

SML#コンパイラを速くする： タスク並列、末尾呼び出し、部分評価機構の開発

上野雄大

2025年6月15日 関数型まつり

自己紹介

上野雄大（うえのかつひろ）

- ▶ 2004年 学生として SML#コンパイラの開発に参加
- ▶ 2009年 職業として SML#コンパイラの開発に従事
- ▶ 2025年現在：新潟大学自然科学系准教授
（専門：計算機科学，ソフトウェア工学）

関数型言語開発歴 21 年（うち職業として 16 年）

- ▶ SML#開発歴 ≒ SML#使用歴
- ▶ SML# is 人生
- ▶ 大堀淳先生（SML#プロジェクト代表者）には感謝しかない

SNS（というかオンラインコミュニケーション全般）が苦手ですすみません…

SML#とは

Standard ML を包摂する日本発のフルスケール関数型言語.

ML を「ML に閉じない世界」で実現する

- ▶ C 言語とのシームレスな相互運用
- ▶ 既存のハードウェア・ソフトウェア資源をそのまま活用
- ▶ マシンネイティブなデータ表現
- ▶ OS ネイティブなマルチスレッドサポート
- ▶ C 言語と同様の分割コンパイルとリンク
- ▶ 強力なレコード演算（多相レコード，自然結合）
- ▶ SQL との統合
- ▶ 動的型による JSON の型付き操作

詳しくは <https://smlsharp.github.io/> をチェック

今日のおはなし：関数型言語を「作る」視点

- ▶ マルチコア CPU でベストパフォーマンスを出す (3.6.0～)
- ▶ Aarch64 対応 (4.2.0～)
- ▶ 部分評価に基づくコードの最適化 (4.2.0～)

これらの開発経験を題材に、関数型言語処理系を作る視点を共有したい。

- ▶ どんな問題に対して、何を狙い、何をしたのか

今日のおはなし：関数型言語を「作る」視点

- ▶ マルチコア CPU でベストパフォーマンスを出す (3.6.0～)
- ▶ Aarch64 対応 (4.2.0～)
- ▶ 部分評価に基づくコードの最適化 (4.2.0～)

これらの開発経験を題材に、関数型言語処理系を作る視点を共有したい。

- ▶ どんな問題に対して、何を狙い、何をしたのか

すべて SML# 4.2.0 (2025 年 3 月リリース) で試せます。

- ▶ ぜひ使ってみてね！

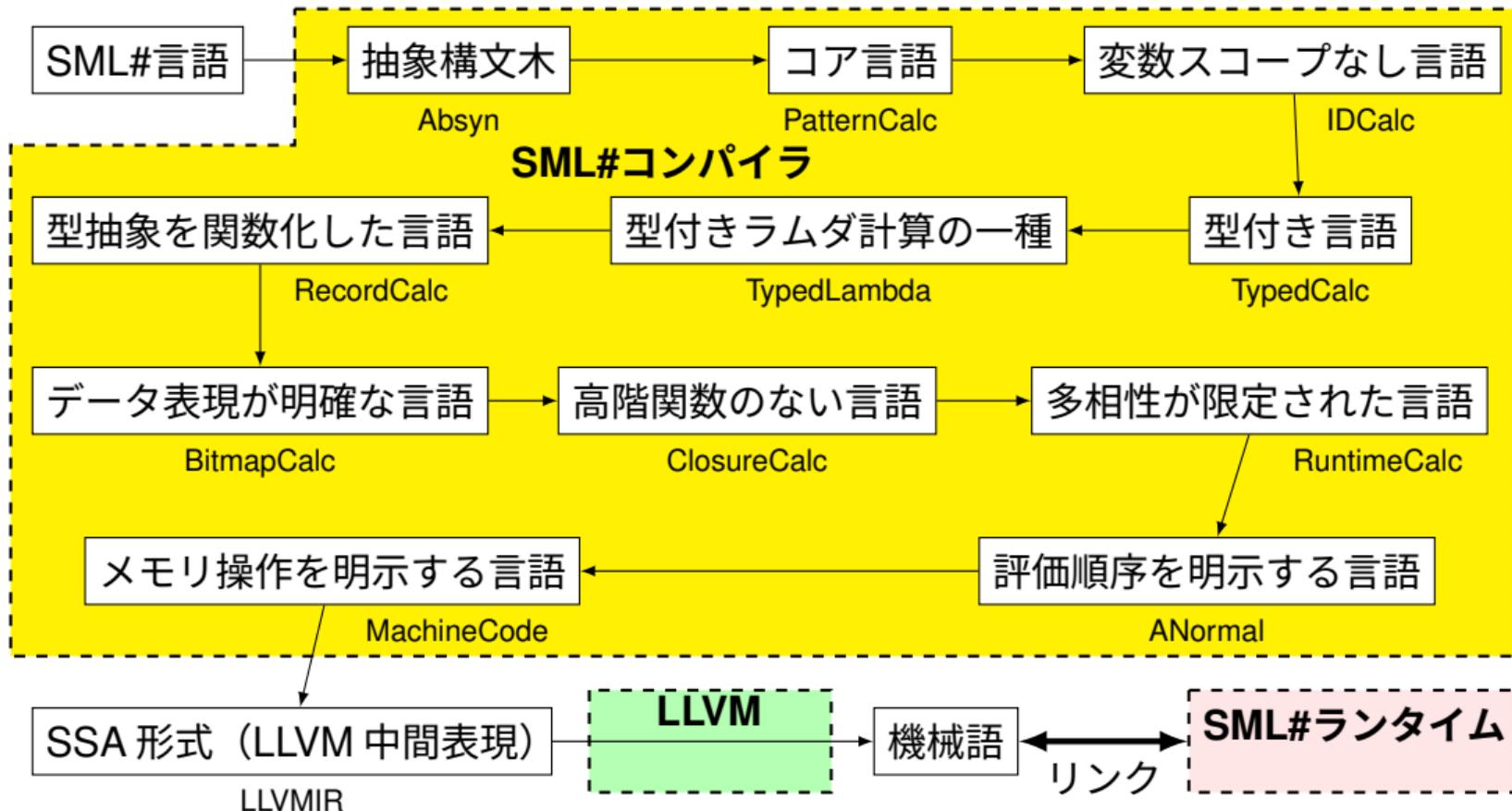
SML#コンパイラアーキテクチャ

SML#言語

???

機械語

SML#コンパイラアーキテクチャ



マルチコア CPU でベストパフォーマンスを出す

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

#

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

```
# val puts = _import "puts" : string -> int;
```

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

```
# val puts = _import "puts" : string -> int;  
val puts = fn : string -> int  
#
```

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

```
# val puts = _import "puts" : string -> int;  
val puts = fn : string -> int  
# puts "Hello";
```

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

```
# val puts = _import "puts" : string -> int;  
val puts = fn : string -> int  
# puts "Hello";  
Hello  
val it = 10 : int  
#
```

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

```
# val puts = _import "puts" : string -> int;  
val puts = fn : string -> int  
# puts "Hello";  
Hello  
val it = 10 : int  
# app (ignore o puts) ["Hello", "World"];
```

SML#はC言語との連携が得意

様々なC関数をSML#にそのままインポートできる。
インポートしたC関数にはMLの関数型が付く。
データ変換なしにSML#の値（文字列や配列など）をC関数に渡せる。

```
# val puts = _import "puts" : string -> int;
val puts = fn : string -> int
# puts "Hello";
Hello
val it = 10 : int
# app (ignore o puts) ["Hello", "World"];
Hello
World
val it = () : unit
#
```

pthread_create も直接呼べる

```
fun spawn f =  
  let  
    val 'a#boxed pthread_create =  
      _import "pthread_create"  
      : (unit ptr ref, unit ptr, 'a -> unit ptr, 'a) -> int  
    val r = ref (Pointer.NULL ())  
    fun callback f = (f (); Pointer.NULL ())  
  in  
    pthread_create (r, Pointer.NULL (), callback, f);  
    !r  
  end
```

(同様の関数が Pthread.Thread.create として標準提供されている)

細粒度スレッドライブラリ MassiveThreads を使ってタスク並列

MassiveThreads ならスレッドを山ほど作れる.

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n - 1) + fib (n - 2)
```

細粒度スレッドライブラリ MassiveThreads を使ってタスク並列

MassiveThreads ならスレッドを山ほど作れる。

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = if n < 16 then fib (n - 1) + fib (n - 2) else
            let val t = Myth.Thread.create (fn () => fib (n - 2))
            in fib (n - 1) + Myth.Thread.join t
            end
```

細粒度スレッドライブラリ MassiveThreads を使ってタスク並列

MassiveThreads ならスレッドを山ほど作れる。

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = if n < 16 then fib (n - 1) + fib (n - 2) else
            let val t = Myth.Thread.create (fn () => fib (n - 2))
            in fib (n - 1) + Myth.Thread.join t
            end
```

純粋な再帰関数とタスク並列は相性がいい

- ▶ fib (n - 1) の計算と fib (n - 2) の計算は互いに独立だよな
- ▶ それぞれの計算を別の物理コアで独立に並列実行したら倍速で終わるよな
- ▶ …本当に？

ボトルネック：ガベージコレクション（GC）

計算を進めるにはヒープからのメモリの供給が必要。

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = if n < 16 then fib (n - 1) + fib (n - 2) else
            let val t = Myth.Thread.create (fn () => fib (n - 2))
            in fib (n - 1) + Myth.Thread.join t
            end
```

この関数クロージャは

- ▶ グローバルなヒープのどこかにアロケートされ、
- ▶ `Myth.Thread.create` によってどこかの CPU と共有され、
- ▶ 不要になったら自動的に回収されなければならない

大域的な GC がスレッド間に暗黙の相互依存を生み出してしまう…。

世界を止めない並行並列 GC

SML#からマルチコア CPU の並列計算性能を最大限に活用するためには、大域的な資源管理をしながらもスレッドを止めない GC が必要.

- ▶ stop-the-world などもってのほか
- ▶ GC 都合のメモリ競合も最小限に

SML#の GC は世界を止めない並行並列 GC

- ▶ 全てのスレッドがミューテータにもコレクタにもなる
- ▶ 全体の統括者や同期をせずにグローバルな状態管理をする
- ▶ 世界を止めずにルートセット列挙
- ▶ スレッド数 1 のときはほぼ普通のマークスイープ GC

世界を止めない GC の難しさ

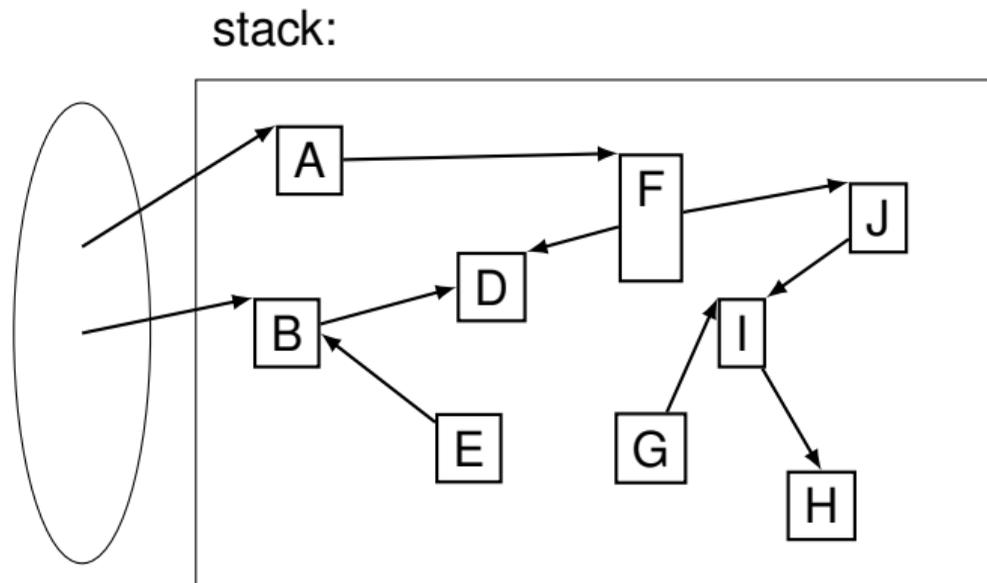
GC が成立するには「ある瞬間」における「ルートセット全体」と「そこから到達可能なオブジェクト集合」の両方が必要。

世界を止めずにこれらを手に入れるか？

1. トレーシングの最中にオブジェクトグラフが書き換わる
 - Yuasa のスナップショットライトバリアでオブジェクトグラフ全体のスナップショットを取る。
2. 全スレッドのルートセットを同時取得できない
 - Levanoni と Petrank のスヌーピングライトバリアでルートセット列挙のタイムラグを埋める。

