# 計算効果の推論技術

関山 太朗

国立情報学研究所

# 自己紹介

- 略歴
  - 2016　　京都大学 情報学研究科 博士（情報学）
  - 2016-2018　IBM 東京基礎研究所
  - 2018-現在　国立情報学研究所 助教

- 研究分野
  - プログラミング言語理論
  - 型理論
  - プログラム検証（と機械学習の融合）

# 計算効果の「推論」

**プログラムがどんな計算効果を、どのように使っているかを推論する**

例えばランダムな文字列を出力するプログラムの場合

```
int main() {
  size_t s = sizeof(char) * 10;
  while (true) {
    char* x = malloc(s);
    for (int i = 0; i < 10; ++i)
      x[i] = 'A' + (rand() % 26);
    printf("%s", x);
    free(x);
}
```

# 何故「推論」したい？

- **プログラムが意図通り動作することを確認するため**
- 例えば次のことが確認できる
  - **プログラムが計算効果を正しく使っているか？**
  - **プログラムが意図しない計算効果を使っていないか？**

# 計算効果の正しい使い方・間違った使い方

例：C 言語のポインタ操作

```
int main() {
  size_t s = sizeof(char) * 10;
  while (true) {
    char* x = malloc(s);
    for (int i = 0; i < 10; ++i)
      x[i] = 'A' + (rand() % 26);
    printf("%s", x);
    free(x);
}
```

```
int main() {
  size_t s = sizeof(char) * 10;
  while (true) {
    char* x; // メモリ割当忘れ
    for (int i = 0; i < 10; ++i)
      x[i] = 'A' + (rand() % 26);
    printf("%s", x);
    free(x);
}
```

# 計算効果の正しい使い方・間違った使い方

例：C 言語のポインタ操作

```
int main() {
  size_t s = sizeof(char) * 10;
  while (true) {
    char* x = malloc(s);
    for (int i = 0; i < 10; ++i)
      x[i] = 'A' + (rand() % 26);
    printf("%s", x);
    free(x);
}
```

```
int main() {
  size_t s = sizeof(char) * 10;
  while (true) {
    char* x = malloc(s);
    for (int i = 0; i < 10; ++i)
      x[i] = 'A' + (rand() % 26);
    printf("%s", x);
    // 解放し忘れ
}
```

# 意図しない計算効果の使用

例：デバッグ・テスト用コードが残っている

```
int main() {
  size_t s = sizeof(char) * 10;
  srand(42); // テスト用に乱数のシードを固定
  while (true) {
    char* x = malloc(s);
    for (int i = 0; i < 10; ++i)
      x[i] = 'A' + (rand() % 26);
    smart_printf("%s", x);
    free(x);
  }
}
```

# 意図しない計算効果の使用

例：悪意のあるライブラリの利用

```
#include <smart_printf.h>
int main() {
  size_t s = sizeof(char) * 10;
  while (true) {
    char* x = malloc(s);
    for (int i = 0; i < 10; ++i
      x[i] = 'A' + (rand() %
    smart_printf(x);
    free(x);
  }
}
```

```
void smart_printf(…) {
  … // 引数を別サーバーに送信
}
```

# 本講演

計算効果が正しく使われていることを確認するための
**形式的推論技術**を**「ざっくりと」**紹介します

# 形式的推論

- 数学的に意味付けされた規則に則りプログラムの性質を「証明」
  - 推論の正しさが数学的に保証されている
  - 数学の表現能力を使った説明が可能
    - 「関数 sum は任意の引数 $n \geq 0$ について $\Sigma_{i=0}^{n}$ を返す」など
- 今日はあまり深入りしません

```
int sum(int n) {
  int x = n, y = 0;
  while (x != 0)
  { y = y + x; x = x - 1; }
  return y; }
```

プログラム

$$\dfrac{n \geq 0 \Longrightarrow n \geq 0 \wedge n = n \wedge 0 = 0 \quad \dfrac{\vdots}{\{n \geq 0 \wedge n = n \wedge 0 = 0\}\ x := \mathrm{n}; \mathrm{y} := 0\ \{n \geq 0 \wedge x = n \wedge y = 0\}} \quad \dfrac{\vdots}{\{n \geq 0 \wedge x = n \wedge y = 0\}\ \text{while}\ (x \neq 0)\ \{y := y + x; x := x - 1;\}\ \{\mathrm{y} = \Sigma_{i=0}^{n} i\}}}{\{n \geq 0 \wedge n = n \wedge 0 = 0\}\ x := \mathrm{n}; \mathrm{y} := 0; \text{while}\ (x \neq 0)\ \{y := y + x; x := x - 1;\}\ \{\mathrm{y} = \Sigma_{i=0}^{n} i\}}$$

$$\{n \geq 0\}\ x := \mathrm{n}; \mathrm{y} := 0; \text{while}\ (x \neq 0)\ \{y := y + x; x := x - 1;\}\ \{\mathrm{y} = \Sigma_{i=0}^{n} i\}$$

