依存型付多段階計算体系

Dependently Typed Multi-stage Calculus

河田 旺

五十嵐・末永研究室

2020年2月12日

目次

多段階計算とはなにか

以下の 2 つを可能にするプログラミング手法

- **実行時のコード生成**
- **生成したコードの実行**

応用例

- **実行時の情報を使ったプログラムの高速化** [Taha'07]
- **領域特化言語 (DSL) の効率的な実装** [Kiselyov'18]

MetaOCamlによる多段階計算の例1

- ▶ でコードの生成 (▷ int は int 型のコードの型)

```
# let a = ▶ (1 + 2)
val a : ▷ int = ▶ (1 + 2)
```

- ◀ で、生成したコードを他のコードに埋め込める

```
# let b = ▶ ((◀ a) + (◀ a))
val b : ▷ int = ▶ ((1 + 2) + (1 + 2))
```

MetaOCamlによる多段階計算の例2

run でコードの実行

```
# run (▶ (1 + 2))
- : int = 3
# run b
- : int = 6
```

多段階計算の応用例: vadd-gen

ベクトルの長さを受け取って、
その長さのベクトルを足し合わせる関数を作る関数

```
val vadd-gen :  int -> ▷ (vec -> vec -> vec)
# let vadd3 = vadd-gen 3
val vadd3 :  ▷ (vec -> vec -> vec)
     = ▶ (fun v w ->
        [v[0]+w[0];v[1]+w[1];v[2]+w[2]] )
```

多段階計算の応用例: vadd-gen

ベクトルの長さを受け取って、
その長さのベクトルを足し合わせる関数を作る関数

```
val vadd-gen :  int -> ▷ (vec -> vec -> vec)
# let vadd3 = vadd-gen 3
val vadd3 :  ▷ (vec -> vec -> vec)
      = ▶ (fun v w ->
          [v[0]+w[0];v[1]+w[1];v[2]+w[2]] )
```

ループを使わないプログラム

vadd-gen によって生成された vadd3 の問題点

誤った長さのベクトルを渡せてしまう

```
val vadd3 : ▷ (vec -> vec -> vec)
# (run vadd3) [1;2;3] [4;5;6]
- : vec = [5;7;9]
# (run vadd3) [1;2] [4;5]
---> 実行時エラー発生
```

目次

アイデア: 依存型の導入

依存型とは

- 項がパラメータとしてついている型
- ベクトルの長さが表現できる

```
[1;2]: vec 2
vadd3 : ▷ (vec 3 -> vec 3 -> vec 3)

# (run vadd3) [1;2] [4;5]
---> 実行する前に型で誤りを検知
```

目次

本研究の貢献: λ^{MD}

既存の多段階計算体系 $\lambda^{\triangleright\%}$ [Hanada&Igarashi'14] を拡張して依存型付多段階計算体系 λ^{MD} を設計した

- 多段階計算の機能
- 依存型の機能

技術的な貢献

- λ^{MD} の形式化
- 型安全性の証明
- 型推論アルゴリズムの設計と正当性の証明

λ^{MD} の項

項

$$
\begin{array}{lll}
M ::= & & \\
& c \mid x \mid \lambda x : \tau . M \mid M \ M & \text{(通常の } \lambda \text{ 計算の項)} \\
& \mid \blacktriangleright M & \text{(コードの生成)} \\
& \mid \blacktriangleleft M & \text{(コードの埋め込み)} \\
& \mid \texttt{run} \ M & \text{(コードの実行)} \\
& \mid \% M & \text{(ステージを跨ぐ項の} \\
& & \text{ コードへの埋め込み (CSP))}
\end{array}
$$

* 説明の為に単純化しています

ステージとは何か?