Yuasaのスナップショット

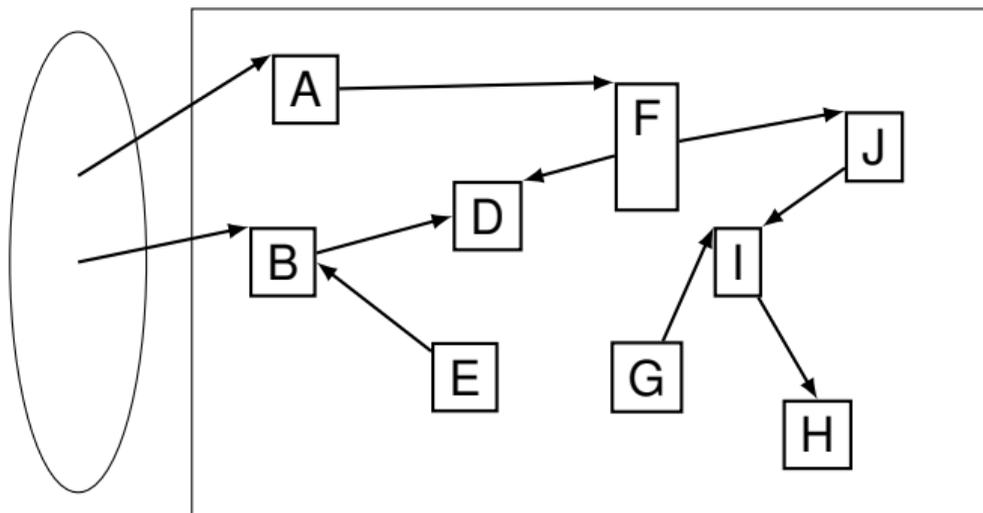
ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

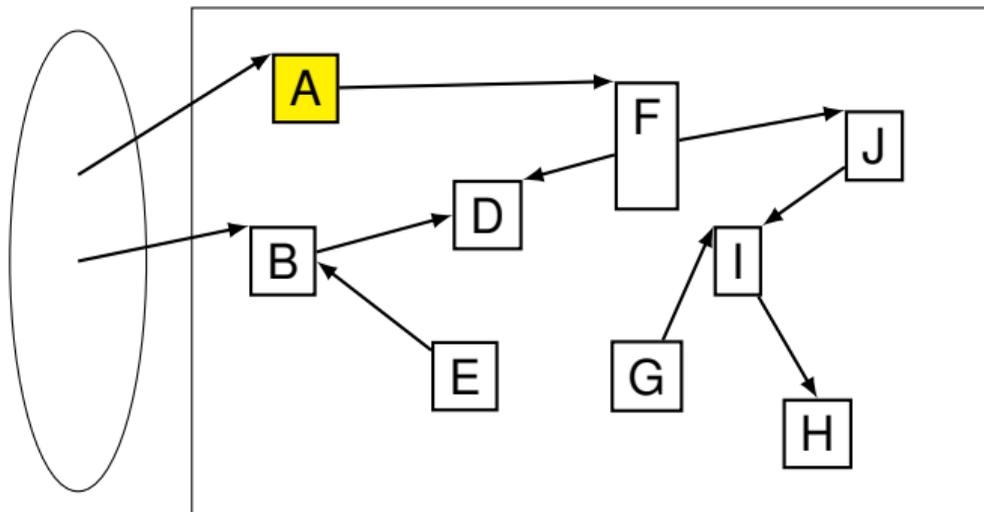
mark stack:



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

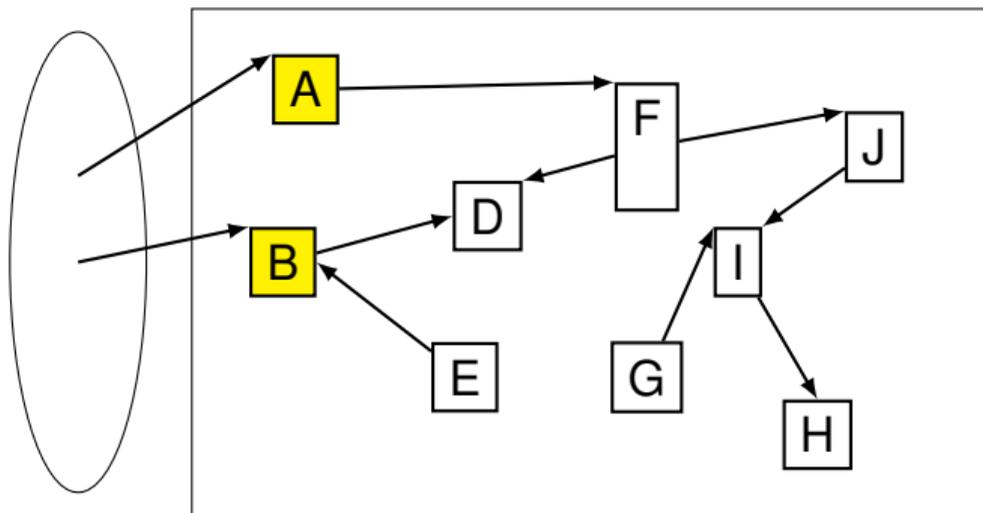
mark stack: A



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

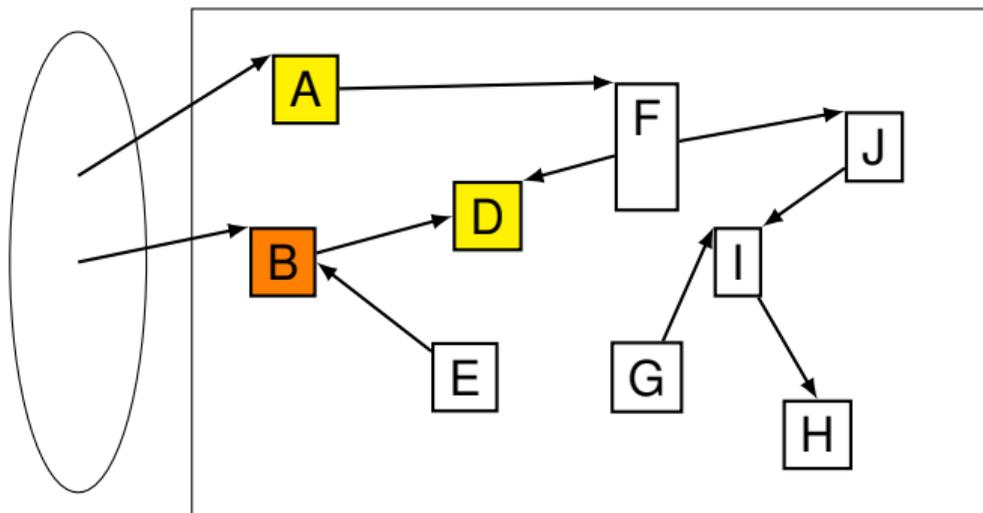
mark stack: AB



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

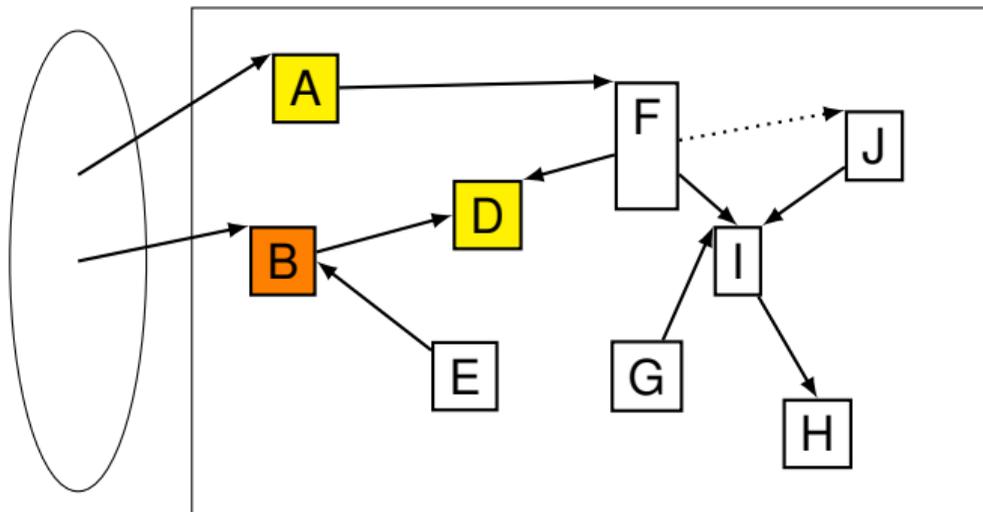
mark stack: AD



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

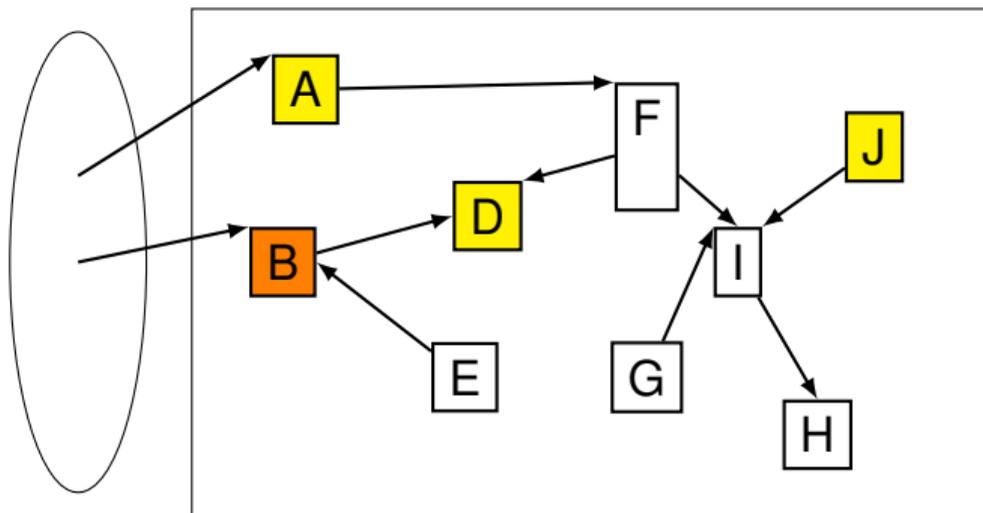
mark stack: AD



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

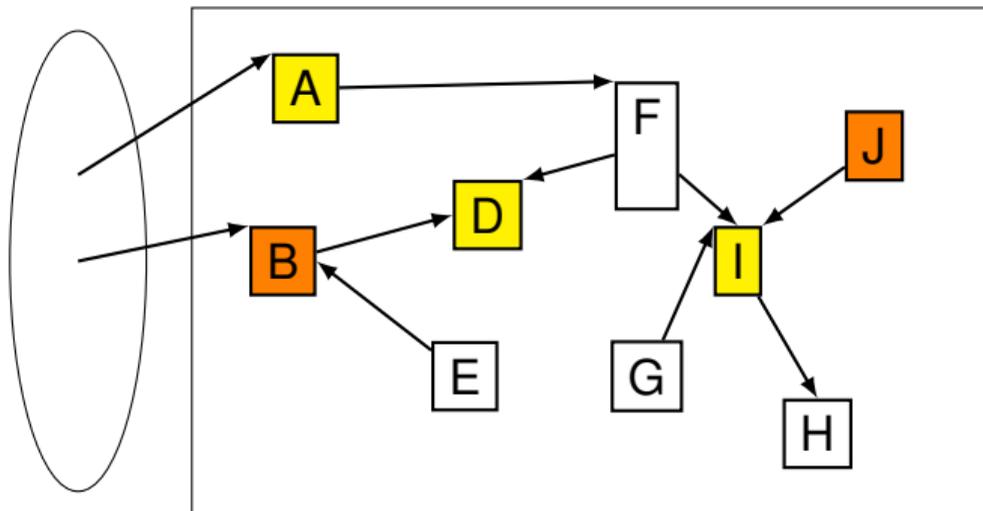
mark stack: ADJ



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

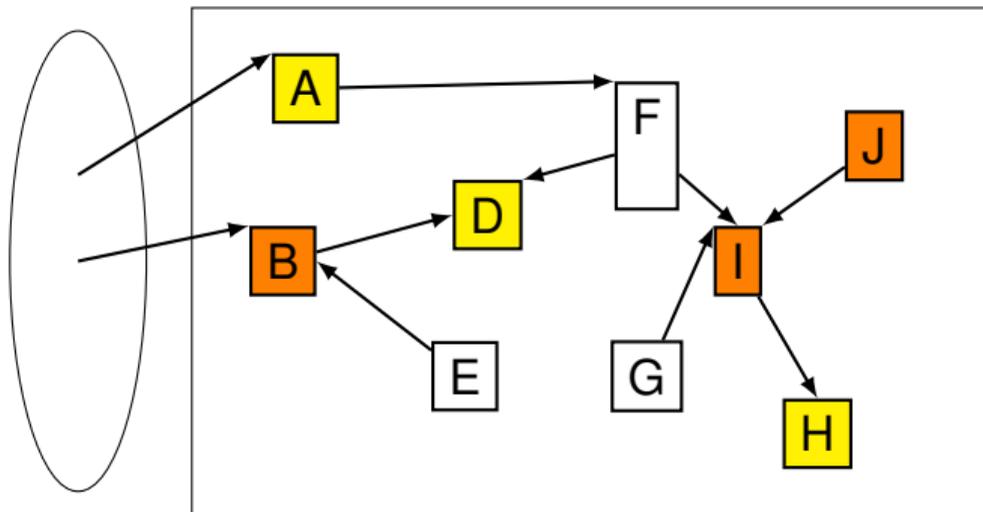
mark stack: ADI



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

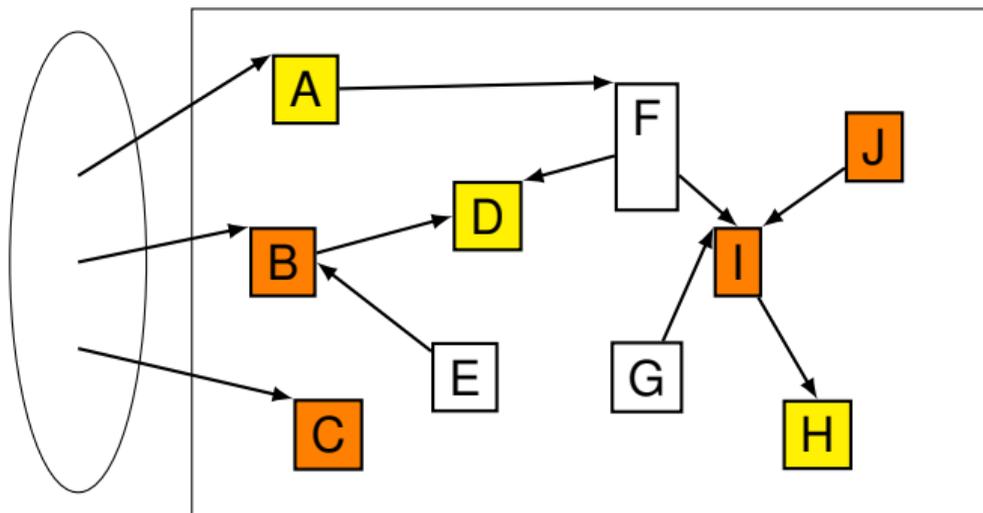
mark stack: ADH



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

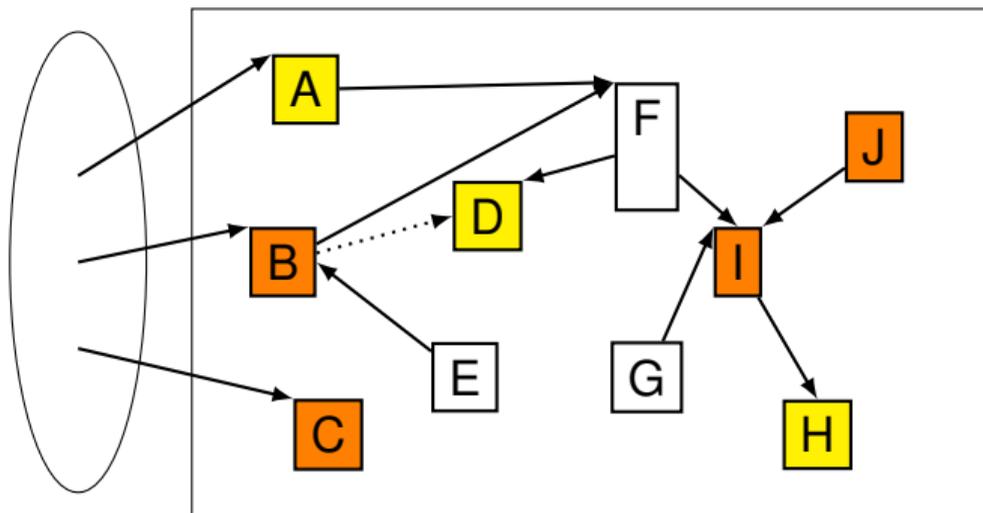
mark stack: ADH



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

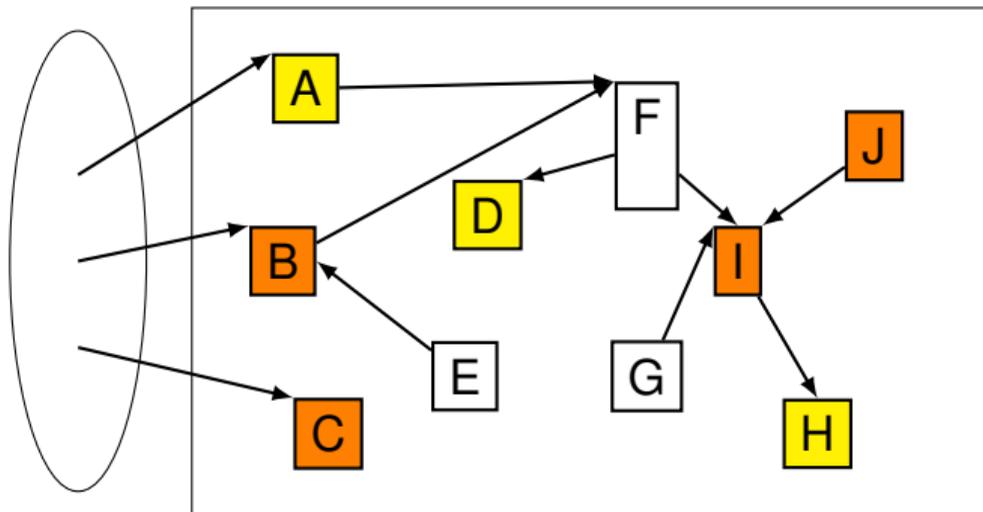
mark stack: ADH



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

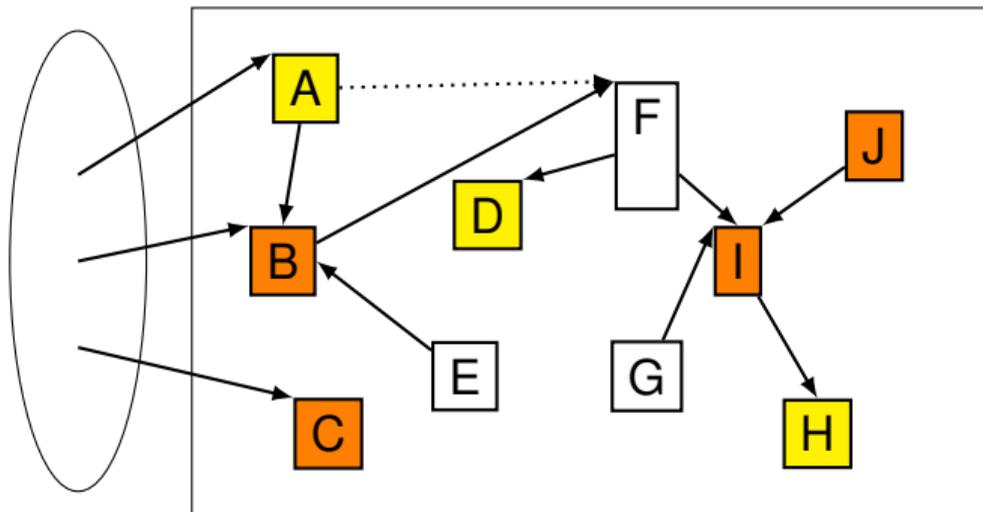
mark stack: ADH



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

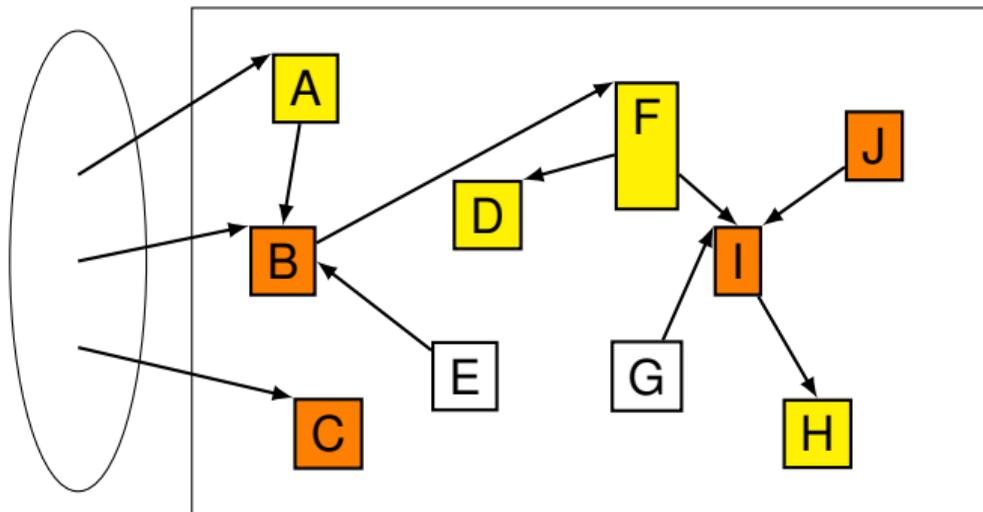