証明

$$\{\phi[e/x]\}\ x = e\ \{\phi\} \qquad \{\phi\}\ skip\ \{\phi\} \qquad \frac{\{\phi_1\}\ s_1\ \{\phi_2\} \quad \{\phi_2\}\ s_2\ \{\phi_3\}}{\{\phi_1\}\ s_1; s_2\ \{\phi_3\}}$$

$$\frac{\{\phi_1 \wedge e\}\ s_1\ \{\phi_2\} \quad \{\phi_1 \wedge e\}\ s_2\ \{\phi_2\}}{\{\phi_1\}\ \text{if}\ e\ \text{then}\ s_1\ \text{else}\ s_2\ \{\phi_3\}} \qquad \frac{\{\phi \wedge e\}\ s\ \{\phi\}}{\{\phi\}\ \text{while}\ e\ \text{do s}\ \{\phi \wedge \neg e\}} \qquad \frac{\phi_1 \Rightarrow \phi_1' \quad \{\phi_1'\}\ s\ \{\phi_2\} \quad \phi_2' \Rightarrow \phi_2}{\{\phi_1\}\ \mathrm{s}\ \{\phi_2\}}$$

推論規則

# 計算効果の形式的推論では

計算効果の「特徴」に着目して推論する

## 計算効果を確認するには?

`int f(void);` に計算効果はあるか?

以下の二つの関数の動作に差があるならば、f は計算効果を持つ

```
int copy1() {
  int y1 = f();
  int y2 = f();
  printf("%d, %d", y1, y2);
}
```

```
int copy2() {
  int y = f();
  printf("%d, %d", y, y);
}
```

注意: 差が無い場合は f に計算効果があるかはわからない

- `int f() {printf("hello"); return 0;}`
- `int f() {return rand();}`

特徴があいまいで
推論には使いづらい...

# 本講演での計算効果

計算効果 = **資源**に関する命令

- 計算効果と資源の例

| 計算効果 | 資源 |
|---|---|
| メモリ操作（malloc, free, ＊演算子) | ヒープメモリ |
| ファイル操作（fopen, fclose, …） | ファイル記述子 |
| 疑似乱数 (rand) | シード値 |
| ユーザー入出力 (printf, scanf, …) | ターミナル |
| 制御演算子 (shift/reset など) | 継続 |

# 本講演での計算効果

計算効果 = **資源**に関する命令

**特徴 ①**　命令が正しく動くかは資源の状態に依存

**特徴 ②**　エイリアスによる資源の共有が可能

# 特徴 ①：資源の状態に依存

- ポインタ命令：メモリの割当状態に依存

```
// 適切なメモリアドレスであれば安全に読み書きできる
😄 int* x = malloc(…); *x = 42;
😭 int* x = malloc(…); free(x); *x = 42;
```

- ファイルの読み書き：ファイル記述子の状態に依存

```
// ファイル記述子が有効であれば安全に読み書きできる
😄 FILE* fp = fopen(…); fprintf(fp, "hoge");
😭 FILE* fp = fopen(…); fclose(fp); fprintf(fp, "hoge");
```

# 特徴 ②：エイリアスによる資源の共有

```
// 違う変数が同じアドレスを共有することができる

int* x = malloc(…); int* y = malloc(…);
free(y); *x = 42;

int* x = malloc(…); int* y = x;
free(y); *x = 42;
```

# 推論技術に必要な要素

① 資源の状態管理

② 資源のエイリアスがあっても正しく推論できる

# 本講演で紹介する形式的推論技術

- Effect System
- 線形型 (linear type)
- 所有権 (ownership)

# 簡単な比較比較

| | 資源の取り扱い | エイリアス対応 |
|---|---|---|
| **Effect System** | 第二級に制限 | × |
| **線形型** | 制限なし | × |
| **所有権** | 制限なし | ○ |

