Agdaによる定式化された型推論器の拡張と改良

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、その一方で関数型プログラミング言語でもある。Agda では依存型 (Dependent 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。既存研究で我々は、McBride の理論をもとにして Agda による unification を実現し、それを用いて正当性が保証された型推論器を Agda により構成した。しかし、そこで完成させた型推論器は型変数の数を人の手によって計算・管理するもので、証明は煩雑であった。また、型を固定されたコンストラクタとして定義していたことにより、構文や型を拡張する際にさまざまな証明を書きなおさなければならなかった。そこで本研究では変数の数を不等式によって管理し、加えて *type descriptor* を用いて型を一般的に定義することで、構文の拡張と証明の簡略化を可能にした。

1 Introduction

Agda [7] 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り、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。依存型と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「ある言語の項のうち、言語内で Int 型が付けられるもの」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 [9] に基づいている。

1.1 研究背景

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 [5]。我々は昨年、この unification の理論をもとにして unification を実装し、それを用いて型変数の数を管理することによって、正当性の証明のついた型推論器を完全に pure functional に実装した [3]。

しかし、実装した型推論では、型変数の数を count 関数で数え、型変数の数が矛盾しないことを巧妙に管理する必要があった。この手法は、各構文に合わせて必要な型変数の数をいちいち数えるとともに、その数を型に含む命題を証明しなくてはならないため、人手で型変数の数を合わせなくてはならず、証明が（不可能ではないにしろ）非常に困難であった。さらに、これまでの型推論は、特定の型を対象にして作られていたため、構文を拡張して別の型を導入しようとすると、新しい構文の型推論に加えて、代入や unification など基本的な操作を全て再実装する必要があった。そのため、新しい型の導入が困難であった。

1.2 研究概要

前節で示した 2 つの問題のうち、前者の問題を回避するため、今回の型推論器では、型変数を数ではなく不等式を持ち歩くことで管理する手法を採用する。正確な数ではなく、満たすべき条件を不等式の形で持ち歩くことで、その条件さえ満たせば型推論を行えるようになる。満たすべき条件

は、証明の他の部分から必然的に導かれてくるため、必要な証明を Agda に指示されるままに証明すれば良く、証明は以前と比べてずっと簡明になる。その結果、型変数の数を考慮した型推論を入出力付きの型規則の形で定式化できた。

一方、もうひとつの問題を回避するため、型定義に generic programming [6] の考え方を取り入れる。McBride が示した unification [5] は関数型のみを対象とするものだったが、それを sum of product の形で表現される任意の型に拡張する。いったん、このような拡張を行うと、任意のデータ型に対する unification を行えるようになり、新たな型を導入する際にも同じ unification をそのまま使用することができるようになる。その結果、新たな構文を加える際には、その構文に対する型推論部分のみを実装すれば良いことになる。本稿では、その例として組（Fst, Snd, Cons）を示す。

本稿の貢献は、以下の 4 点にまとめられる。

- 不等式に基づいた型推論を Agda で実装するとともに、それを型規則の形で定式化した。不等式制約を用いることで、以前の実装よりも簡潔な証明が得られた。
- McBride が示した unification の機構 [5] を sum of product の形で表現される任意の型に拡張した。これにより、型の拡張を簡単にした。
- 型推論の結果、得られた型付き項から型を取り除くと元の項が得られることも証明した。
- 型推論の拡張の例として、組を導入した。

以下、2 節では、型推論器で用いる構文と型定義を示す。3 節では、 McBride による unification に加えて unification の正当性を保証した機構とその実装について述べる。4 節では、正当性の証明がついた型推論器の実装について、不等式を用いた代入について述べたのち、必要な証明に留意しながら述べる。5 節では、構文を拡張するにあたり使用した generic programming の手法について述べる。6 節では、5 節で拡張した構文 Fst, Snd, Cons それぞれに対して型推論の実装を与えていく。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。
`http://github.com/kdxu/InferAgda`

2 型と構文の定義

本節では、まず型推論器を実装するにあたって用いる型と構文、 unification の定義を見ていく。

2.1 型定義と推論規則

型定義は以下のように表現する。

$$\tau \quad = \quad m \mid \tau_1 \to \tau_2$$

自然数 m は型変数を表し、$\tau_1 \to \tau_2$ は関数型を表している。

項は以下のように表現する。

$$M \quad = \quad x \mid \lambda x.\, M_1 \mid App\ M_1\ M_2$$

型判断は以下の通りに行う。

$$\Gamma \vdash M : \tau$$

これは「型環境 Γ のもとで項 M は型 τ を持つ」と読む。

また、型規則は以下のようになる。

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_2 \vdash M_1 : \tau_1}{\Gamma \vdash \lambda x.\, M_1 : \tau_2 \to \tau_1} \qquad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash App\ M_1\ M_2 : \tau_1}$$

上に示したのは通常の型規則だが、本論文では型推論を表現するために以下のような特別な型判断を使う。

$$\sigma[\downarrow_{m'}^{m''}](\uparrow_{m}^{m''} \Gamma^{(m)}) \vdash M : \tau^{(m')}$$

ここで、入力（赤）は Γ, m, M で、出力（青）は σ, m', m'', τ である。（$\uparrow_{m}^{m''}$ は m と m'' から自動で挿入される。）Γ と τ の肩の添字は、「最大でいくつの型変数が現われ得るか」を示す。例えば、$\Gamma^{(m)}$ は 0 から $m-1$ までの番号がついた型変数が現われ得る。この型判断は以下のように読む。「型変数を最大 m 個持つ型環境 Γ のもとで項 M の型を推論すると、型変数を最大 m' 個含む型 τ が推論される。その推論途中で、新しい型変数を m'' 個まで割り当てる。その m'' 個の型変数は、$\sigma[\downarrow_{m'}^{m''}]$ で表される代入によって単一化が起こり、m' 個に減る。推論結果は以下を満たす。型環境 Γ の型変数の数を m'' まで持ち上げてから代入 σ をかけて得られた型環境のもとで項 M は型 τ を持つ。」