mark stack: ADH



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

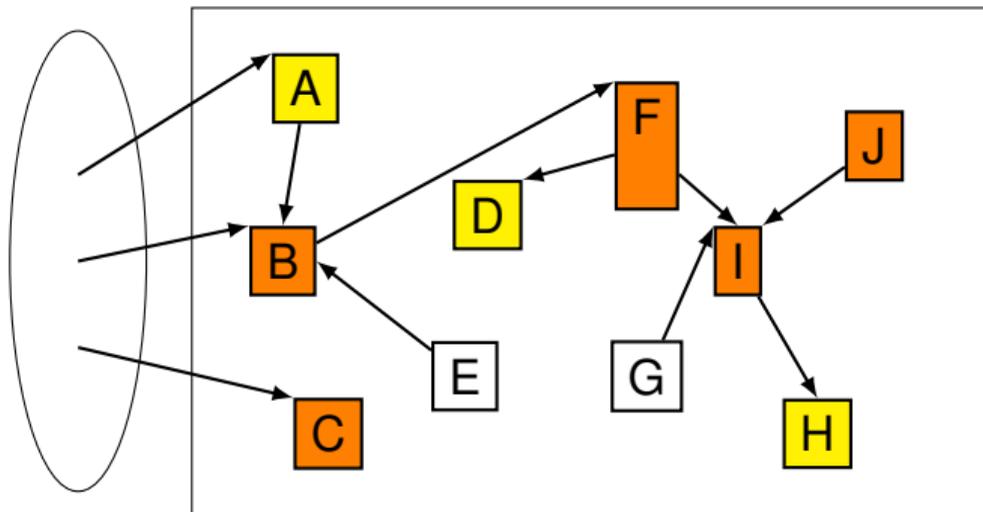
mark stack: ADHF



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

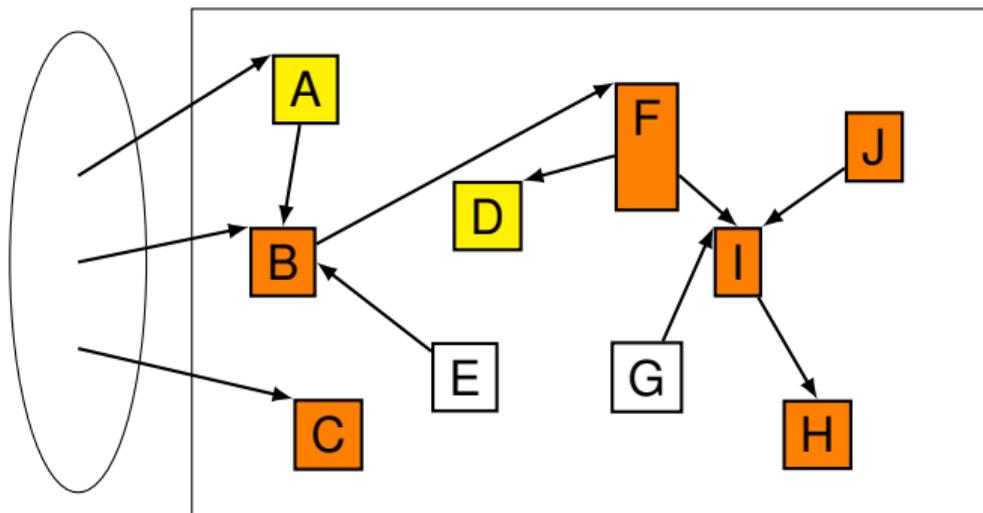
mark stack: ADH



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

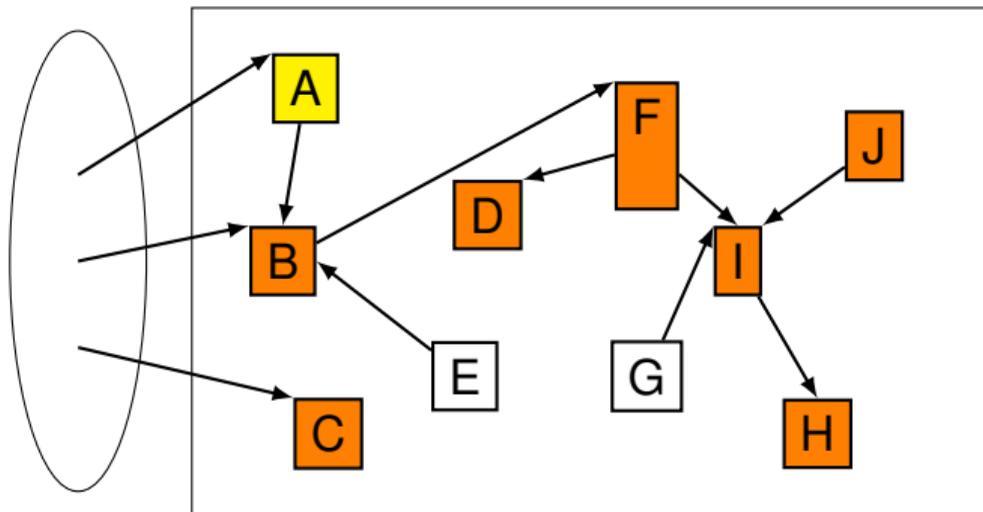
mark stack: AD



Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

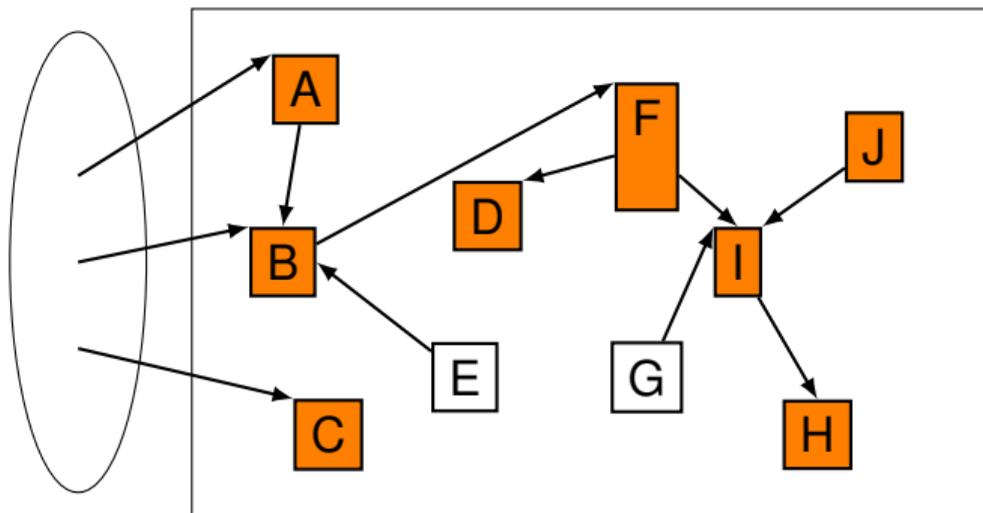
mark stack: A



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

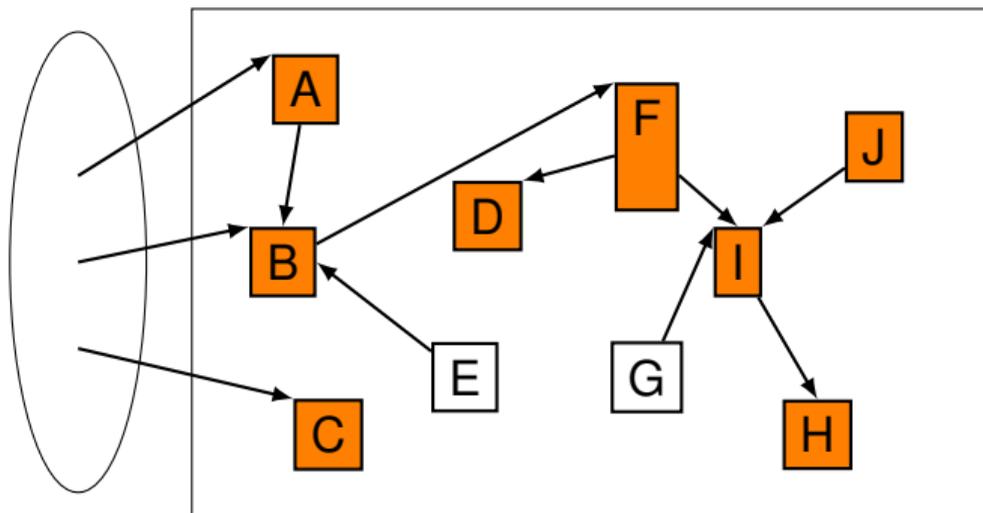
mark stack:



Yuasaのスナップショット

ミュートータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

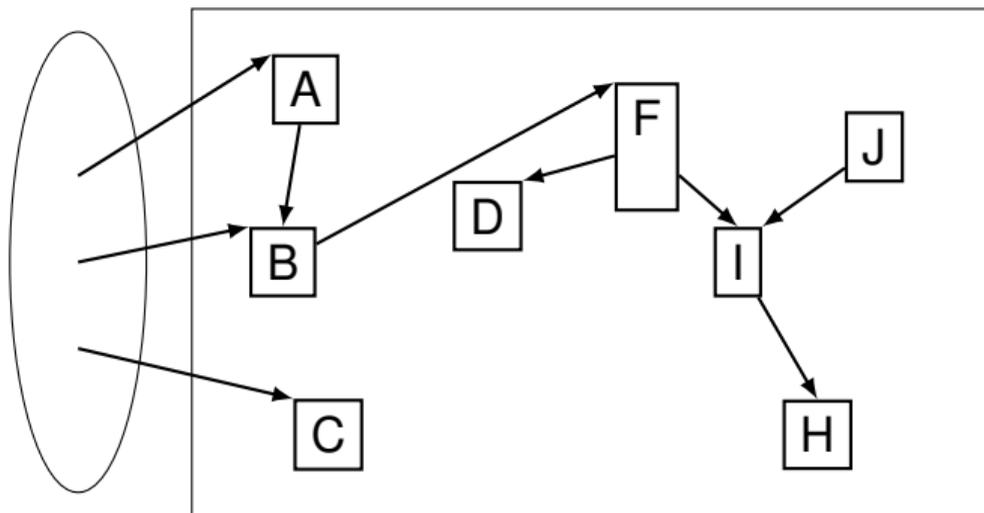
sweep stack:



Yuasaのスナップショット

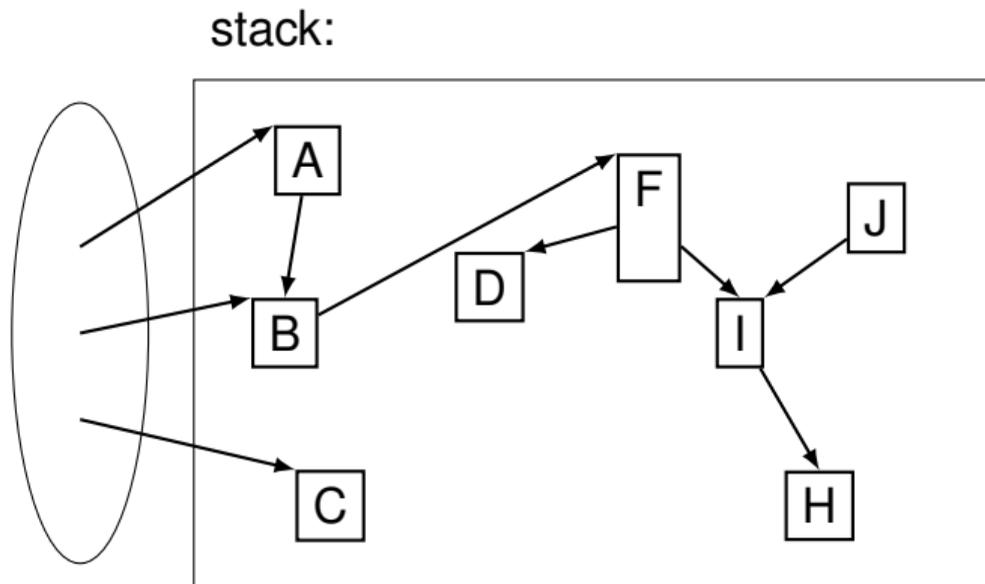
ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。

sweep stack:



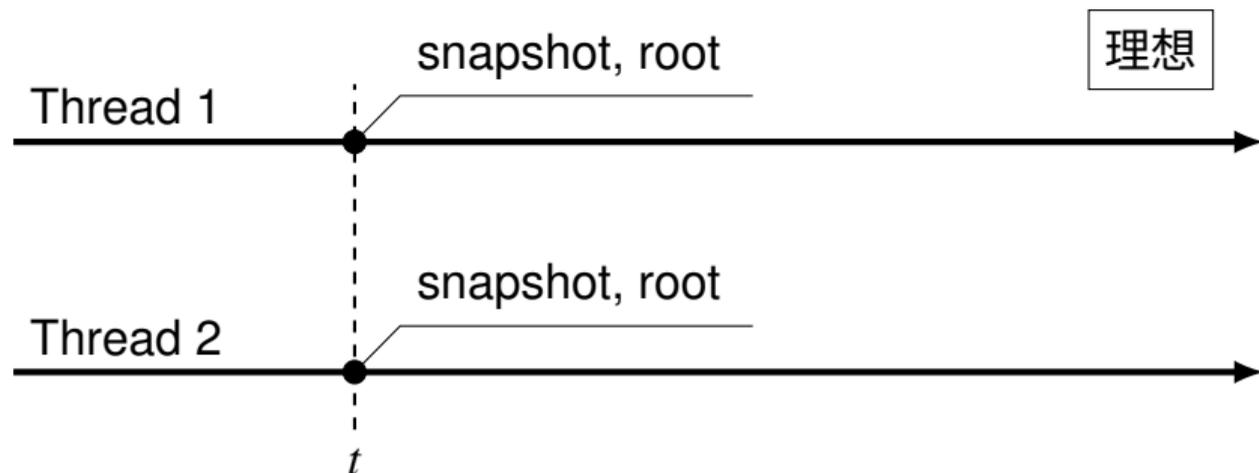
Yuasaのスナップショット

ミューテータがポインタを書き換えるとき、古いポインタの値を覚える（スナップショットライトバリア）ことで、ライトバリアを有効にした時点のオブジェクトグラフを保存する。



Levanoni & Petrankのスライディングビュー

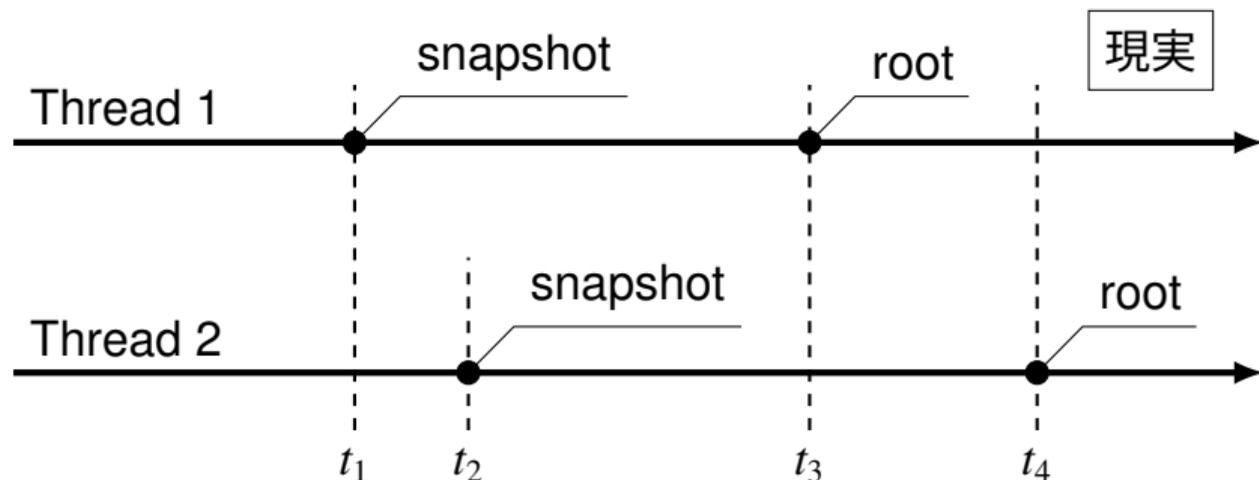
世界を止めないので，全スレッド同時にスナップショットライトバリアを有効にしたり，ルートセット列挙したりできない。



代わりに，全スレッドのスナップショットライトバリアを確実に有効にしてからルートセットを列挙する． t_3 と t_4 のギャップは，ポインタ書き換え時に新しいポインタの値を覚える（スヌーピングライトバリア）ことで埋める．

