新春最適化手法解説: *Tail Modulo Cons*

@第五回関数型プログラミング（仮）の会

びしょ〜じょ

IAM...

こんにちは、びしょ〜じょです

🏢 株式会社カンム
Golangerしてます

🪁 すきなもの
代数的効果、継続、処理系

👤

Nymphium
Nymphium

突然ですが

突然ですが

関数型プログラミングといえば…?

突然ですが

関数型プログラミングといえば…?

再帰でしょ

本日の内容

本日の内容

OCaml(≧ 4.14)で

本日の内容

OCaml(≧ 4.14)で

再帰関数に

本日の内容

OCaml(≧ 4.14)で

再帰関数に

`[@tail_mod_cons]`

と書くと

本日の内容

OCaml(≧ 4.14)で

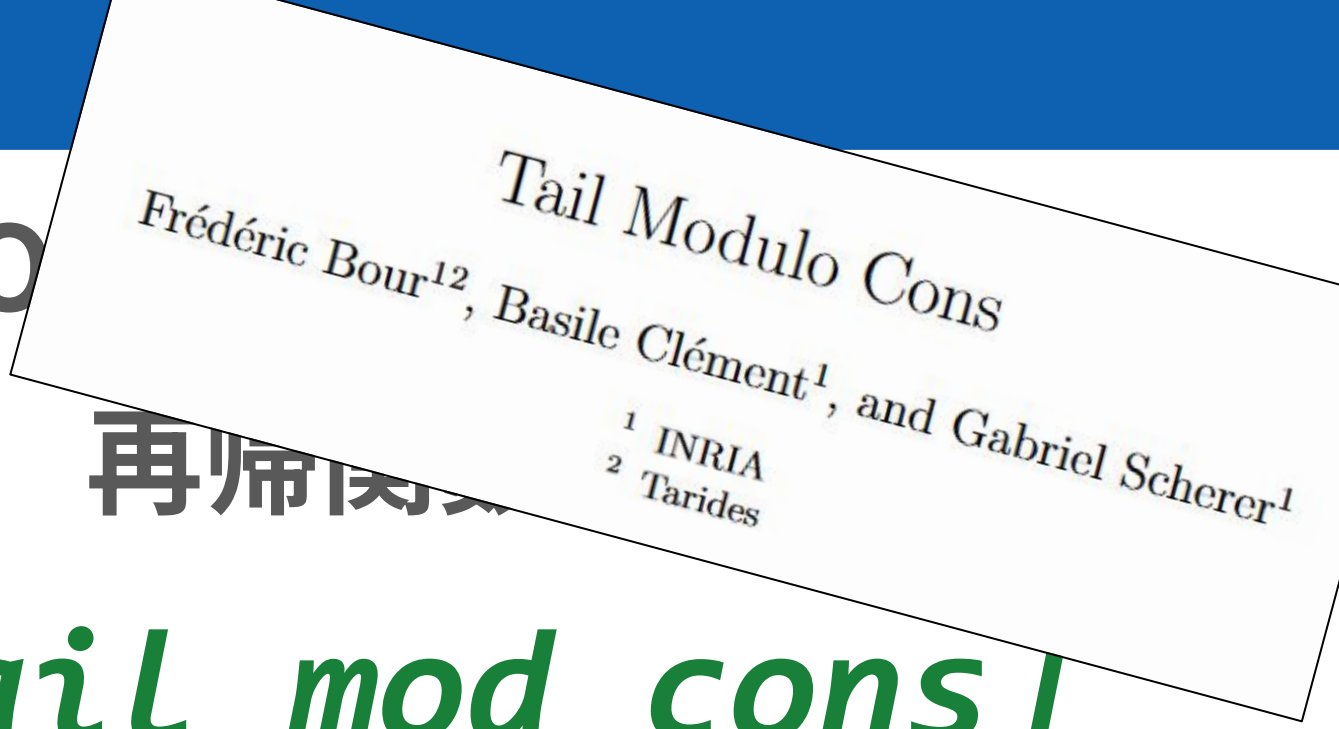
再帰関数に

`[@tail_mod_cons]`

と書くと

速くなる 場合がある

本日の内容


Tail Modulo Cons

Frédéric Bour[1,2], Basile Clément[1], and Gabriel Scherer[1]