このような形で型推論を記述すると、型推論の結果、得られる型と代入が満たす性質も同時に記述できる。この書き方は後に述べる Agda による型推論の実装と厳密に対応したものとなっている。Agda による型推論の実装をこのようなわかりやすい形で提示し、メタ変数の数も含めて厳密な形で型推論を定式化したのは、本論文の貢献のひとつである。

同様にして、型推論時の unification は、以下のように表現する。

$$\sigma[\downarrow_{m'}^{m''}](\tau_1{}^{(m'')}) \doteq \sigma[\downarrow_{m'}^{m''}](\tau_2{}^{(m'')})$$

ここで、入力（赤）は τ_1, τ_2, m'' で、出力（青）は σ, m' である。τ_1 と τ_2 の肩の添字は、型変数の数を示す。この unification は以下のように読む。

「型変数を最大 m'' 個持つ型 τ_1 と τ_2 は、型変数を m'' 個から m' 個まで減らす代入 $\sigma[\downarrow_{m'}^{m''}]$ によって unification 可能である。」

2.2 構文定義

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。

入力される構文として、 自由変数の数が n 個の well-scoped な項として以下を定義する。

```
- 自由変数の数が n 個の well-scope な項
data WellScopedTerm (n : ℕ) : Set where
  Var : Fin n → WellScopedTerm n
  Lam : (s : WellScopedTerm (suc n)) → WellScopedTerm n
  App : (s1 : WellScopedTerm n) → (s2 : WellScopedTerm n) → WellScopedTerm n
```

ここで、Fin n は 0 から $n-1$ までの有限集合の型である。

出力される構文として、 well-typed な項を定義する。

```
- WellTypedTerm Γ t :  自由変数の数が n 個
- 型が t であるような well-typed な項
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'
```

これは自由変数の数が n 個、型環境 Γ において型が 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。

本節の型推論では、well-scoped な term を引数に取り型と必要な代入、対応する well-typed な term を返す関数 infer を作ることを目的とする。また、well-typed term から型情報を削除して well-scoped term に戻す erase 関数を用いて、もとの term と推論後の term が同一であることを確かめている。

3 Unification

本節では、型推論に必要な型の unification について述べる。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 [10]。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。m 個の「具体化されていない型変数」を Fin m 型の数字で表現し、その m 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。

今回はこの McBride の手法を用いて unification の関数 mgu を Agda で (今回使用する型定義の上で) 実装した。型定義は以下のようになる。

```
mgu : {D : Desc} → {m : ℕ} → (t1 t2 : Fix D m) → Maybe (Σ [ m' ∈ ℕ ] AList D m m')
```

ここで Desc 型は 5 節で説明する generic な型定義、Fix はその不動点である。Fix 型は unification を行う対象の型である。また AList $D\ m\ m'$ は、型変数の数を m から m' にする代入の型である。

加えて、well-typed な型推論器の実装においては、unification の際に単一化を行う代入 σ を得るのみでなく、「確かに単一化されている」という証明

$$\sigma[\downarrow_{m'}^{m''}](\tau_1{}^{(m'')}) \doteq \sigma[\downarrow_{m'}^{m''}](\tau_2{}^{(m'')})$$

も返すことが必要となる。

そこで、mgu にこの証明を加えて返す mgu2 関数を実装した。型定義は以下のようになる。

```
- 型 t1 と t2 を unify する代入を、確かに unify できることの証明とともに返す
mgu2 : {D : Desc} → {m : ℕ} → (t1 t2 : Fix D m) →
  Maybe (Σ [ m' ∈ ℕ ] Σ [ σ ∈ AList D m m' ] substFix σ t1 ≡ substFix σ t2)
```

ここで、substFix σ t は型 t に対して代入 σ を施す関数である。

4 型推論

本節では、以上の型定義、型規則、unification の実装を元に実際に型推論器を構成していく。本稿の型推論では、algorithm W [1] 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。本節では、まず Var、Lam、App に関して証明を与える。

4.1 不等式を用いた代入

既存研究 [3] による型推論では、型変数の数を count 関数で数え、型変数の数が矛盾しない証明と共に infer 関数に再帰的に持ち歩かせる必要があった。この手法は、構文に応じて型変数の数を人が厳密に数えて証明を実装する必要があり、型変数の数が複雑になる構文では実装が困難になるという問題があった。そこで今回の型推論器では、型変数を数ではなく不等式を持ち歩くことで管理する手法を採用した。

そもそも、代入 $\sigma[\downarrow_{m'}^{m''}]$ は m'' 個の型変数を持つ型を m' 個の型変数を持つ型にする代入であり、型判断は以下のように表現される。

$$\sigma[\downarrow_{m'}^{m''}](\uparrow_{m}^{m''} \Gamma^{(m)}) \vdash M : \tau^{(m')}$$

型環境における型変数の数が m である Γ に $\sigma[\downarrow_{m'}^{m''}]$ を施すとき、$\uparrow_{m}^{m''}$ は m と m'' から自動で挿入される。これは「型変数の数を m'' から m' にする代入 σ を型変数の数が m である Γ に施すとき、まず Γ のメタ変数の数を持ち上げて型変数の数を m'' に増やしてから代入を施す」という意味である。既存研究では、この m'' がいくつであるか厳密に計算していた。しかし、この代入を施すためには m'' が m 以上でなければならず、またその条件を満たす m'' が存在すれば良い、ということがわかる。よって、$m \leq m''$ のとき、かつその時に限り、

$$\sigma[\downarrow_{m'}^{m''}](\uparrow_{m}^{m''} \Gamma^{(m)})$$

を行うことができ、このとき、m'' は「条件を満たしていれば」なんでもよく、型変数の数をつねに厳密に把握している必要はない。よって、持ち歩くべき制約は $m \leq m''$ で十分だということが分かった。型変数に関する制約がゆるくなったことにより、人は不等式の証明だけすればよくなり、型変数を事細かに手計算する必要がなくなった。