# 簡単な比較

関数が資源に相当する値（メモリアドレスやファイル記述子など）を**返せない**

| | 資源の取り扱い | エイリアス対応 |
|---|---|---|
| **Effect System** | 第二級に制限 | × |
| **線形型** | 制限なし | × |
| **所有権** | 制限なし | ○ |

# 目次

- Effect System
- 線形型
- 所有権

# Effect System

- プログラムが起こす計算効果を静的に見積もる技術
- 例：Java のコンパイル時例外検査

```
class FileReader … {
  public FileReader(String fileName)
              throws FileNotFoundException { … } … }
```

  - メソッドが起こす例外を静的に見積もることで異常終了を防ぐ
  - 検査対象の例外を無視するとコンパイルエラーに

```
public void readFile() {
      FileReader r = FileReader(…); …
}
```

# その他の応用例

例：ポインタ操作

- メモリアクセス解析 [Lucassen & Gifford ’88]
- ライフタイム管理 [Tofte & Talpin ’94]

例：マルチスレッド / マルチプロセス

- スレッド / プロセス間通信解析 [Nielson & Nielson ’93]
- デッドロック検出 [Suenaga ’08]

# 例外検査の Effect System

次の規則に従ってステートメントの構造に従い発生の可能性のある例外の集合を見積る

- ステートメントが throw new Ex(...) であれば { Ex } を割当
- ステートメントが計算効果を起こさない命令であれば ø を割当
- ステートメントがステートメントの連結 $s_1; s_2$ で
  $s_1$ に { $Ex_{11}$, ..., $Ex_{1n}$ } が、$s_2$ に { $Ex_{21}$, ..., $Ex_{2m}$ } が割り当てられたとき
  全体には { $Ex_{11}$, ..., $Ex_{1n}$, $Ex_{21}$, ..., $Ex_{2m}$ } を割当
- 他も場合も同様

# 例外検査の Effect System

$$S \ \mathit{throws}\ \{\ \mathrm{Ex}_1,\ \ldots,\ \mathrm{Ex}_n\ \}$$

**「ステートメント $S$ を実行するとクラス Ex1, ..., Exn の例外が発生するかもしれない」**

$S\ \mathit{throws}\ \{\ \mathrm{Ex}_1,\ \ldots,\ \mathrm{Ex}_n\ \}$ を導くための規則（一部）

$$\frac{v_1, \cdots, v_n \text{ は値} \quad op \text{ は例外を起こさない命令}}{x = op(v_1, \cdots, v_n)\ \mathit{throws}\ \{\ \}}$$

$$\frac{S_1\ \mathit{throws}\ \{\ \mathrm{Ex}_{11},\ \ldots,\ \mathrm{Ex}_{1n}\ \} \quad S_2\ \mathit{throws}\ \{\ \mathrm{Ex}_{21},\ \ldots,\ \mathrm{Ex}_{2m}\ \}}{S_1; S_2\ \mathit{throws}\ \{\mathrm{Ex}_{11},\ \ldots,\ \mathrm{Ex}_{1n},\ \mathrm{Ex}_{21},\ \ldots,\ \mathrm{Ex}_{2m}\ \}}$$

$$\frac{}{\text{throw new Ex(...)}\ \ \mathit{throws}\ \{\ \mathrm{Ex}\ \}}$$

# 一般的な Effect System の基本構成要素

1. 計算効果を起こす命令 $F$

例外検査でいうと…

1. 例外を投げる throw 命令

# 一般的な Effect System の基本構成要素

1. 計算効果を起こす命令 $F$
2. 見積った計算効果の表現 $e$
   - 2a. 命令 $F$ の見積り $e_F$
   - 2b. 計算効果が起こらないことを示す $e_\epsilon$

例外検査でいうと…

1. 例外を投げる throw 命令
2. 例外クラスの有限集合
   { EX1, EX2, … }
   - 2a. $e_{\text{throw new Ex(…)}}$ = { Ex }
   - 2b. $e_\epsilon$ = { }

# 一般的な Effect System の基本構成要素

1. 計算効果を起こす命令 $F$
2. 見積った計算効果の表現 $e$
   - 2a. 命令 $F$ の見積り $e_F$
   - 2b. 計算効果が起こらないことを示す $e_\epsilon$
3. 見積った計算効果の合成 $e_1 \cdot e_2$

例外検査でいうと…

1. 例外を投げる throw 命令
2. 例外クラスの有限集合
   { EX1, EX2, … }
   - 2a. $e_{\text{throw new Ex(…)}}$ = { Ex }
   - 2b. $e_\epsilon$ = { }
3. $e_1 \cdot e_2 = e_1 \cup e_2$

