多段階計算の型システムの基礎

型システム祭りオンライン

2020 年 4 月 17 日

T. Suwa (@bd_gfngfn)

概要

- **多段階計算 (*multi-stage programming*)** とは：
  - 動的にコードを生成してそれを走らせる機構を備えた，
    計算が複数の**ステージ**からなる意味論をもつ体系

    ＋**それを安全に行なうための型システム**
  - メタプログラミングの一種
    - 例： λ○ [Davies 1996]， MetaML [Taha & Sheard 1997]， etc.
- 発表のねらい：
  - 多段階計算の意味論と型システムについて簡単に紹介し，
    直観を獲得してもらう

概要

- あるとうれしい前提知識：
  - **単純型つき λ 計算**のごく基礎
    - λ 項の構文や β 簡約を知っている,
    - 型判定の導出規則を読んだことがある, etc.
  - OCaml 風の構文が読める

- **多段階計算を考える動機**
- 構文と操作的意味論
- 型システム
- まとめ

一般のメタプログラミングの動機

- 計算効率上のパフォーマンス改善
  - “最終的な実行時のオーバーヘッドを減らすために
    事前に計算できるところを計算しておける仕組み” がほしい
  - 部分評価 (partial evaluation) と関連
- マクロ機構として使う
  - ボイラープレートコードを減らして保守性を高めるために
    マクロが書ける仕組みがほしい

多段階計算のありがたみ

“**前処理結果が安全**” であることが事前に保証できる

マクロ機構としてみたとき：

- “マクロの定義そのもの” が型検査される
- “**マクロ定義が型検査に通れば，そのマクロの適用を展開した後のプログラムにも必ず型がつく**” ことが保証されている