項の外側にある ▶ の数

$$\underbrace{\texttt{λx:int.(▶}\overbrace{\texttt{(λy:int.y+1)}}^{\text{stage } 1}\texttt{)}}_{\text{stage } 0}$$

項には型とステージがある

- x は int 型のステージ 0 の変数
- y は int 型のステージ 1 の変数

ステージの存在意義

簡約を正しい順序で行う

$$\underbrace{\texttt{run}\ \ (\blacktriangleright\overbrace{(1+2)}^{\text{stage 1}})}_{\text{stage 0}}$$

ステージの存在意義

簡約を正しい順序で行う

$$\underbrace{\mathtt{run}\ \ (\blacktriangleright\ \overbrace{(1{+}2)}^{\text{stage }1}\ )}_{\text{stage }0}$$

$$\longrightarrow\ 1{+}2$$

ステージの存在意義

簡約を正しい順序で行う

$$\underbrace{\texttt{run}\ (\blacktriangleright\ \overbrace{(1+2)}^{\text{stage }1}\ )}_{\text{stage }0}$$

$$\longrightarrow 1+2$$

$$\longrightarrow 3$$

ステージの存在意義

簡約を正しい順序で行う

$$\underbrace{\texttt{run}\ (\blacktriangleright \overbrace{(1+2)}^{\text{stage } 1})}_{\text{stage } 0}$$

誤った簡約順序

$\longrightarrow \texttt{run}\ (\blacktriangleright 3)$

$\longrightarrow 3$

$\longrightarrow 1+2$

$\longrightarrow 3$

ステージの存在意義

簡約を正しい順序で行う

$$\underbrace{\texttt{run}\ (\blacktriangleright\ \overbrace{\texttt{(1+2)}}^{\text{stage } 1}\ )}_{\text{stage } 0}$$

誤った簡約順序

$\longrightarrow$ `run (▶3)`

$\longrightarrow$ 3

$\longrightarrow$ 1+2

$\longrightarrow$ 3

原則として異なるステージの項は使えない

✗ λ`x:int.(▶(`λ`y:int.x+1))`

ステージを跨ぐ項のコードへの埋め込み(CSP)

簡約の順序を保ちつつ
低いステージの項をより高いステージの中に
埋め込むための機能

$$\lambda\, x\!:\!\mathtt{int}\ .(\blacktriangleright(\ \lambda y\!:\!\mathtt{int}.\%x{+}1\ ))$$

ステージを跨ぐ項のコードへの埋め込み(CSP)

簡約の順序を保ちつつ
低いステージの項をより高いステージの中に
埋め込むための機能

$$\lambda\ \underline{\mathtt{x:int}}\ .(\blacktriangleright(\ \lambda\mathtt{y:int.\%x+1}\ ))$$

x **はステージ** 0 **の項**

ステージを跨ぐ項のコードへの埋め込み(CSP)

簡約の順序を保ちつつ
低いステージの項をより高いステージの中に
埋め込むための機能

$\lambda\ \underline{\mathtt{x:int}}\ .(\blacktriangleright(\ \underline{\lambda\mathtt{y:int.\%x+1}}\ ))$

ここはステージ1なので%が必要

xはステージ0の項

λ^{MD} の型

型

$$\begin{aligned}
\tau \; ::= \;& \mathsf{X} && \text{(型レベル定数)} \\
\mid \;& \triangleright \, \tau && \text{(コード型)} \\
\mid \;& x : \tau \to \tau && \text{(依存関数型)} \\
\mid \;& \tau \; M && \text{(型の項への適用)}
\end{aligned}$$

λ^{MD} の型の例

依存関数型の例

```
(* 長さnのベクトルの各要素を2倍する関数 *)
val double: (n:int -> vec n -> vec n)
# double 3
- : vec 3 -> vec 3
```

コード型の例

```
# let double-code = ▶(%(double 3))
val double-code : ▷(vec 3 -> vec 3)
```

λ^{MD} における vadd-gen

ベクトルの長さ (n) を受け取って、
その長さのベクトルを足し合わせる関数を作る関数

```
let rec vadd-gen = λn:int.
  ▶ ( λv1:(vec   n).λv2:(vec   n). … )
```

λ^{MD} における vadd-gen

ベクトルの長さ (n) を受け取って、
その長さのベクトルを足し合わせる関数を作る関数

```
let rec vadd-gen = λn:int.
  ▶ ( λv1:(vec %n).λv2:(vec %n). … )
```