Levanoni & Petrankのスライディングビュー

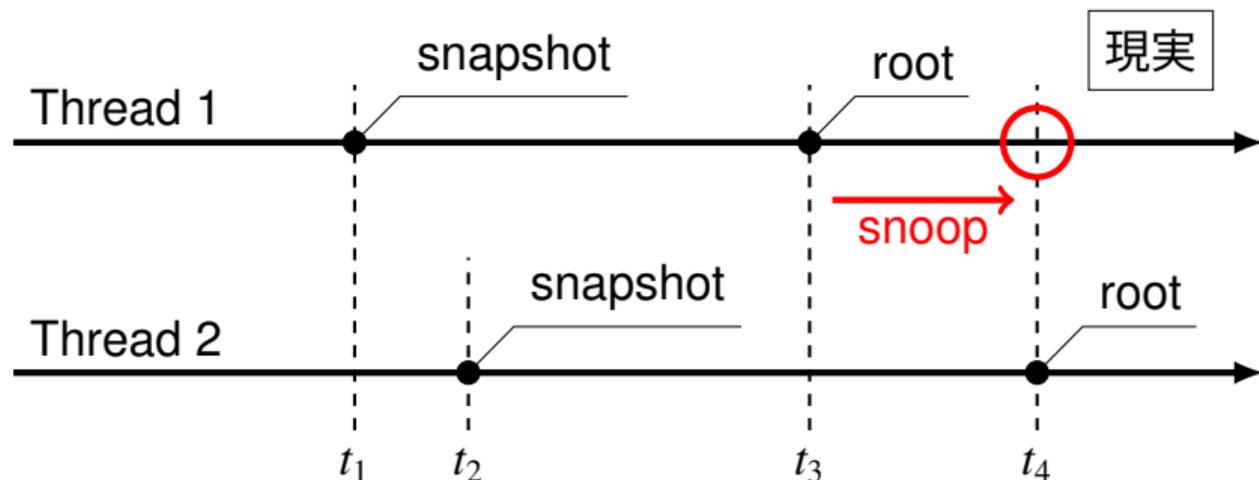
世界を止めないので，全スレッド同時にスナップショットライトバリアを有効にしたり，ルートセット列挙したりできない。



代わりに，全スレッドのスナップショットライトバリアを確実に有効にしてからルートセットを列挙する． t_3 と t_4 のギャップは，ポインタ書き換え時に新しいポインタの値を覚える（スヌーピングライトバリア）ことで埋める。

Levanoni & Petrankのスライディングビュー

世界を止めないので、全スレッド同時にスナップショットライトバリアを有効にしたり、ルートセット列挙したりできない。



代わりに、全スレッドのスナップショットライトバリアを確実に有効にしてからルートセットを列挙する。 t_3 と t_4 のギャップは、ポインタ書き換え時に新しいポインタの値を覚える（スヌーピングライトバリア）ことで埋める。

本当にこの戦略で動くのか…？

並行並列プログラミングは人類の手に追えない…！

- ▶ 動かすたびに違うことが起こる
- ▶ デバッガを使ったりデバッグプリントを入れると挙動が変わる
- ▶ それでいて確実に動いてくれないと困る
- ▶ 苦悶の跡：<https://github.com/smlsharp/smlsharp/blob/master/src/runtime/control.c> (2017 行のうち 1122 行が日本語のコメント)

こんなときは…

本当にこの戦略で動くのか…？

並行並列プログラミングは人類の手に追えない…！

- ▶ 動かすたびに違うことが起こる
- ▶ デバッガを使ったりデバッグプリントを入れると挙動が変わる
- ▶ それでいて確実に動いてくれないと困る
- ▶ 苦悶の跡：<https://github.com/smlsharp/smlsharp/blob/master/src/runtime/control.c> (2017 行のうち 1122 行が日本語のコメント)

こんなときは…

数理モデルを立てて証明

定理 [Ueno&Ohori 2022]

Let thread i ($1 \leq i \leq n$) enumerate its root set at t_i , t_0 be any time before t_i , and t be $\max(t_1, \dots, t_n)$. For any thread i and time $t' \geq t$, the following set \mathcal{G} is disjoint with $\mathcal{R}(t')^*(\mathcal{S}_i(t'))$.

$$\mathcal{G} = H \setminus T$$

where

$$H = \mathcal{H}(t_0) \cup \mathcal{A}_1[t_0, t_1] \cup \dots \cup \mathcal{A}_n[t_0, t_n],$$

$$S = \mathcal{S}_1(t_1) \cup \dots \cup \mathcal{S}_n(t_n),$$

$$B = \text{snd}(\mathcal{W}[t_0, t]) \cup \text{snd}(\mathcal{D}[t_0, t]),$$

$$T = \mathcal{R}(t)^*(S \cup B).$$

定理 [Ueno&Ohori 2022]

Let thread i ($1 \leq i \leq n$) enumerate its root set at t_i , t_0 be any time before t_i , and t be $\max(t_1, \dots, t_n)$. For any thread i and time $t' \geq t$, the following set \mathcal{G} is disjoint with $\mathcal{R}(t')^*(\mathcal{S}_i(t'))$.

$$\mathcal{G} = H \setminus T$$

where

$$H = \mathcal{H}(t_0) \cup \mathcal{A}_1[t_0, t_1] \cup \dots \cup \mathcal{A}_n[t_0, t_n], \quad \leftarrow \text{回収範囲}$$

$$S = \mathcal{S}_1(t_1) \cup \dots \cup \mathcal{S}_n(t_n), \quad \leftarrow \text{ルートセット}$$

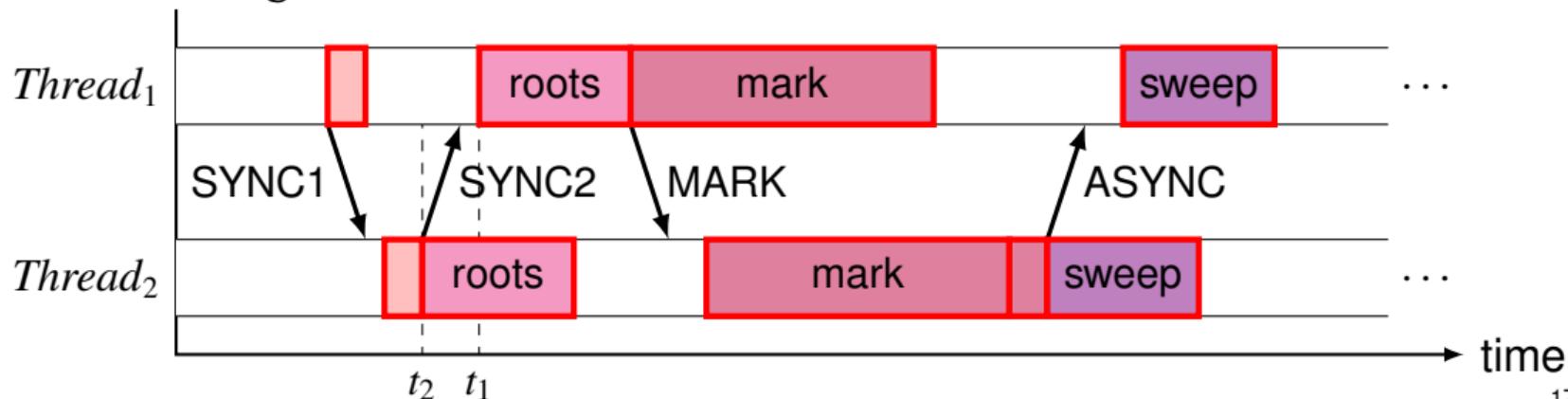
$$B = \text{snd}(\mathcal{W}[t_0, t]) \cup \text{snd}(\mathcal{D}[t_0, t]), \quad \leftarrow \text{ライトバリア}$$

$$T = \mathcal{R}(t)^*(S \cup B). \quad \leftarrow \text{トレース}$$

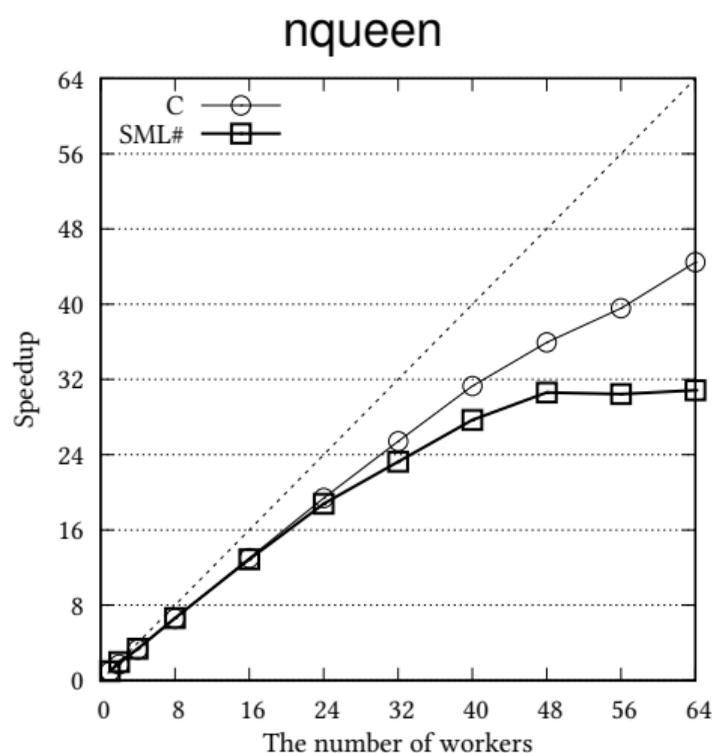
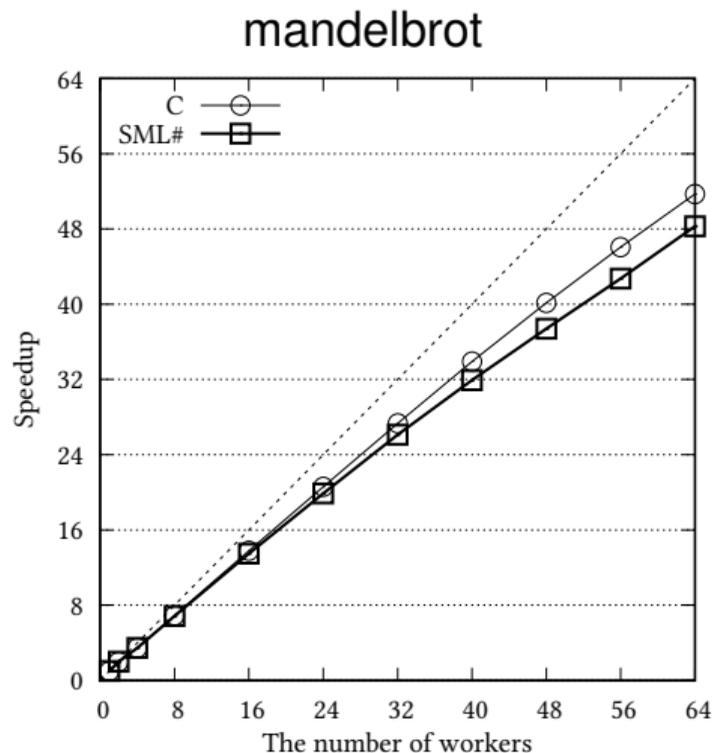
SML#の並行並列 GC アルゴリズム

4つのシグナルをスレッド間で送りあってハンドシェイク

1. SYNC1: 2つのライトバリアをオン. これで B が手に入る. 全スレッドがライトバリアをオンにした時点をも t_0 とする.
2. SYNC2: スレッドごとにルートセット列挙. これで S が手に入る. さらに現在のアロケーションポインタを覚えることで H を手にいれる.
3. MARK: スレッドごとにスヌーピングライトバリアをオフにしてトレース. T が手に入る.
4. ASYNC: G を回収する.



ベンチマーク (2022年 (SML# 3.6.0) 時点)



評価環境：AMD EPYC 7501 2.0 GHz (32 コア) × 2, 128 GB, Debian GNU/Linux 11

Aarch64 対応

ARM 対応の何が問題？

Q: LLVM のターゲットを切り替えるだけで ARM コード吐けるんじゃない？

A: 吐くだけならできます。動くかはなんとも…

動かない原因：

1. MassiveThreads が ARM 対応してない → 2021 年に解決
2. GC が ARM 対応してない？ → 実はほぼ何もしなくてよかった (!)
3. 例外機構が ARM 対応してない？ → amd64 と同じ Itanium ABI だった
4. LLVM の末尾呼び出し最適化が効かない！ → **今日の話**
5. LLVM の nested function が使えない！ → (将来の課題)

mod_poppo 氏による Apple Silicon 対応 SML# の試み (Thanks!)