---

cf. 他のメタプログラミング的手法：

- トークン列の置換（例： C言語の `#define` ）
- ADT による抽象構文木の取扱い（例： PPX, Template Haskell）

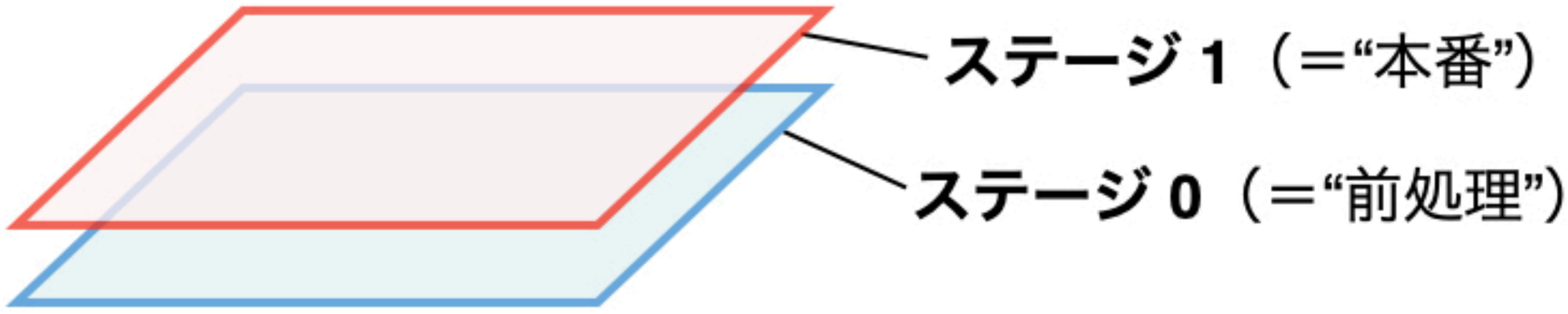
いずれも展開結果の安全性は保証しない

- 多段階計算を考える動機
- **構文と操作的意味論**
- 型システム
- まとめ

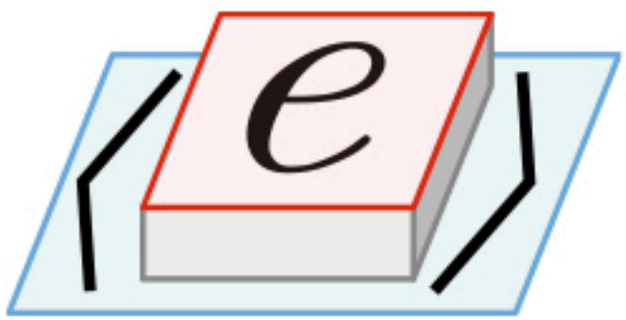
構文

$$
\begin{array}{lll}
e ::= x & \text{変数} & \left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{最も基本的な構成要素} \\
\quad | \ e\ e & \text{適用} & \\
\quad | \ \boldsymbol{\lambda} x.\ e & \lambda\text{抽象} & \\
\quad | \ \mathbf{fix}\ e & \text{不動点} & \left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{型つき・値呼びの}\lambda\text{計算では実用上必要} \\
\quad | \ \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \text{条件分岐} & \\
\quad | \ \langle e \rangle & \mathbf{bracket} & \left.\begin{array}{l}\\ \\ \end{array}\right\} \mathbf{多段階計算特有の構成要素} \\
\quad | \ {\sim} e & \mathbf{escape} &
\end{array}
$$

bracket と escape の直観



bracket：$\langle e \rangle$ は 山



“次の（＝上の）ステージの
コードをつくる”

escape：$\sim e$ は 穴

“下のステージで e が評価されて
コードになり穴が埋まる”

操作的意味の直観

特にステージが 2 つの場合：

- ステージ 0 の部分だけが通常の β 簡約などで評価される
  - したがってコードになる部分は穴の内側が簡約される
- 穴の内側が山だけになったら相殺

- 全体が穴のないコードになったら “前処理” の終了に相当し，
  完成したコードをステージのない普通のプログラムとして使う

計算の例

自然数 m を与えると「m 乗函数のコード」を返す函数

使用例：

```
let cubic = ~(genpower 3) in …
```

cf. 通常の累乗函数の部分適用による実装：

```
let cubic = power 3 in …
```

これだと cubic の適用のたびに再帰が走って計算コストが高い

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
```

ステージ 1 での変数名は
“束縛箇所を最初に読むとき” に
freshに生成
（これはいわゆる衛生性のため）

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  ⟶*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            ⟶*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
```

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
```

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
```

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~⟨1⟩⟩⟩⟩
```

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~⟨1⟩⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * 1⟩⟩⟩
```

計算の例： genpower の実装

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

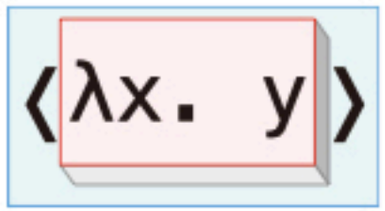
```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~⟨1⟩⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * 1⟩⟩⟩
            →*  ⟨λa. a * (a * 1)⟩
```

- 多段階計算を考える動機
- 構文と操作的意味論
- **型システム**
- まとめ

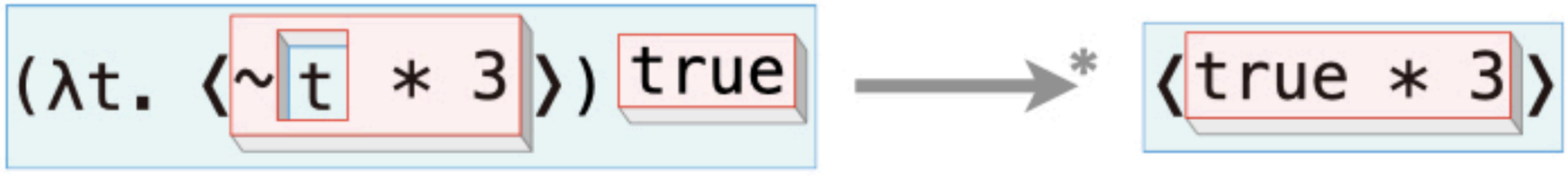
型システムで弾きたい状況

特に “**前処理の過程では失敗しないが生成されたコードがおかしい**”
という状況が望ましくない：

- 生成されたコードに束縛されていない変数がある
  (※ 計算途中のコード片に自由変数が出現することは無問題)



- 生成されたコードの型が合っていない



どんな型をつけるか

- 基本的には“上のステージで τ 型の式となるコードの型”として $\langle \tau \rangle$ を追加するだけ：

$$\tau ::= b \mid \tau \to \tau \mid \langle \tau \rangle$$
$$b ::= \texttt{bool} \mid \texttt{int} \mid \cdots$$

  - 裏を返せば，型の拡張を筋良く済ませるために前述のように構文と操作的意味論を定めている
  - let 多相などにも拡張できるが簡単のため省略

例：

$\texttt{genpower} : \texttt{int} \to \langle \texttt{int} \to \texttt{int} \rangle$