[1] INRIA
[2] Tarides

再帰関数

`[@tail_mod_cons]`

と書くと

速くなる 場合がある

再帰

最も基本的な構造

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

再帰

最も基本的な構造

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

定義に自身を参照

型、関数…

末尾再帰

関数の末尾位置で自身を再帰呼び出し

```
let rec fold_left f x = function
    | [] -> x
    | x' :: xs' -> fold_left f (f x x') xs'
```

末尾再帰

関数の末尾位置で自身を再帰呼び出し

```
let rec fold_left f x = function
    | [] -> x
    | x' :: xs' -> fold_left f (f x x') xs'
```

末尾位置

末尾再帰

関数の末尾位置で自身を再帰呼び出し

```
let rec fold_left f x = function
    | [] -> x
    | x' :: xs' -> fold_left f (f x x') xs'
```

末尾位置

再帰呼び出し

末尾位置+再帰で

末尾再帰

末尾呼び出し最適化

末尾再帰にすると**末尾呼び出し最適化**ができる

- 再帰呼び出し(call)をjumpに置き換えられ
- コールスタックの操作が不要になりperformant
- そしてスタックオーバーフローもしない!

末尾呼び出し最適化

末尾再帰にすると**末尾呼び出し最適化**ができる

- 再帰呼び出し(call)をjumpに置き換えられ
- コールスタックの操作が不要になりperformant
- そしてスタックオーバーフローもしない!

ところで

mapって…末尾再帰じゃない

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

mapって…末尾再帰じゃない

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

A正規化すると…

```
let x' = f x in
let xs'' = map f xs' in
x' :: xs''
```

後続処理があって
無理や…

mapを救いたい

どうする…?

mapを救いたい

どうする…?

- map_revしてrev
  - performantでないケース
- 継続渡し方式(CPS)
  - 手で書くのですか…?

mapを救いたい

どうする…?

- map_revしてrev
  - performantでないケース
- 継続渡し方式(CPS)
  - 手で書くのですか…?

- ***Tail-Recursion Modulo Cons***

Tail-Recursion Modulo Cons

Tail-Recursion Modulo Cons とは…

Tail-Recursion Modulo Cons

Tail-Recursion Modulo Cons とは…

末尾再帰

Tail-Recursion Modulo Cons

Tail-Recursion Modulo Cons とは…

末尾再帰

＋

Destination Passing Style

(*DPS*)

Tail-Recursion Modulo Cons

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

Tail-Recursion Modulo Cons

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
  | x :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

Tail-Recursion Modulo Cons

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
  | x :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

Tail-Recursion Modulo Cons

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
    :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