注意

長さ n はステージ 0 で定義されているので、
ステージ 1(▶ の中) で使うには CSP(%) が必要

vadd-gen と vadd-gen で生成した関数の型

```
val vadd-gen : n:int ->
      (▷(vec %n -> vec %n -> vec %n))
(* ▶と同様に▷の場合も%が必要 *)

let vadd3 = vadd-gen 3;;
val vadd3: ▷(vec %3 -> vec %3 -> vec %3)
```

生成した関数 (vadd3) が長さ 3 のベクトルだけを受け取ることが型から判断できる

vadd3 の技術的な問題

他の関数と組み合わせるのが難しい

2 つのベクトルを足し合わせて 2 倍する関数のコード

$$\texttt{val vadd3: } \triangleright(\texttt{vec \%3 -> vec \%3 -> vec \%3})$$

$$\texttt{val double-code : } \triangleright(\texttt{vec 3 -> vec 3})$$

$$\texttt{let add-double = } \blacktriangleright(\lambda \texttt{v:vec \%3.} \lambda \texttt{w:vec \%3.}$$
$$\blacktriangleleft \texttt{double-code} \; (\blacktriangleleft \texttt{vadd3 v w}))$$

vadd3の技術的な問題

他の関数と組み合わせるのが難しい

2つのベクトルを足し合わせて2倍する関数のコード

$\texttt{val vadd3: } \triangleright\texttt{(vec \%3 -> vec \%3 -> vec \%3)}$

$\texttt{val double-code : } \triangleright\texttt{(vec 3 -> vec 3)}$

✗ $\texttt{let add-double = } \blacktriangleright(\lambda\texttt{v:vec \%3.}\lambda\texttt{w:vec \%3.}$

✗ $\quad\blacktriangleleft\texttt{double-code (} \blacktriangleleft\texttt{vadd3 v w))}$

vadd3 の技術的な問題

他の関数と組み合わせるのが難しい

2 つのベクトルを足し合わせて 2 倍する関数のコード

```
  val vadd3: ▷(vec %3 -> vec %3 -> vec %3)
  val double-code : ▷(vec 3 -> vec 3)

✗ let add-double = ▶(λv:vec %3.λw:vec %3.
✗       ◀double-code ( ◀vadd3 v w ))
```

vec 3 -> vec 3

vadd3 の技術的な問題

他の関数と組み合わせるのが難しい

2 つのベクトルを足し合わせて 2 倍する関数のコード

```
  val vadd3: ▷(vec %3 -> vec %3 -> vec %3)
  val double-code : ▷(vec 3 -> vec 3)

✗ let add-double = ▶(λv:vec %3.λw:vec %3.
✗       ◀double-code ( ◀vadd3 v w ))
```

vec 3 -> vec 3

vec %3

vadd3 の技術的な問題

他の関数と組み合わせるのが難しい

2 つのベクトルを足し合わせて 2 倍する関数のコード

```
   val vadd3: ▷(vec %3 -> vec %3 -> vec %3)
   val double-code : ▷(vec 3 -> vec 3)

✗ let add-double = ▶(λv:vec %3.λw:vec %3.
✗      ◀double-code ( ◀vadd3 v w ))
```

`vec 3 -> vec 3`

`vec %3`

型が合わないので組み合わせることができない

解決策

項 M と 項 $\%M$ を以下の条件下で同一視する

- 条件: M が自由変数を含まない

同一視される型の例

- `vec %3` と `vec 3`
- $\triangleright$`(vec 3 -> vec 3 -> vec 3)`
  と $\triangleright$`(vec %3 -> vec %3 -> vec %3)`

解決策

項 M と 項 $\%M$ を以下の条件下で同一視する

- 条件: M が自由変数を含まない

同一視される型の例

- vec %3 と vec 3
- ▷(vec 3 -> vec 3 -> vec 3)
  と ▷(vec %3 -> vec %3 -> vec %3)

このような同一視をしても型安全性に
影響がないことを示した

λ^{MD} の形式化

- **項** M, **型** τ
- **簡約関係** $M \longrightarrow M'$

$$(\lambda x : \tau . M) N \longrightarrow_{\beta} M[x \mapsto N]$$
$$\blacktriangleleft_{\alpha}(\blacktriangleright_{\alpha} M) \longrightarrow_{\blacklozenge} M$$
$$(\Lambda \alpha . M)\ A \longrightarrow_{\Lambda} M[\alpha \mapsto A]$$