4.2 infer 関数の型

実装すべき型推論器の関数定義は以下のようになる。

```
infer : (m : ℕ) → {n : ℕ} → (Γ : Cxt {m} n) → (s : WellScopedTerm n) →
  Maybe (Σ [ m'' ∈ ℕ ]
    Σ [ m' ∈ ℕ ]
    Σ [ m≤m'' ∈ m ≤ m'' ]
    Σ [ σ ∈ AListType m'' m' ]
    Σ [ τ ∈ Type m' ]
    Σ [ w ∈ WellTypedTerm (substCxt≤ σ m≤m'' Γ) τ ]
    erase w ≡ s)
```

この関数は以下のものを入出力する。

入力

- s : 最大で m 個の型変数をもつ WellScopedTerm
- Γ : 型環境

出力

- m'' , m' : Nat
- $m{\leq}m''$: $m \leq m''$ であることの証明
- σ : m'' 個の型変数を持つ型を m' 個の型変数を持つ型にする代入
- τ : 代入後の型 (Type m')
- w : WellTypedTerm Γ' τ
- $s \equiv erase\ w$: 返された well-typed term s から型情報を削除したものが、もとの well-scoped term と等しいという証明

以下の節で各構文に対する型推論の実装を説明する。各節の最後に最終的に得られた型推論を型規則の形で表現する。

4.3 Var

Var の型規則は以下のようになる。

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Var x の場合は、単に型環境に入っている x の型を返せばよいので、

$$\frac{\Gamma(x) = \tau}{\phi[\downarrow^m_m](\Gamma) \vdash x : \tau}$$

となり返す代入は $\phi[\downarrow^m_m]$（空の代入）、返す型は τ 、返す term は Var x となる。

ここで注意する点は、空の代入を施した Γ を well-scoped term の型環境として返しているので、「型環境に対して空の代入を入れたものがもとの型環境と同一である」という証明

$$\phi[\downarrow^m_m]\uparrow^m_m \Gamma^{(m)} = \Gamma^{(m)}$$

を入れることで、 term の型が同一であることを明示的に示すことが必要になる点である。この証明は、型環境の中のそれぞれの型 $\tau^{(m)}$ に対して

$$\phi[\downarrow^m_m](\tau^{(m)}) = \tau^{(m)}$$

となることを示し、型環境の中のそれぞれの型 $\tau^{(m)}$ に対してその証明を再帰的に与えてやればよい。

型推論をまとめると、以下のようになる。

$$\frac{\Gamma^{(m)}(x) = \tau^{(m)}}{\phi[\downarrow^m_m](\uparrow^m_m \Gamma^{(m)}) \vdash x : \tau^{(m)}}$$

4.4 Lam

つぎに term がラムダ式 Lam s の形になっている場合である。型規則は以下のとおりである。

$$\frac{\Gamma,\ x : m^{(m+1)} \vdash s : \tau_1}{\Gamma \vdash Lam\ s : m^{(m+1)} \to \tau_1}$$

ラムダ式の場合は、引数の型がどうなるかすぐにはわからないので、新しい型変数 $m^{(m+1)}$ を x の型に割り振った上で s の型推論を行う。s に型がついた場合には、「$m^{(m+1)} \to$ s の型」が全体の型となる。

まず、新しい型変数 $m^{(m+1)}$ を型環境に加えた上で s の型推論を行うと、

$$\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} ((\uparrow^{m+1}_m \Gamma), x : m^{(m+1)})^{(m+1)}) \vdash s : \tau^{(m')}$$

なる σ、τ を得る。これを変形すると、

$$\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} \uparrow^{m+1}_m \Gamma^{(m)}), x : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \vdash s : \tau^{(m')}$$

となる。

すると、Lam の型規則が適用できて、

$$\frac{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} \uparrow^{m+1}_m \Gamma^{(m)}), x : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \vdash s : \tau^{(m')} \quad m^{(m+1)} \text{ は新しい型変数}}{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} \uparrow^{m+1}_m \Gamma^{(m)}) \vdash Lam\ s : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \to \tau^{(m')}}$$

となる。

ここで、この型推論では、返ってくる代入を $\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_m \Gamma^{(m)})$ の形で出力するが、そのためには $\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} \uparrow^{m+1}_m \Gamma^{(m)})$ という式を $\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_m \Gamma^{(m)})$ に変形出来ればよい。

この変形は「型環境中の型変数の数を一気に持ち上げても、2 回に分けて持ち上げても同じ」という意味であり、直観的には正しいとわかるが、証明が必要になる。これは、Var x の場合と同様に型環境の中のそれぞれの型 τ に対してその性質を示してやればよく、

$$\uparrow^{m'}_{m+k} \uparrow^{m+k}_{m} \tau^{(m)} = \uparrow^{m'}_{m} \tau^{(m)}$$

が成り立っているということを示し、型環境の中のそれぞれの型 τ に対してその証明を再帰的に与えてやればよい。

この変形を用いると、先ほどの型推論は

$$\frac{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} \uparrow^{m+1}_{m} \Gamma^{(m)}), x : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \vdash s : \tau^{(m')} \quad m^{(m+1)} \text{ は新しい型変数}}{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m} \Gamma^{(m)}) \vdash Lam\ s : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \to \tau^{(m')}}$$

となる。必要な代入は $\sigma[\downarrow^{m''}_{m'}]$ であり、返ってくる型は $\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \to \tau^{(m')}$ となる。

以上の型推論をまとめると、以下のようになる。

$$\frac{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} ((\uparrow^{m+1}_{m} \Gamma), x : m^{(m+1)})^{(m+1)}) \vdash s : {\tau_1}^{(m')} \quad m^{(m+1)} \text{ は新しい型変数}}{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m} \Gamma^{(m)}) \vdash Lam\ s : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m+1} m^{(m+1)}) \to {\tau_1}^{(m')}}$$

4.5 App

App の型規則は以下の通り。

$$\frac{\Gamma \vdash s_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash App\ s_1\ s_2 : \tau_1}$$

App $s_1\ s_2$ の型推論では、まず s_1 を型推論すると、

$$\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)}) \vdash s_1 : {\tau_1}^{(m'_1)}$$

なる σ_1 と τ_1 が得られる。

次に、得られた代入 σ_1 をかけた型環境下で s_2 を型推論し、

$$\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} (\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)}))^{(m'_1)}) \vdash s_2 : {\tau_2}^{(m'_2)}$$