▶ https://zenn.dev/mod_poppo/articles/smlsharp-aarch64-mac

末尾再帰とジャンプ

再帰呼び出しの後に計算が残っている＝スタックを消費する（コール）：

```
fun sumList nil = 0
  | sumList (h :: t) = sumList t + h
```

```
__SMLF7sumList:
    sub    sp, sp, #48
    cbz    x0, L4
    ldr    w19, [x0]
    ldr    x0, [x0, #8]
    bl    __SMLF7sumList
    add    w0, w0, w19
L4:      add    sp, sp, #48
    ret
```

末尾再帰とジャンプ

再帰呼び出しの後に計算が残っていない＝スタックを消費しない（ジャンプ）：

```
fun loop nil a = a
  | loop (h :: t) a = loop t (a + h)
fun sumList l = loop l 0
```

```
__SMLF7sumList:
        sub    sp, sp, #48
        mov    w19, wzr
L2:     cbz    x0, L4
        ldr    w9, [x0]
        ldr    x0, [x0, #8]
        add    w19, w9, w19
        b     L2
L4:     mov    w0, w19
        add    sp, sp, #48
        ret
```

末尾呼び出し最適化が効かない

- ▶ SML#の末尾呼び出し最適化は自己末尾再帰を除いてLLVMに頼っていた
- ▶ LLVMはARMターゲットにすると末尾呼び出しをジャンプにしてくれない
- ▶ その結果、相互末尾再帰でスタックオーバーフロー

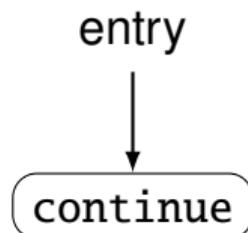
```
frame #53998: 0x000000010048f914 smlsharp`_SMLLL6action_3837 + 24032
frame #53999: 0x000000010048ff08 smlsharp`_SMLLL4scan_3836 + 1204
frame #54000: 0x0000000100490188 smlsharp`_SMLLL8continue_3835 + 296
frame #54001: 0x000000010048f914 smlsharp`_SMLLL6action_3837 + 24032
frame #54002: 0x000000010048ff08 smlsharp`_SMLLL4scan_3836 + 1204
frame #54003: 0x0000000100490188 smlsharp`_SMLLL8continue_3835 + 296
frame #54004: 0x000000010048dafc smlsharp`_SMLLL6action_3837 + 16328
frame #54005: 0x000000010048ff08 smlsharp`_SMLLL4scan_3836 + 1204
frame #54006: 0x0000000100490188 smlsharp`_SMLLL8continue_3835 + 296
frame #54007: 0x0000000100490554 smlsharp`_SMLLL3lex_3896 + 52
frame #54008: 0x000000010009f2d4 smlsharp`_SMLLL3get_31 + 192
frame #54009: 0x000000010009fe14 smlsharp`_SMLLL9parseStep_190 + 444
frame #54010: 0x00000001000a0540 smlsharp`_SMLLL7ssParse_195 + 48
frame #54011: 0x00000001000a6b78 smlsharp`_SMLLL5parse_274 + 2468
```

スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
            | ... => continue ()  
            | ... => result  
        in case ... of ... => action (...)  
          | ... => scan (...)  
          | ... => result  
        end  
      in scan (...)  
      end
```

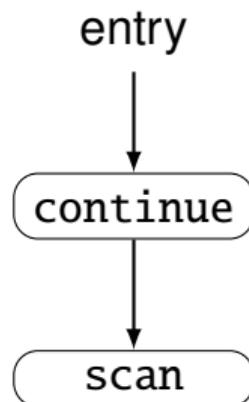
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



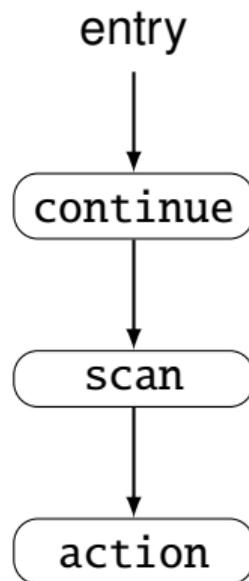
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



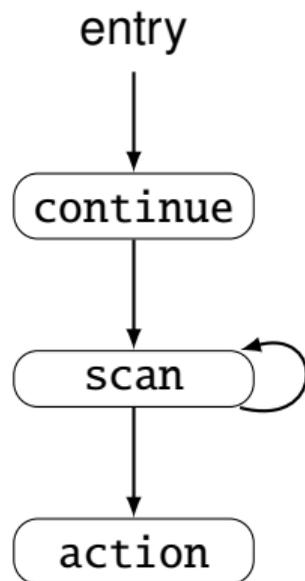
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



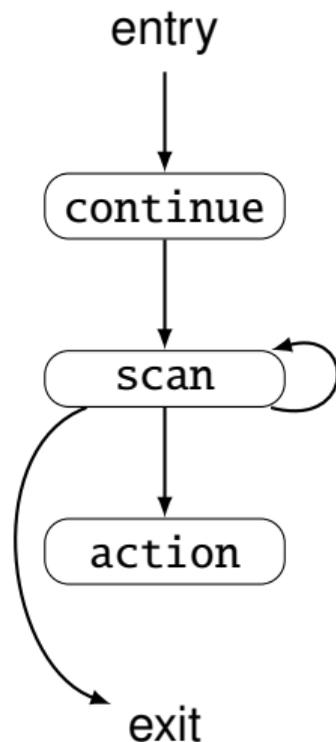
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



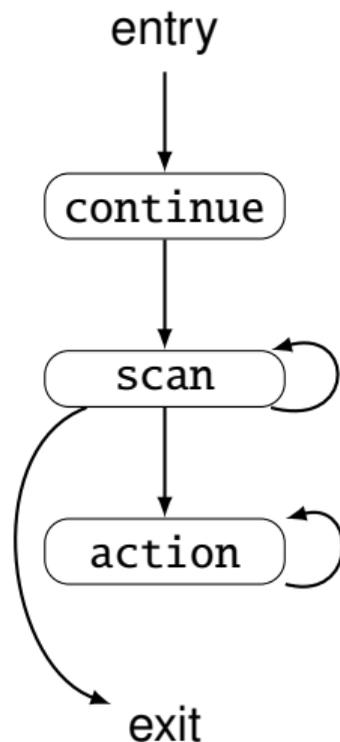
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



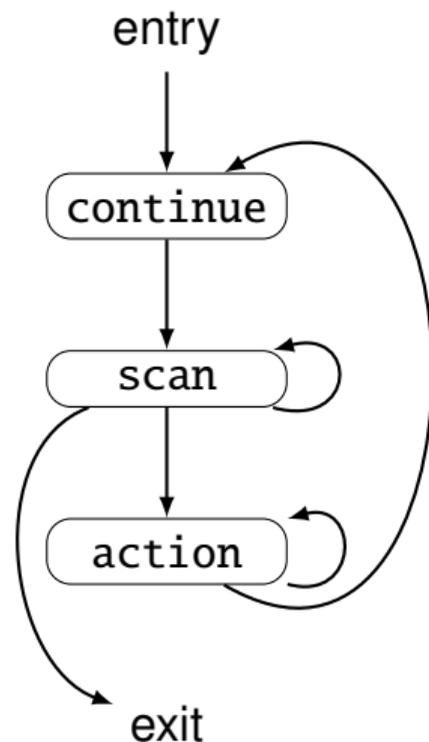
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



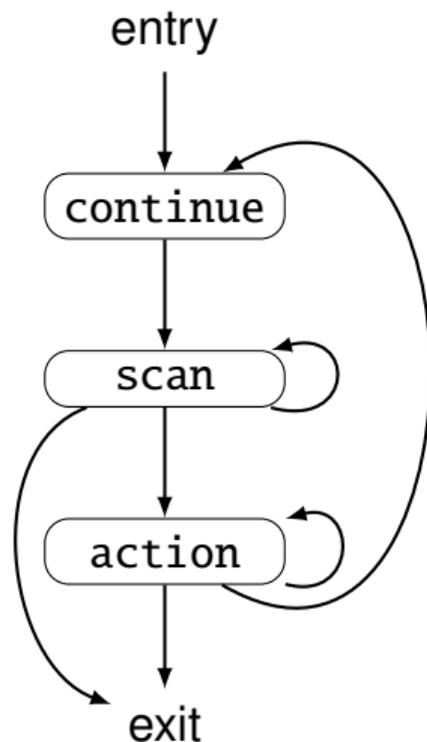
スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



スタック溢れを起こしたコード (iml.lex.sml)

```
fun continue () =  
  let  
    fun scan (...) =  
      let  
        fun action (...) =  
          case ... of ... => action (...)  
                | ... => continue ()  
                | ... => result  
      in case ... of ... => action (...)  
        | ... => scan (...)  
        | ... => result  
      end  
    in scan (...)  
  end
```



どうしよう…

CPS 変換して全ての関数適用を末尾呼び出しにすれば？

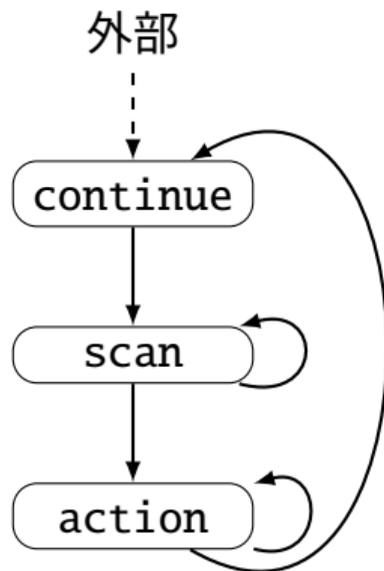
- ▶ C と同じ実行モデルを指向する SML# でそれはちょっと…

ジャンプを意図してそうな末尾呼び出しは全部本物のジャンプにせねばなるまい

- ▶ 再帰呼び出しは関数を越える
 - ▶ このネストした相互再帰関数群から二重ループを作れるか…？
- ▶ コンパイル単位内を跨ぐ場合は諦めてもいい
 - ▶ 逆にコンパイル単位内なら徹底的にやりたい
- ▶ 末尾位置の特定はアンカリー化とも絡む
 - ▶ アンカリー化も同時にやらざるを得ない

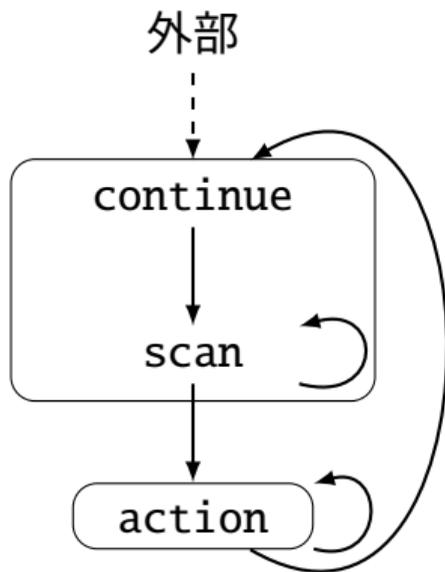
基本的な方針

関数 A が関数 B を末尾呼び出し（カーリー化されている場合を含む）しているとき、 B が（自己末尾再帰を除いて） A からしか呼ばれないならば、 B を A に統合し、末尾呼び出しをジャンプに置き換える。これをできなくなるまで繰り返す。



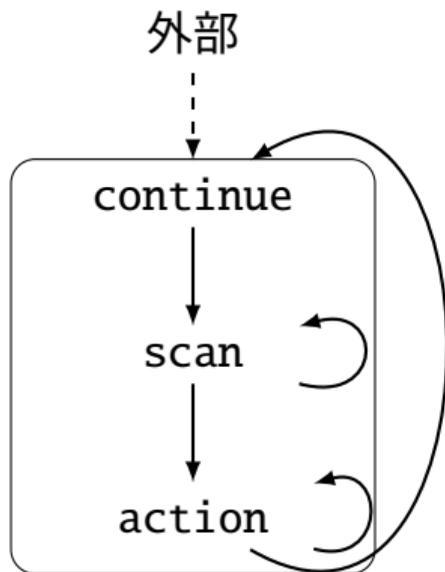
基本的な方針

関数 A が関数 B を末尾呼び出し（カーリー化されている場合を含む）しているとき、 B が（自己末尾再帰を除いて） A からしか呼ばれないならば、 B を A に統合し、末尾呼び出しをジャンプに置き換える。これをできなくなるまで繰り返す。



基本的な方針

関数 A が関数 B を末尾呼び出し（カーリー化されている場合を含む）しているとき、 B が（自己末尾再帰を除いて） A からしか呼ばれないならば、 B を A に統合し、末尾呼び出しをジャンプに置き換える。これをできなくなるまで繰り返す。



ジャンプを関数適用以外でどう表現するか？

SML#では中間言語に「関数を越えない例外機構」を導入.

- ▶ throw すると式のネストを飛び越えて catch にジャンプ
- ▶ catch 内での throw を許すことでバックエッジを表現

```
fun loop nil a = a
  | loop (h :: t) a = loop t (a + h)
fun sumList l = loop l 0
```