プレースホルダー

Tail-Recursion Modulo Cons

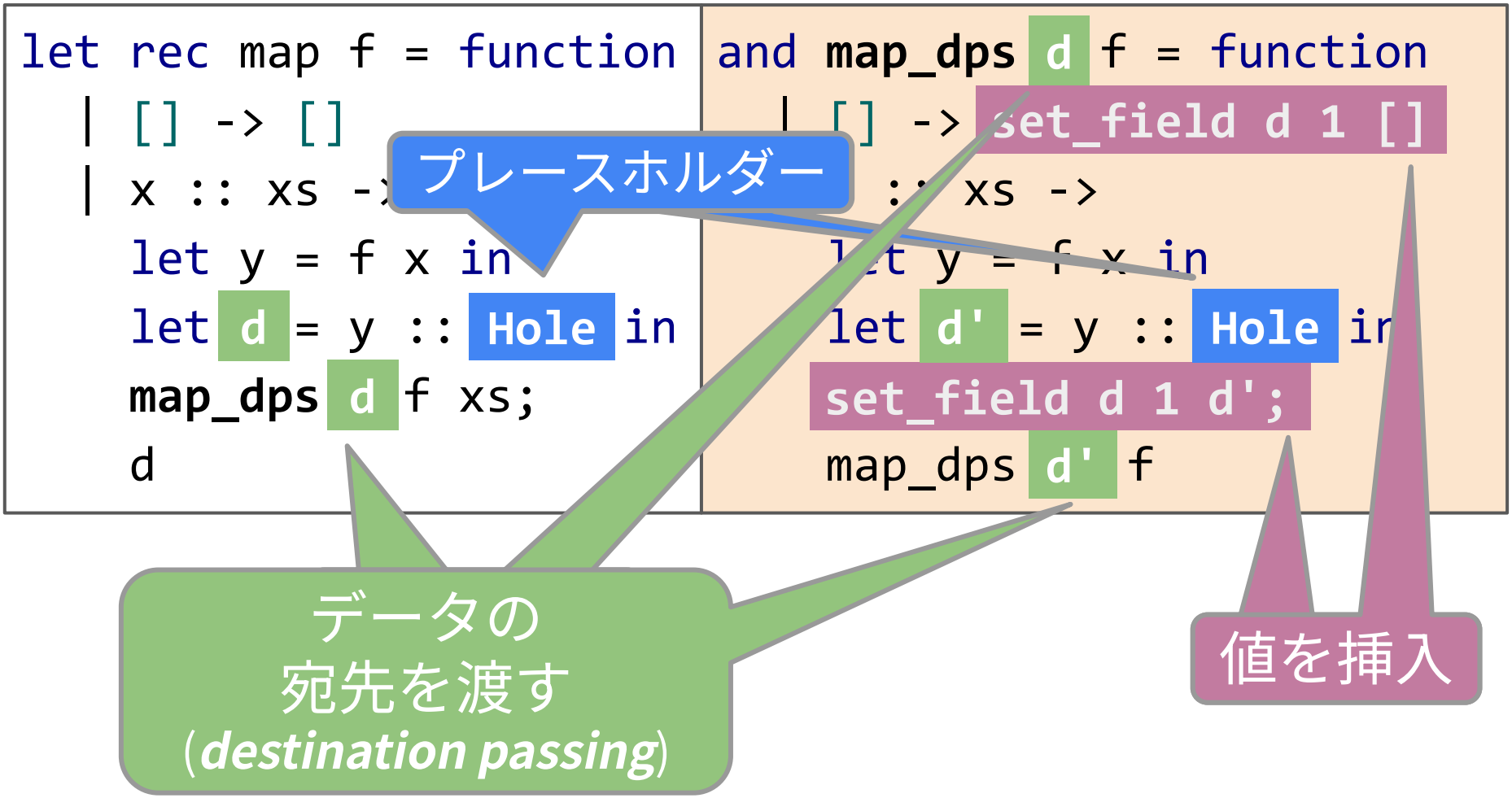
```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
  | x :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

プレースホルダー

データの
宛先を渡す
(*destination passing*)

Tail-Recursion Modulo Cons



Tail-Recursion Modulo Cons

自身は再帰してない

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
  | x :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

プレースホルダー

データの
宛先を渡す
(*destination passing*)

値を挿入

Tail-Recursion Modulo Cons

自身は再帰してない

末尾再帰してる!!

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
  | x :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

プレースホルダー

データの
宛先を渡す
(*destination passing*)

値を挿入

[@tail_mod_cons]

Tail recursion modulo consは古典的な手法

[@tail_mod_cons]

Tail recursion modulo consは古典的な手法

- Lisp(1970〜)、Prolog、etc.
- しかし手書き… 破壊的代入危ない☠️

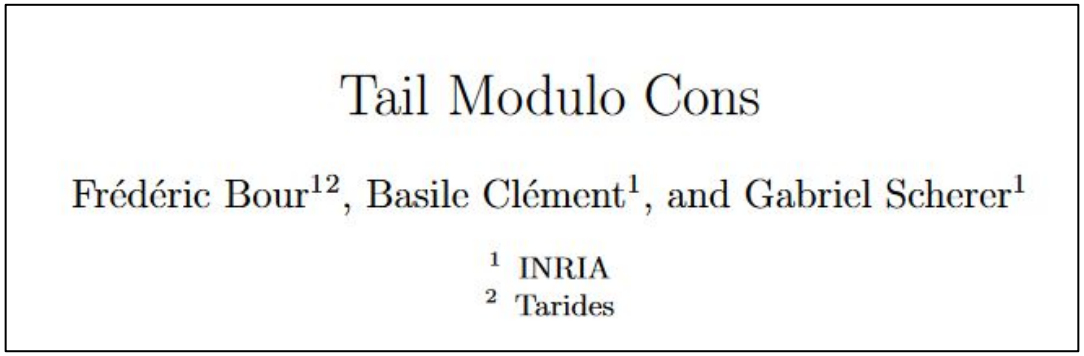
[@tail_mod_cons]

Tail recursion modulo consは古典的な手法

- Lisp(1970〜)、Prolog、etc.
- しかし手書き… 破壊的代入危ない☠️

↓

定式化することで
プログラム変換として
コンパイラに導入😁


Tail Modulo Cons

Frédéric Bour[1,2], Basile Clément[1], and Gabriel Scherer[1]

[1] INRIA
[2] Tarides

以降、式はこの論文より引用

[@tail_mod_cons]

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

[@tail_mod_cons]