なる σ_2 と τ_2 を得る。

型推論が成功するには、τ_1 は $\tau_2 \to \beta$ の形をしているはずなので、unification を行うと、

$$\sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} {\tau_1}^{(m'_1)}))) \doteq \sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} {\tau_2}^{(m'_2)} \to {m'_2}^{(m'_2+1)})$$

$({m'_2}^{(m'_2+1)}$ は新しい型変数$)$ となる σ_3 と、上記の等式 eq が得られる。

次に、導出された式の両辺のメタ変数の数を持ち上げ、代入をかける。ここの四角は、枠内のすべての型変数に枠外のメタ変数の数の持ち上げや代入を行うという意味である。

$$\sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} \boxed{\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)}) \vdash s_1 : {\tau_1}^{(m'_1)}})))$$

すると、

$$\sigma_3[\downarrow^{m'_2+1}_{m'_3}]((\uparrow^{m'_2+1}_{m'_2} \sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} \sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)})))) \vdash s_1 : \sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} \sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} {\tau_1}^{(m'_1)}))$$

のようになり、これの s_1 の型は eq から

$$\sigma_3[\downarrow^{m'_2+1}_{m'_3}]((\uparrow^{m'_2+1}_{m'_2} \sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} \sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)})))) \vdash s_1 : \sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} {\tau_2}^{(m'_2)} \to {m'_2}^{(m'_2+1)})$$

となる。

また、s_2 に関しても同様のことを行い

$$\sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} \boxed{\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} (\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)}))^{(m'_1)}) \vdash s_2 : {\tau_2}^{(m'_2)}})$$

を

$$\sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} \sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} \sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)})^{(m'_1)})) \vdash s_2 : \sigma_3[\downarrow^{m'_2+1}_{m'_3}](\uparrow^{m'_2+1}_{m'_2} {\tau_2}^{(m'_2)})$$

と変形する。

これらを用いると、App の型規則が適用できて

$$\sigma_3[\downarrow_{m_3'}^{m_2'+1}]\uparrow_{m_2'}^{m_2'+1} \sigma_2[\downarrow_{m_2'}^{m_2''}](\uparrow_{m_1'}^{m_2''} \sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma^{(m)})) \vdash s_1 : \sigma_3[\downarrow_{m_3'}^{m_2'+1}](\uparrow_{m_2'}^{m_2'+1} {\tau_2}^{(m_2')} \to {m_2'}^{(m_2'+1)})$$

$$\frac{\sigma_3[\downarrow_{m_3'}^{m_2'+1}](\uparrow_{m_2'}^{m_2'+1} \sigma_2[\downarrow_{m_2'}^{m_2''}](\uparrow_{m_1'}^{m_2''} \sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma^{(m)}))) \vdash s_2 : \sigma_3[\downarrow_{m_3'}^{m_2'+1}](\uparrow_{m_2'}^{m_2'+1} {\tau_2}^{(m_2')})}{\sigma_3[\downarrow_{m_3'}^{m_2'+1}](\uparrow_{m_2'}^{m_2'+1} \sigma_2[\downarrow_{m_2'}^{m_2''}](\uparrow_{m_1'}^{m_2''} \sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma^{(m)}))) \vdash App\ s_1\ s_2 : \sigma_3[\downarrow_{m_3'}^{m_2'+1}]({m_2'}^{(m_2'+1)})}$$

が得られる。ここで、必要な代入を返すにあたり、$\sigma_3, \sigma_2, \sigma_1$ を合成したものを返したくなるが、そのとき、以下の等式を示すことが必要になる。

$$\sigma[\downarrow_{m_3'}^{m_3''}](\uparrow_{m}^{m_3''} \Gamma) = \sigma_3[\downarrow_{m_3'}^{m_2'+1}](\uparrow_{m_2'}^{m_2'+1} (\sigma_2[\downarrow_{m_2'}^{m_2''}](\uparrow_{m_1'}^{m_2''} (\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma)))))$$
$$m_3'' = ((m_2' + 1) - m_2') + (m_2'' - m_1') + m_1''$$

ここで、σ は $\sigma_3, \sigma_2, \sigma_1$ を合成したものである。これを示すには、型環境 $\Gamma^{(m)}$ のそれぞれの型 τ において

$$\sigma[\downarrow_{m_3'}^{m_3''}](\uparrow_{m}^{m_3''} \tau) = \sigma_3[\downarrow_{m_3'}^{m_2'+1}](\uparrow_{m_2'}^{m_2'+1} (\sigma_2[\downarrow_{m_2'}^{m_2''}](\uparrow_{m_1'}^{m_2''} (\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \tau)))))$$

が成り立つことを示し、この証明を各 τ に対して再帰的に与えてやれば良い。

ここで、σ は σ_1 と σ_2 と σ_3 の合成であることから、これは「複数の代入を順に施すのは、代入を合成したものを 1 度に施すのと同じである」という意味であり、直感的にはこれらが等しいことは理解できる。

以下の手順により、これを証明することが出来る。

- 有限集合の型 Fin を不等式を用いて持ち上げる関数 inject$\leq$ に関して推移律を示す。
- それを用いて、型の型変数を持ち上げる関数 liftFix$\leq$ に関して推移律を示す。
- それを用いて、型に 2 つの代入を順に施すのと代入を合成したものを 1 度に施したものは同じであるということを示す。

代入の合成は、以下のように実装している。

- まず σ_1 と σ_2 を合成する場合、後ろの代入 σ_2 を σ_1 の型まで持ち上げ、持ち上げた σ_2 を σ_2' とする。
- σ_2'(のそれぞれの型と型変数の組) を σ_1 の後ろに再帰的に追加する。

代入を持ち上げるとき、各要素の型と型変数を 1 ずつ持ち上げている。型を持ち上げるとき、型変数を持ち上げることが必要になるが、型変数を持ち上げるということは有限集合の型（すなわち型変数の型）Fin を持ち上げるということである。例えば、inject+' m' は Fin m を Fin $(m' + m)$ に持ち上げる関数である。これを用いて、liftFix という関数を型を「ちょうど m' 個」持ち上げる関数として定義する。