# 一般的な Effect System の構成

$$s : e$$

**「ステートメント $s$ の実行によって起こる計算効果は $e$ で見積られる」**

$s : e$ を導くための規則（一部）

$$\frac{v_1, \cdots, v_n \text{ は値} \quad op \text{ は計算効果を起こさない命令}}{x = op(v_1, \cdots, v_n) : e_\epsilon} \qquad \frac{s_1 : e_1 \quad s_2 : e_2}{s_1; s_2 \ : e_1 \cdot e_2}$$

$$\frac{}{F : e_F}$$

# Effect System の一般的枠組みを対象とした研究

- Shin-ya Katsumata: Parametric effect monads and semantics of effect systems. POPL 2014

  Effect System の一般的な枠組みでの圏論（特にモナド）による表示的意味論に関する研究

- Colin S. Gordon: A Generic Approach to Flow-Sensitive Polymorphic Effects. ECOOP 2017

  命令の実行順序が重要となる計算効果のための一般的な Effect System の枠組みを与える研究

（どちらも計算効果の代数的構造に着目。勝股先生パートで説明？）

# Effect System によるグローバル変数のメモリ管理

グローバル変数 `int* x` に割り当てられたアドレスの状態管理

1. 計算効果を起こす命令
   変数 x を対象とした malloc, free, * 演算子
2. 計算効果の表現：{ (a, b) | a,b ∈ { Alloc, Free } } ∪ { None } のいずれか
   2a. 命令への計算効果の割当

   $e_{\text{malloc}}$ = (Free, Alloc), $e_{\text{free}}$ = (Alloc, Free), $e_{*x}$ = (Alloc, Alloc)

   2b. $e_{\epsilon}$ = None
3. 計算効果の合成

   (a,b)・(b,c) = (a,c)  (a, b, c ∈ { Alloc, Free })

   $e$・None = None・$e$ = $e$

# Effect System の限界？

- Effect System の「プログラム実行中の挙動」についての推論技術
- プログラム実行中に現われる「値」については何も保証しない
- 「使用中の資源を値として返す関数」があった場合、
  （単純な effect system では）それ以上資源の追跡は難しい

# 目次

- Effect System
- **線形型**
- 所有権

# 線形型 (linear type) とは

- 値が正確に一回だけ (**exactly once**) 使われることを保証する型
  - **「一回しか使えない」だけではなく「一回は使われる」ことも保証**
- 例：次はどちらも型検査エラー

```
lin<int> x = 42;  // x は一回使用される整数
int y = 3 * x;
printf("3 * %d = %d", x, y);
```

```
lin<int> x = …;  // x は一回使用される整数
int y = 3 * 42;
printf("3 * 42 = %d", 42, y);
```

# 線形型 (linear type) とは

- 値が正確に一回だけ (**exactly once**) 使われることを保証する型
  - **「一回しか使えない」だけではなく「一回は使われる」ことも保証**
- 例：次はどちらも型検査エラー

```
lin<int> x = 42;  // x は一回使用される整数
int y = 3 * x;
printf("3 * %d = %d", x, y);
```

```
lin<int> x = …;  // x は一回使用される整数
int y = 3 * 42;
printf("3 * 42 = %d", 42, y);
```

# 線形型 (linear type) とは

- 値が正確に一回だけ (**exactly once**) 使われることを保証する型
  - **「一回しか使えない」だけではなく「一回は使われる」ことも保証**
- 例：次はどちらも型検査エラー

```
lin<int> x = 42;  // x は一回使用される整数
int y = 3 * x;
printf("3 * %d = %d", x, y);
```

```
lin<int> x = …;  // x は一回使用される整数
int y = 3 * 42;
printf("3 * 42 = %d", 42, y);
// x が一度も使われていない
```

# 線形型による資源管理 [Wadler'90]

- 資源を型を線形型で表現

**通常のメモリ操作関数**

```
int* malloc();
void free(int* p);
void set(int*, int);
int get(int*);
```

**変更後**（未完成）

```
lin<int*> malloc();
void free(lin<int*> p);
void set(lin<int*>, int);
int get(lin<int*>);
```

# 線形型による資源管理 [Wadler'90]

- 資源を型を線形型で表現

**通常のメモリ操作関数**

```
int* malloc();
void free(int* p);
void set(int*, int);
int get(int*);
```

**変更後**（未完成）

```
lin<int*> malloc();
void free(lin<int*> p);
void set(lin<int*>, int);
int get(lin<int*>);
```

```
// 型検査エラー
lin<int*> x = malloc();
set(x, 42); printf("%d", get(x));
```