```
let[@tail_mod_cons]
  rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

アノテーション
つけるだけで

[@tail_mod_cons]

```
let[@tail_mod_cons]
  rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

アノテーション
つけるだけで

プログラム変換!!

(コンパイラの中間言語)

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let y = f x in
    let d = y :: Hole in
    map_dps d f xs;
    d
```

```
and map_dps d f = function
  | [] -> set_field d 1 []
  | x :: xs ->
    let y = f x in
    let d' = y :: Hole in
    set_field d 1 d';
    map_dps d' f
```

Tail Modulo Cons Context

小さな対象言語

$$
\begin{aligned}
\text{Exprs} \ni e, d ::= {} & x, y \\
& \mid n \in \mathbb{N} \\
& \mid f\ e \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\
& \mid K\,(e_i)^i \\
& \mid \texttt{match}\ e\ \texttt{with}\ (p_i \to e'_i)^i \\
& \mid d.e \leftarrow e'
\end{aligned}
$$

$$
\text{FunctionNames} \ni f
$$

$$
\text{Patterns} \ni p ::= x \mid K\,(p_i)^i
$$

$$
\text{Stmt} \ni s ::= \texttt{let rec}\ (f_i\ x = e_i)^i
$$

Tail Modulo Cons Context

小さな対象言語

$$
\begin{aligned}
\text{Exprs} \ni e, d ::= {} & x, y \\
& \mid n \in \mathbb{N} \\
& \mid f\ e \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\
& \mid K\,(e_i)^i \\
& \mid \texttt{match}\ e\ \texttt{with}\ (p_i \to e_i')^i \\
& \mid d.e \leftarrow e'
\end{aligned}
$$

$$
\begin{aligned}
\text{FunctionNames} \ni f & \\
\text{Patterns} \ni p ::= {} & x \mid K\,(p_i)^i \\
\text{Stmt} \ni s ::= {} & \texttt{let rec}\ (f_i\ x = e_i)^i
\end{aligned}
$$

関数定義は
トップレベルのみ

- let式
- 関数適用
- 代数的データ型
- パターンマッチング
- 代入

Tail Modulo Cons Context

ConstrCtx $\ni C\,[\square]$ ::= $\square$
$\mid K\,((e_i)^i, C\,[\square]\,, (e_j)^j)$

TailCtx $\ni T$ ::=
$\mid \square$
$\mid e; T$
$\mid \texttt{let}\ x = e\ \texttt{in}\ T$
$\mid \texttt{match}\ e\ \texttt{with}\ (p_j \to T_j)^j$