- **型判断** $\Gamma \vdash M : \tau @ A$

$$\frac{x : \tau @ A \in \Gamma}{\Gamma \vdash x : \tau @ A}\ (\textsc{T-Var}) \qquad \frac{\Gamma \vdash M : \tau @ A \qquad \Gamma \vdash \tau :: * @ A\alpha}{\Gamma \vdash \%_{\alpha} M : \tau @ A\alpha}\ (\textsc{T-Csp})$$

λ^{MD} の性質

定理 (型安全性)

項 M に型 τ がつく ($\Gamma \vdash M : \tau$) ならば M の簡約中に型エラーは起きない

定理 (強正規化性)

$\Gamma \vdash M : \tau$ ならば M から始まる無限簡約列はない

定理 (合流性)

$\Gamma \vdash M : \tau$ であり、M を簡約して 2 つの異なる項 M' と M'' を得たとき、適切に M' と M'' を簡約すれば同一の項 M''' を得る

目次

λ^{MD} の型推論アルゴリズム

型推論とは

- **入力**: λ^{MD} の項 M と型環境 Γ
- **出力**: τ $(\Gamma \vdash M : \tau)$
  `Fail` (そのような τ が存在しないとき)

λ^{MD} の型推論アルゴリズム

型推論とは

- **入力**: λ^{MD} の項 M と型環境 Γ
- **出力**: $\tau \qquad (\Gamma \vdash M : \tau)$
  `Fail` (そのような τ が存在しないとき)

通常の型判断 $\Gamma \vdash M : \tau$ の問題点

通常の型判断 $\Gamma \vdash M : \tau$ の導出規則からは
型推論アルゴリズムが構築できない

アルゴリズム的型判断

- 型推論アルゴリズムが停止する
  型判断 $\Gamma \mapsto M : \tau$ を新たに設計
- 更に以下を証明した
  - $\Gamma \mapsto M : \tau$ iff $\Gamma \vdash M : \tau$
- ここからアルゴリズム的型判断を元に設計した
  型推論アルゴリズムの正当性が導かれる

目次

1. 背景: 多段階計算とその問題点
2. 問題解決のためのアイデア
3. 本研究について
   3.1. 依存型付多段階計算体系 λ^{MD}
   3.2. λ^{MD} の型推論アルゴリズム
4. まとめ

関連研究

- $\lambda^{\triangleright\%}$ [Hanada&Igarashi'14]
  - **本研究の拡張元**
  - CSP **を含む多段階計算体系**
- Concoqtion [Forgarty et al.'07]
  - MetaOCaml **に制限された形の依存型を導入**
- $\lambda_{H\circ}$ [Pasalic'04]
  - **依存型付きメタプログラミング言語**
  - **コードの評価と** CSP **がない**

結論

多段階計算に依存型を導入した

- 生成されたコードの不正な使用を依存型で防止
- λ^{MD} の形式化
- 型安全性を証明
- 型推論のアルゴリズムを設計

なぜ最初から$\Gamma \vdash\!\!\blacktriangleright M : \tau$でやらないのか?

$\Gamma \vdash M : \tau$ と $\Gamma \vdash\!\!\blacktriangleright M : \tau$ の比較

	$\Gamma \vdash M : \tau$	$\Gamma \vdash\!\!\blacktriangleright M : \tau$
型安全性の証明	**容易**	**困難**
型推論アルゴリズム	**設計不可能**	**設計可能**

なぜ $\Gamma \vdash M : \tau$ の導出規則から型推論アルゴリズムを設計できないのか?

構文主導 (Syntax-directed) **でない導出規則があるから**

$$\frac{\Gamma \vdash M : \underline{\tau}@A \qquad \Gamma \vdash \underline{\tau} \equiv \sigma :: K@A}{\Gamma \vdash M : \sigma@A}\ (\textsc{T-Conv})$$