# 線形型による資源管理 [Wadler'90]

- 資源を解放しない関数は再度資源を返すよう変更

**通常のメモリ操作関数**

```
int* malloc();
void free(int* p);
void set(int*, int);
int get(int*);
```

**変更後（完成）**

```
lin<int*> malloc();
void free(lin<int*> p);
lin<int*> set(lin<int*>, int);
lin<pair<lin<int*>, int> get(lin<int*>);
```

# 線形型により管理されたメモリの使い方

```
lin<int*> malloc();
void free(lin<int*> p);
lin<int*> set(lin<int*>, int);
lin<pair<lin<int*>, int> get(lin<int*>);

int main() {
  lin<int*> x = malloc();
  lin<int*> y = set(x, 42);
  lin<int*> z, int a = get(y);
  printf("%d", a);
  free(z);
  …
}
```

# 線形型により管理されたメモリの使い方

```
lin<int*> malloc();
void free(lin<int*> p);
lin<int*> set(lin<int*>, int);
lin<pair<lin<int*>, int> get(lin<int*>);

int main() {
  lin<int*> x = malloc();
  lin<int*> y = set(x, 42);
  (lin<int*> z, int a) = get(y);
  printf("%d", a);
  free(z); // 解放し忘れると型検査エラー
  …
}
```

# 線形型により管理されたメモリの使い方

```
lin<int*> malloc();
void free(lin<int*> p);
lin<int*> set(lin<int*>, int);
lin<pair<lin<int*>, int> get(lin<int*>);

int main() {
  lin<int*> x = malloc();
  lin<int*> y = set(x, 42);
  (lin<int*> z, int a) = get(y);
  printf("%d", a);
  free(z);
  … // 解放後にメモリを使おうとすると型検査エラー
}
```

# 問題①

- インターフェイスの変更が必要
- 資源を「使用中」と「使用済み」の二状態しか表現できない
  - もっと多くの状態をもつ計算効果もある
    例：通信ソケットの状態は「初期状態」→「ポート割当」→「接続要求待ち」→「接続完了」→「読み書き可能」→「解放」と遷移

[Igarashi & Kobayashi ’05] では線形型の考えを使いつつこれら問題を解決

# 問題②

- エイリアスが扱えない
  - エイリアスは「複数の変数が同じ資源を指している」状態なので単純な線形型での解決は難しい
  - [Igarashi & Kobayashi ‘05] でもエイリアスは未対応

# 目次

- Effect System
- 線形型
- **所有権**

# 復習：エイリアスが問題になるケース・ならないケース

- 問題になるケース

```
// エイリアスを通じて解放したアドレスにアクセス
int* x = malloc(…); int* y = x;
free(y); *x = 42;
```

- 問題にならないケース

```
// エイリアスを通じて有効なアドレスにアクセス
int* x = malloc(…); int* y = x;
*y = 42; printf("%d", *x);
```

# 共有された資源を安全に使うには

エイリアスが問題になる命令を実行する時点で
資源が共有されていないことを保証すればよい

**資源の共有状況を推論できればよい**

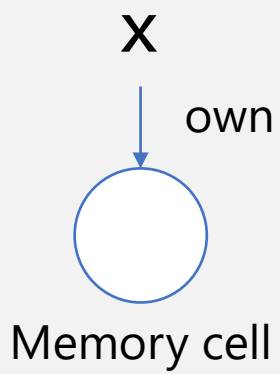
# 所有権による共有状態の推論

- 資源の所有権を型に付与することで資源の状態が以下のいずれかであるかを判定
  1. 資源は一つの変数によって占有されている
     - 占有されている資源は解放できる
  2. 資源は複数の変数によって共有されている
     - 共有されている間は資源の解放はできない

# 所有権による共有状態の推論

- 資源の所有権を型に付与することで資源の状態が以下のいずれかであるかを判定
  1. **資源は一つの変数によって占有されている**
     - 占有されている資源は解放できる
  2. 資源は複数の変数によって共有されている
     - 共有されている間は資源の解放はできない

