Linear Haskell での Rust 流借用の純粋な実現

松下 祐介[1]　田邉 裕大[2]　関山 太朗[3]　五十嵐 淳[1]

[1]京都大学　[2]東京科学大学　[3]国立情報学研究所

背景

命令型「操作」の世界 — 変化・高性能 — C/C++

Tofte+ '94 → +所有権型 → Rust (Matsakis+ '14)

状態を追うのが難しい、脆弱性を誘発

→ 理想：高性能 & 安全 ←

Linear Haskell (Bernardy+ '18) ← +線形型 ← Haskell (Wadler '90)

宣言型「概念」の世界 — 純粋・安全

計算のタイミングによらず常に同じ結果、参照透過

本研究 "Pure Borrow"

Rust の鍵「借用」の Linear Haskell 上での純粋な実現を探究

理論: 借用に対する表示的意味論・等式理論を構築、借用の純粋性を証明・確立

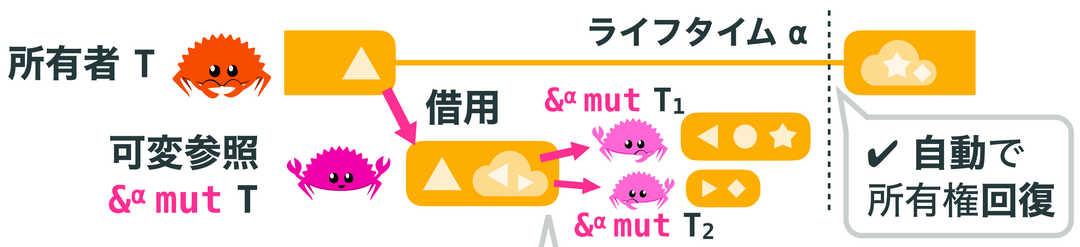
応用: 借用を Linear Haskell 上のライブラリとして実装・性能改善等を評価

所有権型

ポインタ・スレッド操作の安全性を所有権 (アクセス権) の管理で保証

借用 (Grossman+ '02)

所有権を時間で分割、柔軟



✔ 複数の参照で並列なアクセス　例 並列クイックソート　sort(v) → sort(v1) ‖ sort(v2)

純粋性の恩恵

- 並列性　例 `par :: a -> b -> (a, b)`
- 融合変換　例 `map g . map f = map (g . f)`
- CSE　例 `f x * f x = square (f x)`

線形型

線形矢印型 `a -o b`：引数を1度だけ消費

例 `write :: Vec a -o Int -> a -o (Vec a, a)`

従来 ✔ 性能: 破壊的更新　✔ 安全: 線形性により純粋

✘ 直接全データをムーブ、柔軟性・並列化に難

貢献1: Haskell 上の借用 API

線形型 × モナド で柔軟な操作を純粋な形で実現

多様な借用の操作をサポート

```
借用 borrow :: a -o BO^α (Mut^α a, End α -> a)
                              可変参照   所有権回復
更新 update  :: Mut^α a -o (a -o a) -o BO^α (Mut^α a)
消費 consume :: Mut^α a -o ()

再借用 reborrow :: Mut^α a -o BO^(α∧β)
                      (Mut^(α∧β) a, End β -> Mut^α a)
分割   getMany  :: Mut^α (Vec a) -o [Int] -> BO^α [Mut^α a]
共有   share    :: Mut^α a -o BO^α (Ur (Shr^α a))
```

BO^α モナド：ライフタイム α 中だけ実行可能

		cf. 従来の ST モナド
実行	`runB :: Now α -o BO^α a -o (Now α, a)`	✔
	`runB' :: Now α -o BO^(α∧β) a -o BO^β (Now α, a)`	–
逐次	`(>>=) :: BO^α a -o (a -o BO^α b) -o BO^α b`	✔
並列	`parB :: BO^α a -o BO^α b -o BO^α (a, b)`	✘

型でライフタイムを管理

```
newlft :: Lw -o ∃α. (Now α, Ur (Now α -o Ur (End α)))
```

非線形　順序を保証

線形性の証拠 Spiwack+ '22

α が継続中/終了済の証拠 RustBelt Jung+ '18 に由来

挑戦: 借用の純粋性の理論

計算の結果は決定的？(特に回復 `End α -> a`)　BO^α どうしは可換？(特に並列性 `parB`)

✘ 預言モデル (RustHorn 松下+ '20): 非決定的、所有権解放のタイミングの解析が必要

✘ 既存の純粋化 (Electrolysis Ullrich '16, Aeneas Ho+ '22): 扱える借用の操作が限定的

貢献2: 借用の表示的意味論 & 等式理論

BO^α の表示 = 借用の更新記録の Writer モナド

```
更新  update (Mut_x a) f ≜ <{x : f a}, Mut_x (f a)>
                            更新を記録
借用  borrow a ≜ νx. <{x : a}, (Mut_x a, λR. R[x])>
                                          記録から読む
      新しい ID の束縛
      νx.νy.a_xy = νy.νx.a_xy 等

newlft _ ≜ ({}, λR. R)
runB R <R+, a> ≜ (R; R+, a)
                  記録の合成、ID 重複は上書き
```

強力な等式 → 最適化・検証

```
(,) <$> a <*> b = parB a b

mapB f >=> mapB g = mapB (f >=> g)

update a f >>= λa, update a g
                 = update a (g . f)

mapB :: (a -o BO^α b)
     -> [a] -o BO^α [b]
```

現在　形式化・証明・実装等は未完成

将来的発展　Rust・Haskell を融合するような新しい言語？