TailModConsCtx $\ni U$ ::=
$\mid \square$
$\mid e; U$
$\mid \texttt{let}\ x = e\ \texttt{in}\ U$
$\mid \texttt{match}\ e\ \texttt{with}\ (p_j \to U)^j$
$\mid K\,((e_i)^i, U, (e_j)^j)$

Tail Modulo Cons Context

ConstrCtx $\ni C\,[\square] ::= \square \mid K\,((e_i)^i, C\,[\square]\,, (e_j)^j)$

TailCtx $\ni T ::=$
$\mid \square$
$\mid e; T$
$\mid \texttt{let}\ x = e\ \texttt{in}\ T$
$\mid \texttt{match}\ e\ \texttt{with}\ (p_j \to T_j)^j$

TailModConsCtx $\ni U ::=$
$\mid \square$
$\mid e; U$
$\mid \texttt{let}\ x = e\ \texttt{in}\ U$
$\mid \texttt{match}\ e\ \texttt{with}\ (p_j \to U)^j$
$\mid K\,((e_i)^i, U, (e_j)^j)$

コンストラクタに潜っていく ("modulo")

Tail Modulo Cons Context

ConstrCtx $\ni C\,[\square]$ ::= $\square$
$\mid K\,((e_i)^i, C\,[\square], (e_j)^j)$

TailCtx $\ni T$ ::=
$\mid \square$
$\mid e; T$
$\mid$ `let` $x = e$ `in` T
$\mid$ `match` e `with` $(p_j \to T_j)^j$

TailModConsCtx $\ni U$ ::=
$\mid \square$
$\mid e; U$
$\mid$ `let` $x = e$ `in` U
$\mid$ `match` e `with` $(p_j \to U)^j$
$\mid K\,((e_i)^i, U, (e_j)^j)$

コンストラクタに潜っていく ("modulo")

例: `f x :: map f xs`

- $U =$ `f x ::` $\square$
- $\square =$ `map f xs`

TMC変換

$$\frac{}{d.n \leftarrow \square \rightsquigarrow_{dps} \square\,[d.n \leftarrow \square]} \qquad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \square]^i}{d.n \leftarrow \texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{dps} \texttt{let}\ x = e\ \texttt{in}\ T[d_i.n_i \leftarrow \square]^i}$$

$$\frac{\forall j, \quad d.n \leftarrow U_j \rightsquigarrow_{dps} T_j\left[d_{i_j}.n_{i_j} \leftarrow \square\right]^{i_j}}{d.n \leftarrow \texttt{match}\ e\ \texttt{with}\ (p_j \rightarrow U_j)^j \rightsquigarrow_{dps} \texttt{match}\ e\ \texttt{with}\ \left(p_j \rightarrow T_j\left[d_{i_j}.n_{i_j} \leftarrow \square\right]^{i_j}\right)^j}$$

$$\frac{n' = |I| + 1 \qquad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \square]^l}{d.n \leftarrow K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{dps} \begin{array}{l}\texttt{let}\ d' = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ \quad d.n \leftarrow d'; \\ \quad T[d'_l.n'_l \leftarrow \square]^l\end{array}}$$

$$\frac{}{\square \rightsquigarrow_{direct} \square} \qquad \frac{U \rightsquigarrow_{direct} E}{\texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{direct} (\texttt{let}\ x = e\ \texttt{in}\ E)}$$

$$\frac{\forall j,\ U_j \rightsquigarrow_{direct} E_j}{\texttt{match}\ e\ \texttt{with}\ (p_j \rightarrow U_j)^j \rightsquigarrow_{direct} (\texttt{match}\ e\ \texttt{with}\ (p_j \rightarrow E_j)^j)}$$

$$\frac{n = |I| + 1 \qquad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \square]^l}{K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{direct} \begin{array}{l}\texttt{let}\ d = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ \quad T[d_l.n_l \leftarrow \square]^l; \\ \quad d\end{array}}$$