しかし、型推論中、いくつのメタ変数を使うかを全て管理するのはとても煩雑である。むしろ、$m \leq m''$ である任意の m'' まで持ち上げる形の方が後の証明がとても楽になる。それには inject+' ではなく不等式を受け取ってその分だけ持ち上げる関数 inject$\leq$ を全ての型変数に施すことで、不等式を受け取ったら型をその分だけ持ち上げる関数 liftFix$\leq$ を定義する。

代入同士の合成では、この liftFix$\leq$ を再帰的に用いて、代入を lift している。また、代入を実際に施す際も、liftFix$\leq$ を用いて型を持ち上げてから、代入を施している。つまり、liftFix$\leq$ の中で使われている inject$\leq$ に関する推移律を示してやれば、証明はかなり長くなるが、上記の型環境が互いに等しいことを示すことが出来る。

また、ここの証明では、関数同士の同一性を示すため、Agda で用意されているデフォルトの公理に加え、外延性を仮定することが必要になる。

以上の証明を用いて型推論をまとめると、以下のようになる。

$$\frac{\begin{array}{l}\sigma_3[\downarrow^{m_2'+1}_{m_3'}](\uparrow^{m_2'+1}_{m_2'}\ (\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'}\ \boxed{\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m}\ \Gamma^{(m)})\quad \vdash s_1 : \tau_1{}^{(m_1')}}\ )))\\ \sigma_3[\downarrow^{m_2'+1}_{m_3'}](\uparrow^{m_2'+1}_{m_2'}\ \boxed{\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'}\ (\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m}\ \Gamma^{(m)}))^{(m_1')}) \vdash s_2 : \tau_2{}^{(m_2')}}\ )\\ \sigma_3[\downarrow^{m_2'+1}_{m_3'}](\uparrow^{m_2'+1}_{m_2'}\ (\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'}\ \tau_1{}^{(m_1')}))) \doteq \sigma_3[\downarrow^{m_2'+1}_{m_3'}](\uparrow^{m_2'+1}_{m_2'}\ \tau_2{}^{(m_2')} \to m_2'{}^{(m_2'+1)})\\ m_2'{}^{(m_2'+1)} \text{ は新しい型変数}\\ \sigma[\downarrow^{m_3''}_{m_3'}](\uparrow^{m_3''}_{m}\ \tau) = \sigma_3[\downarrow^{m_2'+1}_{m_3'}](\uparrow^{m_2'+1}_{m_2'}\ (\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'}\ (\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m}\ \tau)))))\\ m_3'' = ((m_2'+1)-m_2') + (m_2''-m_1') + m_1''\end{array}}{\sigma[\downarrow^{m_3''}_{m_3'}](\uparrow^{m_3''}_{m}\ \Gamma^{(m)}) \vdash App\ s_1\ s_2 : \sigma_3[\downarrow^{m_2'+1}_{m_3'}](m_2'{}^{(m_2'+1)})}$$

5 構文の拡張

これまでは型をコンストラクタを用いて構成していたが、この構成法だと新しい型（例えば直積型）などを加える際、型に対してパターンマッチを行って構成している mgu 関数や、型や型環境に対して lift や subst を行う関数を定義しなおさなければならず、型の拡張が面倒なものとなっていた。そこで今回は型を *type discriptor* を用いて表現する方法を採用した。

5.1 より一般的な型への拡張

単純型付き λ 計算に対する型推論のみが必要なのであれば、関数の型のみ扱えるような型推論を作れば良いが、いろいろな体系に対する型推論を作ろうと思えば多様な型を扱える必要がある。しかし、それぞれの体系ごとに別の型推論を実装し、その正当性を示すのは現実的ではない。そこで、ここでは generic programming の考え方を取り入れて、ユーザが指定するいろいろな型を扱えるように全体を構築する。generic programming の中でも本稿で実装したのは Regular [6] である。

generic programming では、型の構成法を BNF に似た形で与える。例えば、単純型付き λ 計算の場合、型は次のように定義される。

$$\tau \quad = \quad \mathsf{base} \mid \tau \to \tau$$

このような定義では、「または」を表す縦棒で選択肢が区切られ、各選択肢では列挙された要素で構成される。（例えば $\tau \to \tau$ なら、ふたつの型から構成される。）そして、その要素は再帰的に型自身でも構わない。このような型を表現するため、generic programming では、次のような Desc 型のデータ構造を使う。

```
data Desc : Set₁ where
  base  : Desc
  _:+:_ : Desc → Desc → Desc
  _:*:_ : Desc → Desc → Desc
  rec   : Desc
```

ここで base は適当な基底型（本稿では unit 型とする）、:+: は「または」、:*: は「かつ」を示し、rec はそこで再帰が起こることを示す。例えば、単純型付き λ 計算の型は、次のようなデータで定義する。

```
TypeDesc1 : Desc
TypeDesc1 = base :+: rec :*: rec
```

ここで最初の base が基底型を表し、次の rec :*: rec が $\tau \to \tau$ を表している。関数型の両辺は任意の型が来て良いため rec となっている。

これで型の定義を与えることが出来るようになったが、具体的な型を表現するためには、まず Desc の Agda 上の表現（意味）を以下のように与える。

```
⟦_⟧ : Desc → Set → Set
⟦ base    ⟧ X = ⊤
⟦ d1 :+: d2 ⟧ X = ⟦ d1 ⟧ X ⊎ ⟦ d2 ⟧ X
⟦ d1 :*: d2 ⟧ X = ⟦ d1 ⟧ X × ⟦ d2 ⟧ X
⟦ rec     ⟧ X = X
```

⟦ D ⟧X は、「再帰部分が X であるような Desc 型の値」を示す。この再帰部分 X に自分自身を入れる（再帰を閉じる）と、具体的に Desc 型の値を作ることが出来るようになる。再帰を閉じるには、以下のデータを使う。

```
data Fix (D : Desc) (m : ℕ) : Set where
  F : ⟦ D ⟧ (Fix D m) → Fix D m
  M : (x : Fin m) → Fix D m
```