```
val sumList = fn l =>
  try {
    throw L (l, 0)
  } rec catch L (l', a') {
    case l' of nil => a' | h :: t => throw L (t, a' + h)
  }
```

末尾呼び出しコンパイル (TailCallCompile)

この問題に対してやれることを力技で全部やる！

1. ステップ0：全ての変数を名前替えしてユニークな識別子を振る
2. ステップ1：関数間呼び出し関係の抽出と整理
3. ステップ2：コールグラフの作成とクラスタリング
4. ステップ3：これまでに得た解析結果に基づいてコード変換

(調べるより先に手を動かしたい (←悪い癖))

ステップ 1-1: 呼び出し関係解析

解析結果を蓄える集合を R とする.

1. 関数定義 $\text{val } f = \text{fn } y_1 \Rightarrow \dots \Rightarrow \text{fn } y_m \Rightarrow e$ を見つけたら

$$[f \text{ is fun}(y_1 \dots y_m)]$$

を R に追加.

2. 関数 f の中に関数適用式 $g e_1 \dots e_n$ ($n \geq 0$) を見つけたら

$$[f \text{ calls } g(e_1 \dots e_n) \text{ at } p]$$

を R に追加.

- ▶ p (mid または tail) は末尾位置かそうでないかを表すフラグ
- ▶ トップレベルは Top という特別な関数名を持つとする
- ▶ 関数 f 内の無名関数は $f?$ という名前を持つとする

ステップ 1-2: デッドコード除去

以下の条件をすべて満たす最小の L を求める.

1. $Top \in L$
2. $[f \text{ calls } g(e_1 \cdots e_n) \text{ at } p] \in R$ かつ $f \in L$ ならば $g \in L$
3. $[f? \text{ calls } g(e_1 \cdots e_n) \text{ at } p] \in R$ かつ $f \in L$ ならば $g \in L$

$[f \text{ calls } g(e_1 \cdots e_n) \text{ at } p] \in R$ それぞれについて

- ▶ $f \notin L$ または $g \notin L$ ならば R から削除

ステップ 1-3: caller と callee で引数の最大数を揃える

例えば `[f is fun(y1y2y3)]` に対して

- ▶ `[g calls f(e1e2) at p1]`
- ▶ `[h calls f(e1) at p2]`

のように3未満の引数でしか呼ばれていないとき、`f`を2引数関数に読み替える。

`[f is fun(y1y2y3)], [f calls g(e)]`

```
val f = fn y1y2y3 => ... g e ...
```



```
val f = fn y1y2 => (fn y3 => ... g e ...)
```

`[f is fun(y1y2)], [f? calls g(e)]`

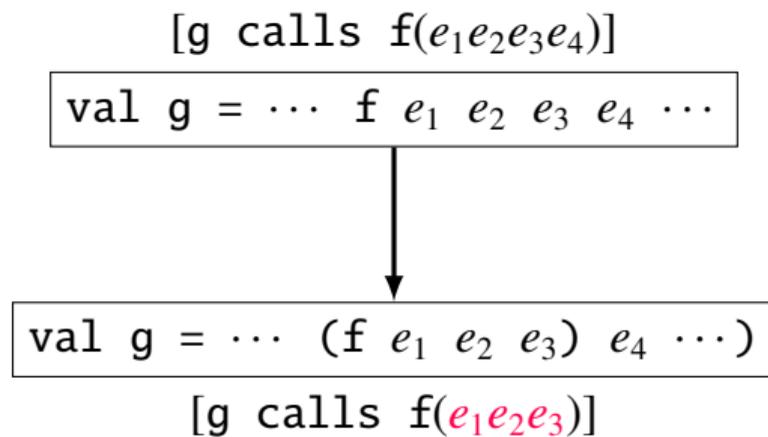
ステップ 1-3: caller と callee で引数の最大数を揃える

逆に $[f \text{ is fun}(y_1 y_2 y_3)]$ に対して

▶ $[g \text{ calls } f(e_1 e_2 e_3 e_4) \text{ at } p_1]$

▶ $[h \text{ calls } f(e_1 e_2) \text{ at } p_2]$

のように3より多い引数で呼ばれている場合があるとき，超過分の引数を削除する。



ステップ 1-4: eta-expansion

例えば `[f is fun(y1y2y3)]` に対して

▶ `[g calls f(e1e2) at p]`

のように 3 未満の引数で呼ばれている場合があるとき,

$$\text{val } f_2 = \text{fn } y_1y_2 \Rightarrow (\text{fn } y_3 \Rightarrow f \ y_1y_2y_3)$$

のような関数定義をあとで挿入することにして, 以下の情報を R に追加する.

▶ `[f2 is fun(y1y2)]`

▶ `[f2? calls f(y1y2y3) at tail]`

▶ `[f2 is eta(f, 2)]`

さらに `[g calls f(e1e2) at p]` を `[g calls f2(e1e2) at p]` に置き換える.

ステップ2-1: コールグラフの作成

関数名を頂点，末尾呼び出しを辺とする有向グラフを構築する．

1. $[f \text{ is fun}(\dots)] \in R$ ならば f を頂点としてグラフに加える
2. $[g \text{ calls } f(\dots) \text{ at tail}] \in R$ ならば g から f への辺をグラフに加える

さらに，以下の条件を満たす頂点 f を「入口頂点」と呼ぶことにする：

ある g が存在し $[g \text{ calls } f(\dots) \text{ at mid}] \in R$

ステップ 2-2: グラフ彩色

入口頂点 f それぞれについて, f から到達可能なすべての頂点 g に色 f を塗る.

2つ以上の色が塗られた頂点 g があれば, g に入る全ての辺を削除し, g を入口頂点に加えて, 彩色を最初からやり直す.

彩色が終わったら, $[f \text{ calls } g(\dots) \text{ at tail}] \in R$ それぞれについて,

- ▶ f と g の色が同じでなければ $[f \text{ calls } g(\dots) \text{ at mid}]$ に置き換える

ここまでで, 統合すべき関数の集合が定まった.

(同じ色を持つ関数は同じ関数の一部となる)

ステップ 2-3: 引数伝播

ある仮引数に必ず特定の実引数が渡されることが静的に読み切れるならば、その仮引数の参照をその特定の値に置き換えたい。

項集合に \top (静的には不明) と \perp (未使用) を加えて束を作り、仮引数に関する連立不等式を立て、その最小解を union-find で求める。

$[f \text{ is fun}(y_1 \cdots y_n)] \in R$ それぞれについて：

g の色が g ならば、連立不等式に $y_1 \geq y_1, \dots, y_n \geq y_n$ を加える

$[g \text{ calls } f(e_1 \cdots e_n) \text{ at tail}] \in R$ それぞれについて：

連立不等式に $y_1 \geq e_1, \dots, y_n \geq e_n$ を加える

ステップ3: 関数定義の変換

関数定義 $\text{val } f = \text{fn } y_1 \Rightarrow \dots \Rightarrow \text{fn } y_m \Rightarrow e$ それぞれについて:

誰からも呼ばれていない ($[g \text{ calls } f(\dots) \text{ at } p] \notin R$) ならば削除.
 $[f \text{ is fun}(y_1 \dots y_M)] \in R$ のはずなので, M 個の引数でアンカーリー化.
 $[f_i \text{ is eta}(f, i)] \in R$ それぞれについて関数定義を生成.
 f の色が f ならば:

$[g \text{ calls } f(\dots) \text{ at mid}] \in R$ がただひとつだけ存在するならば
インライン展開のために記憶して削除.

そうでなければ, 関数定義をこの場にそのまま残す.

$[g \text{ calls } f(\dots) \text{ at } p] \in R$ がただひとつだけ存在するならば
インライン展開のために記憶して削除.

この関数定義を含む関数の色と f の色が同じなら
この場で catch に変換.

以上のいずれにも当てはまらなければ
catch 化のために記憶して削除.

ステップ3: 関数適用式の変換

関数適用式 $f e_1 \cdots e_n$ それぞれについて:

$[f \text{ is fun}(y_1 \cdots y_m)] \in R$ のはず.

$n < m$ ならば, $[f_n \text{ is eta}(f, n)] \in R$ のはずなので, f を f_n に置き換える.

$\min(n, m)$ でアンカー化.

f がインライン関数として記憶されているならば

この位置で f をインライン展開

末尾位置でありかつこの関数適用を含む関数の色と f の色が同じならば

throw に変換

そうでなければ, 関数適用をここに残す.

末尾呼び出しコンパイルフェーズの実装

- ▶ 全部で 1456 行：<https://github.com/smlsharp/smlsharp/tree/master/src/compiler/compilePhases/tailcallcompile/main>
- ▶ 関数型の皮をかぶった手続き的コード
 - ▶ 関数型言語でも手続き的に書けるし書くべきときもある
- ▶ 多相関数（型抽象）にも対応（「色＝関数名＋型インスタンス」と取る）

部分評価に基づくコードの最適化

純粋な速さの追求：関数を関数でなくす

関数型言語では関数が計算手続きを書き表す中核となるため、多量の関数抽象と関数適用がプログラム中に現れる。しかし：

- ▶ 関数抽象（クロージャ）は一般にはメモリ確保を伴う
- ▶ 関数適用（コール）は一般にはスタックフレーム確保を伴う

操作であり、安直に実装すると、遅い。

実行効率のためには、如何にこれらをこのとおりに実装「しない」かが重要。

- ▶ 先ほどの末尾呼び出しコンパイルもこの手の方向性を共有

実行前に消せる関数適用は全部消しておきたい

```
fun score x y = x * 2 + y * 3
```

```
fun left x = score x 4
```

```
fun right y = score 4 y
```

みたいなコードがあったとき、

```
score x 4 → x * 2 + 4 * 3 → x * 2 + 12
```

```
score 4 y → 4 * 2 + y * 3 → 8 + y * 3
```

なわけだから結局

```
fun left x = x * 2 + 12
```

```
fun right x = 8 + y * 3
```

だよな、と実行前に見切っておきたい。

部分評価

あるプログラム（関数）の入力（引数）の一部が静的に定まっているとき、その入力をあらかじめ与えて特化したプログラムを生成する技法。

例：

```
fun score x y = x * 2 + y * 3
```

に対して $x = 4$ であることがあらかじめわかっているとき、 $x = 4$ が関わる計算を全て事前に行って特化した