TMC変換

$$\frac{}{d.n \leftarrow \Box \rightsquigarrow_{dps} \Box\,[d.n \leftarrow \Box]} \qquad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \Box]^i}{d.n \leftarrow \texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{dps} \texttt{let}\ x = e\ \texttt{in}\ T[d_i.n_i \leftarrow \Box]^i}$$

$$\frac{\forall j, \quad d.n \leftarrow U_j \rightsquigarrow_{dps} T_j\left[d_{i_j}.n_{i_j} \leftarrow \Box\right]^{i_j}}{d.n \leftarrow \texttt{match}\ e\ \texttt{with}\ (p_j \rightarrow U_j)^j \rightsquigarrow_{dps} \texttt{match}\ e\ \texttt{with}\ \left(p_j \rightarrow T_j\left[d_{i_j}.n_{i_j} \leftarrow \Box\right]^{i_j}\right)^j}$$

$$\frac{n' = |I| + 1 \qquad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \Box]^l}{d.n \leftarrow K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{dps} \begin{array}{l}\texttt{let}\ d' = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ \quad d.n \leftarrow d'; \\ \quad T[d'_l.n'_l \leftarrow \Box]^l\end{array}}$$

$$\frac{}{\Box \rightsquigarrow_{direct} \Box} \qquad \frac{U \rightsquigarrow_{direct} E}{\texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{direct} (\texttt{let}\ x = e\ \texttt{in}\ E)}$$

$$\frac{\forall j,\ U_j \rightsquigarrow_{direct} E_j}{\texttt{match}\ e\ \texttt{with}\ (p_j \rightarrow U_j)^j \rightsquigarrow_{direct} (\texttt{match}\ e\ \texttt{with}\ (p_j \rightarrow E_j)^j)}$$

$$\frac{n = |I| + 1 \qquad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \Box]^l}{K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{direct} \begin{array}{l}\texttt{let}\ d = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ \quad T[d_l.n_l \leftarrow \Box]^l; \\ \quad d\end{array}}$$

え〜、つまり?
どういうことですか?
3行で頼む

TMC変換

TMC変換

1. 対象の関数からDPS versionの関数を作る

```
map f xs
```

```
map_dps d idx f xs
```

ここで、

```
set_field d idx (map f xs) = map_dps d idx f xs
```

TMC変換

1. 対象の関数からDPS versionの関数を作る

```
map f xs
```

```
map_dps d idx f xs
```

ここで、

```
set_field d idx (map f xs) = map_dps d idx f xs
```

2. 2つの変換

- $\rightsquigarrow_{direct}$でmapの再帰をmap_dpsの呼び出しに
- $\rightsquigarrow_{dps}$　でmapに基づいたmap_dpsの生成

TMC変換

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    f x :: map f xs
```

TMC変換

dps版を作成
bodyをコピー

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    f x :: map f xs
```

```
and map_dps d i f = function
  | [] -> []
  | x :: xs ->
   f x :: map f xs
```

TMC変換

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let d = f x :: Hole in
    map_dps d 1 f;
    d
```

```
and map_dps d i f = function
  | [] -> []
  | x :: xs ->
    f x :: map f xs
```

dps版を作成
bodyをコピー

コンストラクタに潜る

```
f x :: map f xs
```

dps版を呼ぶ

TMC変換

変換の終端
フィールドにセット

dps版を作成
bodyをコピー

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let d = f x :: Hole in
    map_dps d 1 f;
    d
```

```
and map_dps d i f = function
  | [] -> set_field d i []
  | x :: xs ->
    f x :: map f xs
```

コンストラクタに潜る

```
f x :: map f xs
```

dps版を呼ぶ

TMC変換

変換の終端
フィールドにセット

dps版を作成
bodyをコピー

```
let rec map f = function
  | [] -> []
  | x :: xs ->
    let d = f x :: Hole in
    map_dps d 1 f;
    d
```

```
and map_dps d i f = function
  | [] -> set_field d i []
  | x :: xs ->
    let d' = f x :: Hole in
    set_field d i d';
    map_dps d' 1 f
```