ここで Fix D m 型の値は「型変数を m 個持つような D 型の値」を表す。F が具体的な値を作り出す構成子である。F の引数は ⟦ D ⟧ (Fix D m) となっているが、再帰部分に自分自身である Fix D m が入っているところに注目しよう。ここで、再帰を閉じることで任意に大きなデータを作ることが出来る。ここで、型変数を m 個持つ TypeDesc1 型の値（単純型付き λ 計算の型を表す型）を以下のように書くことにする。

```
Type1 : (m : ℕ) → Set
Type1 m = Fix TypeDesc1 m
```

すると、例えば基底型は以下のように定義できる。

```
tbase1 : {m : ℕ} → Type1 m
tbase1 = F (inj₁ tt)
```

また、関数型を作る関数を定義すると、さらに大きな型を定義できる。

```
_=>_ : {m : ℕ} → Type1 m → Type1 m → Type1 m
t1 => t2 = F (inj₂ (t1 , t2))

t1 : Type1 0
t1 = tbase1 => (tbase1 => tbase1)
```

ここで t1 は base → base → base を表す。

ここまでの例では、型のメタ変数は出て来なかったが、型推論を実装する際には任意の型に具体化可能なメタ変数が必要となる。それを表すのが Fix のもうひとつの構成子である M である。例えば

```
t1' : Type1 2
t1' = M zero => (tbase1 => M (suc zero))
```

は、M zero と M (suc zero) のふたつのメタ変数を含む Type1 2 型の値である。

Fix を使って、任意の Desc 型に対応したデータを作れるようになったが、Fix で表現された値を使って計算するには fold の一般形を使う。リストに対する fold では、cons と nil に対応した処理を記述したが、Fix に対する fold では、F と M に対応する処理を記述する形になる。Fix に対する fold は以下のように定義される。

```
{-# NO_TERMINATION_CHECK #-}
fold : {D : Desc} → {m : ℕ} → {X : Set} →
   (f-F : ⟦ D ⟧ X → X) → (f-M : Fin m → X) → Fix D m → X
fold {D} f-F f-M (F d) = f-F (fmap D (fold f-F f-M) d)
fold f-F f-M (M x) = f-M x
```

fold は、Fix D m 型の値を受け取ったら、その F 部分には f-F を適用し、M 部分には f-M を適用する。f-F や f-M は X 型の値を出力し、最終結果も X 型の値となる。f-F の第 1 引数の再帰部分が X となっていることに注意しよう。f-F が受け取る引数の再帰部分は、再帰的に計算をした結果となっている。実際、fold の定義の F d の場合を見ると、f-F の引数には fmap を使って fold f-F f-M 自身を再帰的に d に施した結果が渡されている。ここで、fmap は、Haskell の Functor にも見られる引数の関数を再帰部分に適用する関数で、以下のように定義される。

```
fmap : (D : Desc) → {X Y : Set} → (X → Y) → ⟦ D ⟧ X → ⟦ D ⟧ Y
fmap base f x = tt
fmap (D1 :+: D2) f (inj₁ x) = inj₁ (fmap D1 f x)
fmap (D1 :+: D2) f (inj₂ x) = inj₂ (fmap D2 f x)
fmap (D1 :*: D2) f (x1 , x2) = (fmap D1 f x1 , fmap D2 f x2)
fmap rec f x = f x - 再帰部分に f を適用する
```

このように fold は Fix D m 型のデータの構造に従って再帰的に計算を行う。従って fold の再帰の停止性はデータ構造の有限性から明らかである。しかし、プログラムの表面上は fold f-F f-M という再帰呼び出しが fmap に渡される形になっているため、Agda の停止性チェッカでは停止性を確認できない。そのため、fold 関数の定義では停止性チェックを行わないように指示している。

fold を使うと、例えば受け取った値のノード数を数えるには以下のようにする。

```
size-M : {m : ℕ} → Fin m → ℕ
size-M x = 1

size-F : {D : Desc} → ⟦ D ⟧ ℕ → ℕ
size-F {base} tt = 1
size-F {D1 :+: D2} (inj₁ t) = size-F {D1} t
size-F {D1 :+: D2} (inj₂ t) = size-F {D2} t
size-F {D1 :*: D2} (t1 , t2) = size-F {D1} t1 + size-F {D2} t2
size-F {rec} t = t

size : {D : Desc} → {m : ℕ} → Fix D m → ℕ
size {D} = fold (size-F {D}) size-M
```

size-M は、メタ変数のサイズは 1 であることを、size-F は、base の大きさが 1 で、他は部分木の大きさの和であることを示している。この関数は任意の Desc に対して使うことができ、例えば先の t1' なら size t1' は 3 となる。

また、型のメタ変数の数を m' 個増やす liftFix 関数は、fold を使って以下のように実装される。

```
- liftFix m' t :  t の中の型変数の数を m' だけ増やす
liftFix : {D : Desc} → (m' : ℕ) → {m : ℕ} → Fix D m → Fix D (m' + m)
liftFix {D} m' {m} t = fold F (M ∘ (inject+' m')) t
```

t の中に出てくる全てのメタ変数に対して inject+' m' をかけている。inject+' m' は Fin m を Fin $(m' + m)$ に持ち上げる関数である。

inject≤ を使って $m \leq m''$ である任意の m'' まで型を持ち上げる liftFix≤ 関数も、fold を用いて以下のように実装される。

```
- liftFix≤ m≤m' t :  t の中の型変数の数を m から m' に増やす
liftFix≤ : {D : Desc} → {m m' : ℕ} → (m≤m' : m ≤ m') → Fix D m → Fix D m'
liftFix≤ m≤m' t = fold F (M ∘ (λ x → inject≤ x m≤m')) t
```

これらの liftFix, liftFix≤ を含む fold を使った関数はいずれも任意の Desc に対して動くことに注意しよう。このように generic に定義をしておくと、型推論をユーザの与えた任意の Desc に対して行うことが出来るようになる。