自然数 m を与えると「m 乗函数のコード」を返す函数

型つけ規則（抜粋）

簡単のためステージが 2 つの場合だけ考える

$$\Gamma \vdash^{0} e : \tau \qquad \Gamma \vdash^{1} e : \tau$$

ステージの上下を跨ぐ規則は以下の 2 つ：

$$\frac{\Gamma \vdash^{1} e : \tau}{\Gamma \vdash^{0} \langle e \rangle : \langle \tau \rangle} \qquad \frac{\Gamma \vdash^{0} e : \langle \tau \rangle}{\Gamma \vdash^{1} \sim e : \tau}$$

“ステージ 0 から見て
山の部分はコードになる”

“ステージ 1 から見て穴の部分は
下でコードが生成されねばならない”

型つけ規則（抜粋）

$$\frac{\Gamma[x \mapsto \tau^n] \vdash^n e : \tau'}{\Gamma \vdash^n (\lambda x.\ e) : \tau \to \tau'}$$

型環境は，変数に型だけでなく
どのステージで束縛されたかの
情報も紐づける

$$\frac{\Gamma(x) = \tau^k \quad k = n}{\Gamma \vdash^n x : \tau}$$

変数が出現できるのは
それが束縛されたステージに限る

（※ $k \leq n$ とする流儀もあり，
cross-stage persistence と呼ばれる）

型安全性

- 保存　$\Gamma \vdash^n e : \tau$ かつ $e \xrightarrow{n} e'$ ならば $\Gamma \vdash^n e' : \tau$
- 進行　$\varnothing \vdash^n e : \tau$ ならば e は値または $\exists e.\ e \xrightarrow{n} e'$

系として

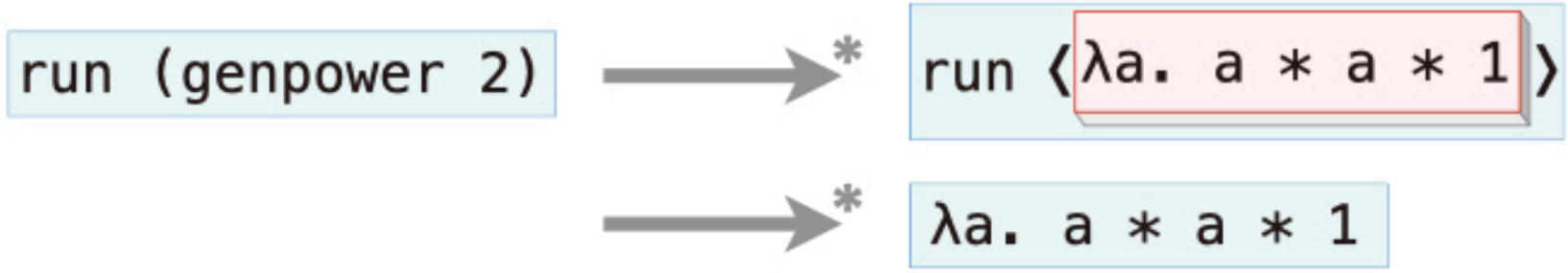
穴のないコード

$\varnothing \vdash^0 e : \langle \tau \rangle$ かつ $e \xrightarrow{0} \langle e' \rangle \not\to$ ならば $\varnothing \vdash^1 e' : \tau$

つまり生成されるコードは必ず型がつくと言える

さらなる拡張

- run primitive
  - 上のステージのコードを下のステージで使う機構



- mutable reference や限定継続との共存

いずれもコード中の変数がスコープを脱出するおそれがあり,
型システムに工夫が必要

cf. $\lambda^{\triangleright}$ [Tsukada & Igarashi 2010], $\lambda^{\text{poly}}_{\text{open}}$ [Kim, Yi & Calcagno 2006],
〈NJ〉 [Kiselyov, Kameyama & Sudo 2016]

ちなみに

- 拙作の “函数型組版処理システム” である SATySFi にも多段階計算の型システムが入っています
  - 前処理用マクロのために導入したもの
  - まだまだドキュメントが整備できていませんが既に使えます
  - 使用例：
    - その正規表現エンジン，インタプリタで満足してる？！ [κeen 2019]
    - SATySFiコード中で整数を16進数で書きたい [@zr_tex8r 2019]

- 多段階計算を考える動機
- 構文と操作的意味論
- 型システム
- **まとめ**

まとめ

- 多段階計算はメタプログラミングの一種で,
  “前処理” で生成されるコードが安全であることが型システムによって保証されているのが特徴

- 今回は多段階計算の構文・意味論・型システムのごく基礎を紹介しました