コンストラクタに潜る

```
f x :: map f xs
```

dps版を呼ぶ

コンストラクタに潜る
1つ前のフィールドにセット
末尾再帰

misc

- OCamlの標準ライブラリでも利用が進んでいる

  - containers、baseなどでも普及

- 最近の関連研究オモロイ(紹介論文は2021)
  - *Tail Recursion Modulo Context*
    - consの一般化、諸問題にtackle
  - *Destination-passing style programming: a Haskell implementation*
    - DPS+線形型、結局手で書きたいパターンも安全に

まとめ

- Tail recursion modulo consという技法
  - 非末尾再帰→末尾再帰
  - 手で書くと面倒だし破壊的代入がある
- OCamlなら`[@tail_mod_cons]`でOK!
- つまり

OCamlを書こう!!!!!

  - 公式も読むと良いですよ
    - https://v2.ocaml.org/manual/tail_mod_cons.html

Reference

- 📝Bour, Frédéric, Basile Clément, and Gabriel Scherer. "Tail modulo cons." *arXiv preprint arXiv:2102.09823* (2021).
- 📝Bagrel, Thomas. "Destination-passing style programming: a Haskell implementation." *arXiv preprint arXiv:2312.11257* (2023).
- 📝Leijen, Daan, and Anton Lorenzen. "Tail Recursion Modulo Context: An Equational Approach." *Proceedings of the ACM on Programming Languages* 7, no. POPL (2023): 1152-1181.

TMC変換

$$\frac{}{\square \rightsquigarrow_{direct} \square} \qquad \frac{U \rightsquigarrow_{direct} E}{\texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{direct} (\texttt{let}\ x = e\ \texttt{in}\ E)}$$

$$\frac{\forall j,\ U_j \rightsquigarrow_{direct} E_j}{\texttt{match}\ e\ \texttt{with}\ (p_j \to U_j)^j \rightsquigarrow_{direct} (\texttt{match}\ e\ \texttt{with}\ (p_j \to E_j)^j)}$$

$$\frac{n = |I| + 1 \qquad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \square]^l}{K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{direct} \begin{array}{l}\texttt{let}\ d = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ T[d_l.n_l \leftarrow \square]^l; \\ d\end{array}}$$

equationでdpsの呼び出しに変換

$$\frac{}{d.n \leftarrow \square \rightsquigarrow_{dps} \square\,[d.n \leftarrow \square]} \qquad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \square]^i}{d.n \leftarrow \texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{dps} \texttt{let}\ x = e\ \texttt{in}\ T[d_i.n_i \leftarrow \square]^i}$$

$$\frac{\forall j, \quad d.n \leftarrow U_j \rightsquigarrow_{dps} T_j\left[d_{i_j}.n_{i_j} \leftarrow \square\right]^{i_j}}{d.n \leftarrow \texttt{match}\ e\ \texttt{with}\ (p_j \to U_j)^j \rightsquigarrow_{dps} \texttt{match}\ e\ \texttt{with}\ \left(p_j \to T_j\left[d_{i_j}.n_{i_j} \leftarrow \square\right]^{i_j}\right)^j}$$

$$\frac{n' = |I| + 1 \qquad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \square]^l}{d.n \leftarrow K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{dps} \begin{array}{l}\texttt{let}\ d' = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ d.n \leftarrow d'; \\ T[d'_l.n'_l \leftarrow \square]^l\end{array}}$$

equationで再帰呼び出しに変換

末尾呼び出し最適化

Q. なぜ**末尾再帰**は**performant**なんですか?

A. **末尾呼び出し最適化**ができるから

Q. 末尾呼び出し最適化とはなんですか?

A. 再帰呼び出し(`call`)を`jump`にする(続)

末尾呼び出し最適化

再帰呼び出し(call)をjumpにする

```
fold_left:
  ......
  call fold_left
```

後続の処理がない!

```
fold_left:
  ......
  jmp fold_left
```

エントリの先頭までjmpすれば
よくないですか?