5.2 新しい構文と追加する型

構文を追加する例として、ここでは Cons, Fst, Snd を追加する。入力する term の構文として、以下を追加する。

```
Fst : WellScopedTerm n → WellScopedTerm n
Snd : WellScopedTerm n → WellScopedTerm n
Cons : WellScopedTerm n → WellScopedTerm n → WellScopedTerm n
```

また、出力する term の構文として、以下を追加する。

```
Fst : {t1 t2 : Type m} → WellTypedTerm Γ (t1 ∏ t2) → WellTypedTerm Γ t1
Snd : {t1 t2 : Type m} → WellTypedTerm Γ (t1 ∏ t2) → WellTypedTerm Γ t2
Cons : {t1 t2 : Type m} → WellTypedTerm Γ t1 → WellTypedTerm Γ t2 → WellTypedTerm Γ (t1 ∏ t2)
```

本稿では、構文上、Agda の直積と区別するため、直積を $\times$ ではなく Π で表現していることに注意したい。

構文を拡張するにあたって、型と型規則を新たに加える。

型：

$$\tau \quad = \quad m \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$$

ここで、新たに直積型 $\tau_1 \times \tau_2$ を加えている。

追加された型規則：

$$\frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash Cons\ s_1\ s_2 : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash s : \tau_1 \times \tau_2}{\Gamma \vdash Fst\ s : \tau_1} \qquad \frac{\Gamma \vdash s : \tau_1 \times \tau_2}{\Gamma \vdash Snd\ s : \tau_2}$$

これらはそれぞれ、Cons、Fst、Snd に関する型規則である。

6 拡張された構文における型推論

前節では、generic programming と不等式の導入により、mgu 関数等を変更することなく、より簡単に構文を拡張することができた。本節では、拡張されたそれぞれの構文に対する型推論の実装を与えていく。

6.1 Cons

Cons の型規則は以下のとおりである。

$$\frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash Cons\ s_1\ s_2 : \tau_1 \times \tau_2}$$

Cons s_1 s_2 の型推論では、まず s_1 を型推論し、σ_1 と型 τ_1、型変数 m_1' を得る。すると、以下の導出が得られる。

$$\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m}\ \Gamma) \vdash s_1 : {\tau_1}^{(m_1')}$$

つぎに、s_2 を（s_1 の型推論で得られた σ_1 を適用した上で）型推論する。

$$\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'}\ \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m}\ \Gamma)) \vdash s_2 : {\tau_2}^{(m_2')}$$

つぎに、先ほどの s_1 の式の両辺を σ_2 で subst して、

$$\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'}\ \boxed{\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m}\ \Gamma^{(m)}) \vdash s_1 : {\tau_1}^{(m_1')}})$$

から

$$\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash s_1 : \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \tau_1)$$

を得る。

すると、Cons の型規則が適用できて、

$$\frac{\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash s_1 : \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \tau_1) \quad \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash s_2 : \tau_2}{\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash Cons\ s_1\ s_2 : \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \tau_1) \times \tau_2}$$

が得られ、ここで σ を σ_1 と σ_2 の合成として

$$\sigma[\downarrow^{m_2'''}_{m_2'}](\uparrow^{m_2'''}_{m} \tau) = \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} (\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \tau)))$$

$(m_2''' = (m_2'' - m_1') + m_1'')$ と変形できれば、

$$\frac{\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash s_1 : \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \tau_1) \quad \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash s_2 : \tau_2}{\sigma[\downarrow^{m_2'''}_{m_2'}](\uparrow^{m_2'''}_{m} \Gamma) \vdash Cons\ s_1\ s_2 : \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \tau_1) \times \tau_2}$$

となり、返ってくる代入として σ が得られる。

ここで、σ は σ_1 と σ_2 の合成であることから、これは「ふたつの代入を順に施すのは、代入を合成したものを 1 度に施すのと同じである」という意味であり、App の場合に用いた証明と同じものを用いて示す事ができる。

以上の型推論をまとめると、以下のようになる。

$$\frac{\begin{array}{l}\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \boxed{\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma^{(m)}) \vdash s_1 : \tau_1{}^{(m_1')}}) \\ \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} (\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma^{(m)}))^{(m_1')}) \vdash s_2 : \tau_2{}^{(m_2')} \\ \sigma[\downarrow^{m_2'''}_{m_2'}](\uparrow^{m_2'''}_{m} \tau) = \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} (\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \tau))) \\ m_2''' = (m_2'' - m_1') + m_1''\end{array}}{\sigma[\downarrow^{m_2'''}_{m_2'}](\uparrow^{m_2'''}_{m} \Gamma^{(m)}) \vdash Cons\ s_1\ s_2 : \sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} \tau_1{}^{(m_1')}) \times \tau_2{}^{(m_2')}}$$

6.2 Fst

Fst の型規則は以下のとおりである。

$$\frac{\Gamma \vdash s : \tau_1 \times \tau_2}{\Gamma \vdash Fst\ s : \tau_1}$$

Fst s の型推論ではまず s を型推論し、以下の導出を得る。

$$\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma) \vdash s : \tau^{(m_1')}$$

また、新しい型変数 2 つ $\uparrow^{m_1'+2}_{m_1'+1} m_1'{}^{(m_1'+1)}$ と $(m_1'+1)^{(m_1'+2)}$ を用いると、s の型 τ_1 は $\uparrow^{m_1'+2}_{m_1'+1} m_1'{}^{(m_1'+1)} \times (m_1'+1)^{(m_1'+2)}$ の形をしているはずなので unification を行い、成功した場合のみ代入 σ_2 と以下の等式 eq が得られる。

$$\sigma_2[\downarrow^{m_1'+2}_{m_2'}](\uparrow^{m_1'+2}_{m_1'} \tau) \doteq \sigma_2[\downarrow^{m_1'+2}_{m_2'}](\uparrow^{m_1'+2}_{m_1'+1} m_1'{}^{(m_1'+1)} \times (m_1'+1)^{(m_1'+2)})$$

ここで、s の導出式の両辺に σ_2 をかける。

$$\sigma_2[\downarrow^{m_1'+2}_{m_2'}](\uparrow^{m_1'+2}_{m_1'} \boxed{\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma^{(m)}) \vdash s : \tau^{(m_1')}})$$

