) (substCxt σ 1 (liftCxt (count s1) Γ)))) t2
```

であるが，$w2'$ として欲しいものの型は上のコードに示した通りである．$w2$ が得られたあとに App *s1 s2* の返り値の型変数をひとつ割り当て，さらに unify を行って σ *3* が得られているので，$w2$ に対して型変数をひとつ増やし σ *3* を施した substWTerm σ *3* (liftWTerm 1 *w2*) の型を考えると

```
WellTypedTerm (substCxt σ 3 (liftCxt 1 (substCxt σ 2 (liftCxt (count s2) (substCxt σ 1 (liftCxt
```

(count $s1$) Γ))))) (substType σ 3 (liftType 1 $t2$))

となる．これと欲しい $w2'$ を見比べると「環境部分」が全く同じではないことがわかる．
　環境部分が同じであることを示すためには，Γ に対して

- (count $s1$) だけ型変数を増やし，σ *1* を施し，
- (count $s2$) だけ型変数を増やし，σ *2* を施し，
- 1 だけ型変数を増やし，σ 3 を施す

のが

- 一度に (count (App *s1 s2*)) だけ型変数を増やし，(σ *3* +!+ (σ *2'* +!+ σ *1'*)) を施す

のと同じことを示す必要がある．count (App *s1 s2*) は count (1 + count $s1$ + count $s2$) になるので，直感的にはこれらが等しいことは理解できる．これを実際に示すには，以下の 3 つを示すことがキーとなる．

- 型変数を増やしてから代入を施すのと，代入を施してから型変数を増やすのは同じである．
- 型変数を 2 度に分けて増やすのと，和を 1 度に増やすのは同じである．
- ふたつの代入を順に施すのは，代入を結合したものを 1 度に施すのと同じである．

これらを使うと，証明はかなり長くなるがふたつの環境が等しいことを示すことができる．
　一方，$w1$ のケースでは，次のようになる．$w1$ の型は

WellTypedTerm (substCxt σ *1* (liftCxt (count $s1$) Γ)) $t1$

であるが，$w1'$ として欲しいものの型は上のコードに示した通りである．$w1$ が得られたあとに一連の操作を行って得られた

substWTerm σ 3 (liftWTerm 1 (substWTerm σ 2 (liftWTerm (count $s2$) $w1$)))

を考えると，その型は

WellTypedTerm (substCxt σ 3 (liftCxt 1 (substCxt σ 2 (liftCxt (count $s2$) (substCxt σ 1 (liftCxt (count $s1$) Γ)))))) (substType σ 3 (liftType 1 (substType σ 2 (liftType (count $s2$) $t1$))))

となる．これと欲しい *w1'* を見比べると「環境部分」については，*w2* の場合と同じなので，同様に示すことができる．
　一方「型部分」については，より複雑なことが起こっている．等しいことを示したいふたつの型は

substType σ 3 (liftType 1 (substType σ 2 (liftType (count $s2$) $t1$)))

と

substType σ 3 (liftType 1 $t2$ ⇒ TVar (fromℕ $m2$))

であるが，これらは一見すると全く違う形をしている．これらが等しいのは，両者が unify によって同じ型になっているはずだからである．しかし，4.3 節で紹介した unify は，単に代入をひとつ返して

いるだけであり，その代入が実際にふたつの型を単一化することを保証はしていない．逆に，unify2 がふたつの型を単一化することを保証してくれるなら，上記のふたつの型は等しくなり，infer2 を完成することができる．これが，上記の infer 関数は unify ではなく unify2 を使っている理由である．unify2 は，代入に加えて，確かにその代入がふたつの型を単一化するという証明 eq' を一緒に返す．これを使うと，$w2$ と $w2'$ を結ぶことができ，infer 関数を完成することができる．

unify2 については，次節で述べる．

6 unify することが証明されている unify 関数

前に述べた well-scoped な項の場合だと unify は「何か代入を返すもの」と定義されているのみで「返って来た代入が確かにふたつの型を単一化するか」は保証されていない．well-typed な型推論器を実装するにおいて必要なのは，以前の unify を書き換えて，ふたつの型が確かに等しいことを保証するような関数である．

unify を書き換えるにあたり，flexFlex，flexRigid など変換前後における型の代入が等しいことの証明を添えて返すように変更する必要がある．以下に新しい flexFlex2，flexRigid2 の型定義を示す．flexFlex2 は 2 つの型変数を単一化する代入 σ に加えて，σ が確かに与えられた 2 つの型変数を単一化するという証明を返す．flexRigid2 も同様である．

```
flexFlex2 : {m : ℕ} → (x y : Fin m) →
  (Σ [ n ∈ ℕ ] Σ [ σ ∈ AList m n ] substType σ (TVar x) ≡ substType σ (TVar y))
flexRigid2 : {m : ℕ} → (x : Fin m) → (t : Type m) →
  Maybe (Σ [ n ∈ ℕ ] Σ [ σ ∈ AList m n ] substType σ (TVar x) ≡ substType σ t)
```

これを利用して，新しい amgu と mgu を定義する．以下の amgu2 が accumulator のついた unification の機構である．ここでは返り値として substType したものが等しいという証明に加えて，さらに amgu2 を定義する際の帰納法の仮定に必要となる返って来た σ が acc の拡張になっているという証明を加えている．

```
amgu2 : {m : ℕ} → (s t : Type m) → (acc : Σ [ n ∈ ℕ ] AList m n) →
  Maybe (Σ [ n ∈ ℕ ] Σ [ σ ∈ AList m n ] (n , σ) extends acc × substType σ s ≡ substType σ t)
```

以下の mgu2 が証明付きの unify である．これがあれば，infer 関数において unify の結果として substType したふたつの型が等しくなるという結果を使えるようになる．unify2 $t1$ $t2$ は 型変数が m 個であるような型 $t1$ と $t2$ を単一化する代入を返すものである．

```
mgu2 : {m : ℕ} → (s t : Type m) →
  Maybe (Σ [ n ∈ ℕ ] Σ [ σ ∈ AList m n ] substType σ s ≡ substType σ t)

unify2 : {m : ℕ} → (s t : Type m) →
  Maybe (Σ [ n ∈ ℕ ] Σ [ σ ∈ AList m n ] substType σ s ≡ substType σ t)
unify2 {m} t1 t2 = mgu2 {m} t1 t2
```

このように well-typed な型推論器の実装においては well-scoped の場合と同じく型変数を巧妙に操作することに加えて unify の拡張，それに基づき flexFlex や flexRigid といった機構の定式化も必要となった．

7　まとめ

本研究では，Agda で simply-typed lambda calculus の unification の機構を McBride の手法 [3] を用い，型推論器の実装を行った．さらに型推論器が正しいことを示すために，正当性の証明がついた型推論器を作成した．今後は syntax の拡張として，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．また，現在の実装だと where 節が冗長なため実行時間が比較的長くなってしまっており，かつ証明が煩雑なので実装の簡略化を試みていきたい．

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).