```
fun score4 y = 8 + y * 3
```

を実行前に作り出す。

部分評価による最適化の可能性

ソースプログラム中に含まれる静的な情報に対して、コンパイル単位内の全ての関数を実行前に特化しておきたい。

インライン展開や定数伝播をする最適化器はすでにある (TCOptimization)

- ▶ インライン展開はするが評価まではしてくれない
- ▶ 評価順序が変わってしまうから β 簡約はしない (let を挿入)
- ▶ 再帰関数は展開しない
- ▶ 性能に満足できない

コンパイラにインタプリタを内蔵すればよいのではないか

- ▶ ML インタプリタはふつう自然意味論に基づいて書くよね
- ▶ 部分評価器を自然意味論のように書けたら嬉しい
- ▶ (関連研究調べる前にとりあえず手を動かしたい…)

自然意味論 (natural semantics)

プログラム M の意味を，環境 E の下で M は値 v に評価される，という判定を導出する規則の積み重ねによって与える意味論．ほぼインタプリタそのもの．

$$\frac{}{E\{x \mapsto v\} \vdash x \Downarrow v}$$

$$\frac{}{E \vdash \lambda x.M \Downarrow \text{Cls}(E, x, M)}$$

$$\frac{E \vdash M_1 \Downarrow \text{Cls}(E', x, M) \quad E \vdash M_2 \Downarrow v_2 \quad E'\{x \mapsto v_2\} \vdash M \Downarrow v}{E \vdash M_1 M_2 \Downarrow v}$$

「横線の上にある判定が全て得られたならば下の判定が得られる」と読む．
式全体を値に移してしまうため残余プログラムは得られない…．

部分評価のための自然意味論

残余プログラムを保存するための場所（ヒープ）を用意しよう。

- ▶ ヒープには「将来使うデータ」や「今は行わない計算」を置くことができる
- ▶ 「未知の値」も，ヒープに置いたことにして，（実体を参照できない）ポインタとして表現できるのでは
- ▶ 実行順序を保存するためポインタにはヒープ内で順序を与える
- ▶ 「今は行わない計算」はポインタやヒープ内順序を介してヒープ内でグラフ構造をなすはず．これは残余プログラムそのもの

部分評価のための自然意味論：評価判定のスケッチ

$$H, E \vdash M \Rightarrow H', v$$

- ▶ H (ヒープ)：「今は行わない計算」を保持するグローバルな記憶装置.
- ▶ E (変数環境)：変数への既知の値の対応付け.
- ▶ M (プログラム)：部分評価対象のプログラム.
- ▶ H' (ヒープ差分)：グローバルなヒープに新たに加えられるデータ列.
- ▶ v (値)：評価結果の値. 定数 c かポインタ p のいずれか.

ヒープ H と環境 E の下でプログラム M を評価すると、ヒープには新たに H' が割り付けられ、その結果として値 v が得られる。

部分評価器のための自然意味論：評価規則のスケッチ (1)

定数式，変数式は，その値を返すだけ。

$$\frac{}{H, E \vdash c \Rightarrow [], c}$$

$$\frac{}{H, E \vdash p \Rightarrow [], p}$$

$$\frac{}{H, E\{x \mapsto v\} \vdash x \Rightarrow [], v}$$

部分評価器のための自然意味論：評価規則のスケッチ (2)

$$\frac{H, E\{x \mapsto p\} \vdash M \Rightarrow H', v}{H, E \vdash \lambda x. M \Rightarrow [p' \mapsto \lambda x. [p \mapsto x](\langle H'; v \rangle)], p'} \quad (p, p' \text{ fresh})$$

- ▶ 仮引数に「未知の値」（何も指さないポインタで表現）を入れて本体を評価する
- ▶ 本体を評価したらヒープ差分 H' が得られる．この H' には M の残余プログラムが保存されている．
- ▶ $\langle H'; v \rangle$ は「 H' にしまわれている計算を v に至る範囲で再生する」ような式を作る演算（いわゆる let-insertion）
- ▶ 生成された関数抽象はヒープに割り付けて、そのポインタを評価結果の値とする．

部分評価器のための自然意味論：評価規則のスケッチ (3)

$$\frac{H, E \vdash M_1 \Rightarrow H_1, v_1 \quad HH_1, E \vdash M_2 \Rightarrow H_2, v_2 \quad HH_1H_2(v_1) = \lambda x.M \quad HH_1H_2, \{x \mapsto v_2\} \vdash M \Rightarrow H_3, v}{H, E \vdash M_1 M_2 \Rightarrow H_1H_2H_3, v}$$

- ▶ M_1 と M_2 をそれぞれこの順に評価する。
- ▶ グローバルなヒープにはヒープ差分が蓄積されていく。
- ▶ ポインタ v_1 を参照してヒープからオブジェクトを取り出す。
- ▶ それが関数抽象ならば、実引数を渡して関数本体を評価する。
- ▶ ヒープにしまわれていた関数抽象は自由変数を持たない（すでに評価されて定数になっている）。

部分評価器のための自然意味論：評価規則のスケッチ (4)

$$\frac{H, E \vdash M_1 \Rightarrow H_1, v_1 \quad HH_1, E \vdash M_2 \Rightarrow H_2, v_2 \quad HH_1 H_2(v_1) \neq \lambda x.M}{H, E \vdash M_1 M_2 \Rightarrow H_1 H_2[p \mapsto v_1 v_2], p} \quad (p \text{ fresh})$$

- ▶ M_1 と M_2 をそれぞれこの順に評価する．グローバルなヒープには評価結果のヒープ差分が蓄積されていく．
- ▶ ポインタ v_1 を参照してヒープからオブジェクトを取り出す．
- ▶ それが関数抽象でない（参照できない場合も含む）ならば、この関数適用は「今は行えない」ので、「関数適用 $v_1 v_2$ を計算すること」をヒープにしまう．
- ▶ ヒープにしまった計算へのポインタを評価結果とする．

評価例

$(\lambda x.\lambda y.(x + (1 + 2)) + y) 3$

評価例

$[], \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$

評価例

$[], \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$

$[], \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$

評価例

$$\square, \{x \mapsto p_1\} \vdash \lambda y.(x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x.\lambda y.(x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x \Rightarrow \square, p_1$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x \Rightarrow \square, p_1$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash 1 + 2 \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x \Rightarrow \square, p_1$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash 1 + 2 \Rightarrow \square, 3$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x \Rightarrow \square, p_1$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash 1 + 2 \Rightarrow \square, 3$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow [p_3 \mapsto p_1 + 3], p_3$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow [p_3 \mapsto p_1 + 3], p_3$$
$$[p_3 \mapsto p_1 + 3], \{x \mapsto p_1, y \mapsto p_2\} \vdash y \Rightarrow$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow [p_3 \mapsto p_1 + 3], p_3$$

$$[p_3 \mapsto p_1 + 3], \{x \mapsto p_1, y \mapsto p_2\} \vdash y \Rightarrow \square, p_2$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow [p_3 \mapsto p_1 + 3], p_3$$

$$[p_3 \mapsto p_1 + 3], \{x \mapsto p_1, y \mapsto p_2\} \vdash y \Rightarrow \square, p_2$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow [p_3 \mapsto p_1 + 3, p_4 \mapsto p_3 + p_2], p_4$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y.(x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash \lambda x.\lambda y.(x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow [p_3 \mapsto p_1 + 3], p_3$$

$$[p_3 \mapsto p_1 + 3], \{x \mapsto p_1, y \mapsto p_2\} \vdash y \Rightarrow \square, p_2$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow [p_3 \mapsto p_1 + 3, p_4 \mapsto p_3 + p_2], p_4$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow [p_5 \mapsto \lambda y. (p_1 + 3) + y], p_5$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash x + (1 + 2) \Rightarrow [p_3 \mapsto p_1 + 3], p_3$$

$$[p_3 \mapsto p_1 + 3], \{x \mapsto p_1, y \mapsto p_2\} \vdash y \Rightarrow \square, p_2$$

$$\square, \{x \mapsto p_1, y \mapsto p_2\} \vdash (x + (1 + 2)) + y \Rightarrow [p_3 \mapsto p_1 + 3, p_4 \mapsto p_3 + p_2], p_4$$

$$\square, \{x \mapsto p_1\} \vdash \lambda y. (x + (1 + 2)) + y \Rightarrow [p_5 \mapsto \lambda y. (p_1 + 3) + y], p_5$$

$$\square, \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow [p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], p_6$$

$$\square, \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$[], \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow [p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], p_6$

$[p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], \emptyset \vdash 3 \Rightarrow$

$[], \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$

評価例

$[], \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow [p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], p_6$

$[p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], \emptyset \vdash 3 \Rightarrow [], 3$

$[], \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$

評価例

$[], \emptyset \vdash \lambda x. \lambda y. (x + (1 + 2)) + y \Rightarrow [p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], p_6$

$[p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], \emptyset \vdash 3 \Rightarrow [], 3$

$[p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], \{x \mapsto 3\} \vdash \lambda y. (x + 3) + y \Rightarrow$

$[], \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow$

評価例

$$\frac{[p_6 \mapsto \lambda x. \lambda y. (x + 3) + y], \{x \mapsto 3\} \vdash \lambda y. (x + 3) + y \Rightarrow}{[], \emptyset \vdash (\lambda x. \lambda y. (x + (1 + 2)) + y) 3 \Rightarrow}$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow [], 6$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow [], 6$$
$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash y \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow [], 6$$
$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash y \Rightarrow [], p_7$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow [], 6$$
$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash y \Rightarrow [], p_7$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow [p_8 \mapsto 6 + p_7], p_8$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow [], 6$$
$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash y \Rightarrow [], p_7$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow [p_8 \mapsto 6 + p_7], p_8$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow [p_9 \mapsto \lambda y.(6 + y)], p_9$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow$$

評価例

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash x + 3 \Rightarrow [], 6$$
$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash y \Rightarrow [], p_7$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3, y \mapsto p_7\} \vdash (x + 3) + y \Rightarrow [p_8 \mapsto 6 + p_7], p_8$$

$$[p_6 \mapsto \lambda x.\lambda y.(x + 3) + y], \{x \mapsto 3\} \vdash \lambda y.(x + 3) + y \Rightarrow [p_9 \mapsto \lambda y.(6 + y)], p_9$$

$$[], \emptyset \vdash (\lambda x.\lambda y.(x + (1 + 2)) + y) 3 \Rightarrow [p_6 \mapsto \lambda x.\lambda y.(x + 3) + y, p_9 \mapsto \lambda y.(6 + y)], p_9$$

従来の自然意味論との違い

- ▶ ヒープの存在
 - ▶ とはいえヒープを持たないインタプリタはないので不自然ではない
- ▶ 「未知の値」もポインタにすることでどんなプログラムもとりあえず評価できる．関数本体も呼び出し前に評価してしまう
 - ▶ 不正な状態 (*Wrong*) にならない．プログラムが止まるなら必ず正当な値が得られる．*Wrong* を導くはずだった誤った計算は「今はできない・やらない」のでヒープにしまわれる．
- ▶ ポインタとポインタ順序を再帰的に辿ることで残余プログラムが得られる
 - ▶ 関数適用の回数と順序を保存しなければならないことに気を付ける．

フルスケールの SML#へのスケールアップ

「ヒープがある」のでポインタ操作を明示する必要があることさえわかっているならば、インタプリタをさくさく書ける。

- ▶ 部分評価器のコアの実装は 1028 行：
<https://github.com/smlsharp/smlsharp/tree/master/src/compiler/compilePhases/partialevaluation/main>
 - ▶ TCOptimization は 1417 行
- ▶ 部分評価をどこまでやるかは実装者の裁量
 - ▶ 大きな関数の適用はやらない
 - ▶ 相互再帰関数間で相互再帰の相手を展開する

評価とまとめ

結局 SML#はどれだけ速くなったか

比較対象：

- (A) SML# 4.1.0 でコンパイルした SML# 4.2.0
- (B) SML# 4.2.0 でコンパイルした SML# 4.2.0

計測対象：

- ▶ SML# 4.2.0 の MatchCompiler.sml のコンパイルにかかる user time
- ▶ 10 回測って平均を求める

計測環境：

- ▶ AMD EPYC 7501 2.0GHz, 128 GB RAM, NixOS unstable (2025-06-13)
- ▶ LLVM 19.1.7

結果：

- (A) 4.919 秒
- (B) 3.270 秒 (-33.5%)

各最適化はどれだけ効いているか

SML# 4.2.0 で最適化オプションを変えてコンパイルした SML# 4.2.0 で MatchCompiler.sml をコンパイルするのにかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果
				6.527 秒
✓				3.716 秒
✓		✓		3.264 秒
✓			✓	3.713 秒
	✓			5.171 秒
	✓	✓		4.385 秒
	✓		✓	4.649 秒

各最適化はどれだけ効いているか

SML# 4.2.0 で最適化オプションを変えてコンパイルした SML# 4.2.0 で MatchCompiler.sml をコンパイルするのにかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果	
				6.527 秒	←基準
✓				3.716 秒	<u>(-43.1%)</u>
✓		✓		3.264 秒	
✓			✓	3.713 秒	
	✓			5.171 秒	(-20.8%)
	✓	✓		4.385 秒	
	✓		✓	4.649 秒	

各最適化はどれだけ効いているか

SML# 4.2.0 で最適化オプションを変えてコンパイルした SML# 4.2.0 で MatchCompiler.sml をコンパイルするのにかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果
				6.527 秒
✓				3.716 秒
✓		✓		3.264 秒
✓			✓	3.713 秒
	✓			5.171 秒
	✓	✓		4.385 秒
	✓		✓	4.649 秒

(-28.1%)

←基準

各最適化はどれだけ効いているか

SML# 4.2.0 で最適化オプションを変えてコンパイルした SML# 4.2.0 で MatchCompiler.sml をコンパイルするのにかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果	
				6.527 秒	
✓				3.716 秒	←基準
✓		✓		3.264 秒	<u>(-15.2%)</u>
✓			✓	3.713 秒	(- 0.1%)
	✓			5.171 秒	←基準
	✓	✓		4.385 秒	<u>(-12.2%)</u>
	✓		✓	4.649 秒	(-10.1%)

各最適化にどれだけ時間がかかるか

SML# 4.2.0 で最適化オプションを変えて MatchCompiler.sml をコンパイルするのにかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果
				2.879 秒
✓				2.757 秒
✓		✓		3.259 秒
✓			✓	2.811 秒
	✓			2.673 秒
	✓	✓		3.212 秒
	✓		✓	2.712 秒
参考：SML# 4.1.0 デフォルト				4.161 秒

各最適化にどれだけ時間がかかるか

SML# 4.2.0 で最適化オプションを変えて MatchCompiler.sml をコンパイルするのにかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果	
				2.879 秒	←基準
✓				2.757 秒	<u>(-4.2%)</u>
✓		✓		3.259 秒	
✓			✓	2.811 秒	
	✓			2.673 秒	(-7.2%)
	✓	✓		3.212 秒	
	✓		✓	2.712 秒	
参考：SML# 4.1.0 デフォルト				4.161 秒	

各最適化にどれだけ時間がかかるか

SML# 4.2.0 で最適化オプションを変えて MatchCompiler.sml をコンパイルするのにかかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果	
				2.879 秒	
✓				2.757 秒	←基準
✓		✓		3.259 秒	<u>(+18.2%)</u>
✓			✓	2.811 秒	(+ 2.0%)
	✓			2.673 秒	←基準
	✓	✓		3.212 秒	<u>(+20.2%)</u>
	✓		✓	2.712 秒	(+ 1.5%)
参考：SML# 4.1.0 デフォルト				4.161 秒	

各最適化にどれだけ時間がかかるか

SML# 4.2.0 で最適化オプションを変えて MatchCompiler.sml をコンパイルするのにかかった user time

TailCallComp	UncurryOpt	PartEval	TCOpt	測定結果
				2.879 秒
✓				2.757 秒
✓		✓		3.259 秒
✓			✓	2.811 秒
	✓			2.673 秒
	✓	✓		3.212 秒
	✓		✓	2.712 秒
参考：SML# 4.1.0 デフォルト				4.161 秒

(-21.7%)

←基準

まとめ

SML#めっちゃ速くなりました！

- ▶ ぜひ使ってみてね！

言語処理系を作る視点から以下のトピックを説明しました．

- ▶ マルチコア CPU でベストパフォーマンスを出す
 - GC がボトルネック．世界を止めない並行並列 GC の開発
- ▶ Aarch64 対応
 - 末尾再帰への対応．やれることを全部やる
- ▶ 部分評価に基づくコードの最適化
 - 自然意味論に基づくインタプリタのように実装してみた