すると、以下のようになる。

$$\sigma_2[\downarrow^{m_1'+2}_{m_2'}](\uparrow^{m_1'+2}_{m_1'} \sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma)) \vdash s : \sigma_2[\downarrow^{m_1'+2}_{m_2'}](\uparrow^{m_1'+2}_{m_1'} \tau)$$

ここで、eq を適用すると、Fst の型規則が適用できて、

$$\frac{\sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} \sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma)) \vdash s : \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'+1}^{m_1'+2} {m_1'}^{(m_1'+1)} \times (m_1'+1)^{(m_1'+2)})}{\sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} \sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma)) \vdash Fst\ s : \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'+1}^{m_1'+2} {m_1'}^{(m_1'+1)})}$$

が得られ、返り値として得られる代入 σ は、σ_1 と σ_2 を合成したものであり、

$$\sigma[\downarrow_{m_2'}^{m_2''}](\uparrow_{m}^{m_2''} \tau) = \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} (\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \tau)))$$
$$m_2'' = ((m_1'+2) - m_1') + m_1''$$

を満たすものである。ここで用いる証明は App で用いた「ふたつの代入を順に施すのは、代入を合成したものを 1 度に施すのと同じである」の証明と同じである。

以上の推論をまとめると以下のようになる。

$$\frac{\begin{array}{l}\sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} \boxed{\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma^{(m)}) \vdash s : \tau^{(m_1')}}) \\ \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} \tau_1^{(m_1')}) \doteq \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'+1}^{m_1'+2} {m_1'}^{(m_1'+1)} \times (m_1'+1)^{(m_1'+2)}) \\ {m_1'}^{(m_1'+1)}, (m_1'+1)^{(m_1'+2)} \text{ は新しい型変数} \\ \sigma[\downarrow_{m_2'}^{m_2''}](\uparrow_{m}^{m_2''} \tau) = \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} (\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \tau))) \\ m_2'' = ((m_1'+2) - m_1') + m_1''\end{array}}{\sigma[\downarrow_{m_2'}^{m_2''}](\uparrow_{m}^{m_2''} \Gamma^{(m)}) \vdash Fst\ s : \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'+1}^{m_1'+2} {m_1'}^{(m_1'+1)})}$$

6.3 Snd

Snd の型規則は以下のとおりである。

$$\frac{\Gamma \vdash s : \tau_1 \times \tau_2}{\Gamma \vdash Snd\ s : \tau_2}$$

Snd s に関しては、返される型が $\tau \times \tau'$ の τ' の方であるという部分のみ異なり、示すべき証明は Fst s と同じである。

型推論は以下のようになる。

$$\frac{\begin{array}{l}\sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} \boxed{\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \Gamma^{(m)}) \vdash s : \tau^{(m_1')}}) \\ \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} \tau_1^{(m_1')}) \doteq \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'+1}^{m_1'+2} {m_1'}^{(m_1'+1)} \times (m_1'+1)^{(m_1'+2)}) \\ {m_1'}^{(m_1'+1)}, (m_1'+1)^{(m_1'+2)} \text{ は新しい型変数} \\ \sigma[\downarrow_{m_2'}^{m_2''}](\uparrow_{m}^{m_2''} \tau) = \sigma_2[\downarrow_{m_2'}^{m_1'+2}](\uparrow_{m_1'}^{m_1'+2} (\sigma_1[\downarrow_{m_1'}^{m_1''}](\uparrow_{m}^{m_1''} \tau))) \\ m_2'' = ((m_1'+2) - m_1') + m_1''\end{array}}{\sigma[\downarrow_{m_2'}^{m_2''}](\uparrow_{m}^{m_2''} \Gamma^{(m)}) \vdash Snd\ s : \sigma_2[\downarrow_{m_2'}^{m_1'+2}]((m_1'+1)^{(m_1'+2)})}$$

7 まとめと今後の課題

7.1 まとめ

本研究では、既存研究である McBride [5] の手法を用いた正当性の証明がついた型推論器 [3] を型変数を数ではなく不等式を用いて表現する方式に改良し、また型定義を Type Descriptor を用いてより一般的に改良することで、syntax の拡張と証明の簡略化を実現した。

7.2 今後の課題

今後の課題として、1 つ目はより syntax を拡張すること（例えば、 多相の Let 文など）である。本論文では generic programming の考え方を使って任意の型を扱えるようにしたが、多相の let 文を入れようと思うと（オブジェクト言語レベルの）型変数（型パラメタ）が必要になる。しかし、今回、採用した generic programming は型パラメタを扱えない。これを扱うには PolyP [2] や Indexed Functors [4] など型パラメタを扱えるものを使う必要がある。

2 つ目は、完全性の証明を加えることである。例えば現時点ではすべて nothing を返す infer 関数を実装することもできてしまう。つまり、型推論が失敗した時に nothing を返すのではなく、入力された構文に対応する型付けされた構文が存在しないという証明、式にすると

$$\neg(\exists\ \mathrm{w} :\ \mathrm{WellTypedTerm}\ (\mathrm{erase}\ w \equiv s))$$

を返す infer を実装することが必要になる。

謝辞　今回、コメントを下さった査読者の方々に感謝いたします。本研究は、JSPS 科研費 25280020 の助成を受けたものです。

参考文献

[1] 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 (1982).

[2] Jansson, P., and J. Jeuring “PolyP—a polytypic programming language extension," *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 470–482 (1997).

[3] 門脇 香子, 浅井 健一「Agda による型推論器の定式化」第１７回プログラミングおよびプログラミング言語ワークショップ論文集.(2015)

[4] Löh, A. and J. P. Magalhães “Generic programming with indexed functors,” *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP’11)*, pp. 1–12 (2011).

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

[6] Noort, T. van, A. R. Yakushev, S. Holdermans, J. Jeuring, and B. Heeren “A lightweight approach to datatype-generic rewriting,” *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP’08)*, pp. 13–24 (2008).

[7] 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).

[8] 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 (2013).

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

[10] 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).