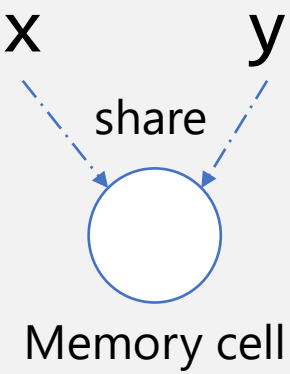
```
int* x = malloc(…);
free(x);
```



# 所有権による共有状態の推論

- 資源の所有権を型に付与することで資源の状態が以下のいずれかであるかを判定
  1. 資源は一つの変数によって占有されている
     - 占有されている資源は解放できる
  2. **資源は複数の変数によって共有されている**
     - 共有されている間は資源の解放はできない

```
int* x = malloc(…);
int* y = x;
```

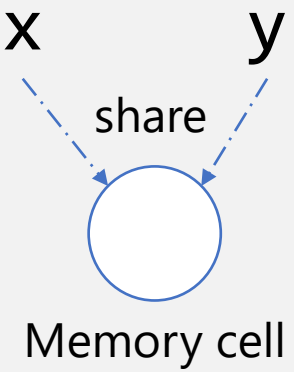


# 所有権による共有状態の推論

- 資源の所有権を型に付与することで資源の状態が以下のいずれかであるかを判定
  1. 資源は一つの変数によって占有されている
     - 占有されている資源は解放できる
  2. 資源は複数の変数によって共有されている
     - 共有されている間は資源の解放はできない

```
int* x = malloc(…);
int* y = x;
```

- 1 → 2 へは自由に遷移できる
- 2 → 1 への遷移を許し方によっていくつか方法がある
  - **Fractional ownerships**
  - **変数のライフタイム管理**



# Fractional Ownership [Suenaga & Kobayashi' 09]

- 資源の所有権を r ∈ [0,1] の実数で表現
  - r = 1：r が付く変数が資源を占有
  - r ≠ 1：他の変数と資源を共有
- エイリアスが作られるたびに所有権が分割される
- 「共有」から「占有」への遷移には注釈（assert）が必要

```
int* x = malloc(…);
// この時点では x の所有権は 1
int* y = x;
// この時点で x, y の所有権は 0.5
assert(x == y);
```

# 変数のライフタイム管理

- ライフタイム = 変数の有効期間
  - ライフタイムが終わると自動的に所有権は元の所有者に戻される
  - 典型的にはライフタイム = 変数のスコープ
- Rust で採用
- 実際にはもっと複雑なことをしている
  - RustBelt [Jung et al. '18] などを参照

```
int* x = malloc(…);
{
  int* y = x;
  … // x, yで資源を共有
}
// ここでは x が資源を占有
```

# まとめ

- 資源を利用する計算効果を例にいくつかの推論技術を紹介
- 資源の状態と共有をいかに扱うかが重要
- それぞれの技術は相補完的で（うっすらと）つながっている

# 今日話せなかった話題

- 分離論理 (資源推論のためのプログラム論理)
- shift/reset の型システム (with answer type modification)
- セッション型（並行プログラムの型システム）
- プログラムの停止性判定
- 確率的プログラムや量子プログラムの推論
- etc.

# 参考文献


- [Lucassen & Gifford ’88] John M. Lucassen, David K. Gifford: Polymorphic Effect Systems. POPL 1988: 47-57
- [Nielson & Nielson ’93] Flemming Nielson, Hanne Riis Nielson: From CML to Process Algebras (Extended Abstract). CONCUR 1993: 493-508
- [Tofte & Talpin ’94] Mads Tofte, Jean-Pierre Talpin: Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions. POPL 1994: 188-201
- [Suenaga ’08] Kohei Suenaga: Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. APLAS 2008: 155-170

# 参考文献


- [Katsumata ‘14] Shin-ya Katsumata: Parametric effect monads and semantics of effect systems. POPL 2014: 633-646
- [Gordon ’17] Colin S. Gordon: A Generic Approach to Flow-Sensitive Polymorphic Effects. ECOOP 2017: 13:1-13:31
- [Wadler ’90] Philip Wadler: Linear Types can Change the World! Programming Concepts and Methods 1990: 561-
- [Igarashi & Kobayashi ‘05] Atsushi Igarashi, Naoki Kobayashi: Resource usage analysis. ACM Trans. Program. Lang. Syst. 27(2): 264-313 (2005)

# 参考文献


- [Suenaga & Kobayashi ‘09] Kohei Suenaga, Naoki Kobayashi: Fractional Ownerships for Safe Memory Deallocation. APLAS 2009: 128-143
- [Jung et al. ‘18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer: RustBelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. 2(POPL): 66:1-66:34 (2018)