# Shibuya.lisp Tech Talk #1

- fujita-y
  - 藤田善勝
  - Ypsilon の開発者
  - リトルウイング代表取締役プログラマー
  - http://code.google.com/p/ypsilon/
  - http://d.hatena.ne.jp/fujita-y
  - http://www.littlewing.co.jp/

# Ypsilon について

- ピンボールゲーム組込みのために開発している R6RS 準拠の Scheme インタープリタ。
- マルチコアに最適化したコンカレント GC 。
- 極めて短い GC 停止時間と並列実行によるパフォーマンスの向上。
- システム全体のレスポンスと信頼性を重視。
- VM の並列化は現在進行中。
- ネイティブコード生成は並列化安定後に予定。

# YpsilonVM の開発

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# YpsilonVM の開発

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# 基本的な最適化

- 対象となる言語、実装に使用する言語を問わずインタープリタの VM で広く使われている手法。
  - 資料も実装も豊富に存在する。
  - ただし古い資料は想定している CPU の特性が今日のものとはことなることに注意する必要がある。

# Direct Threading

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Direct Threading

- Dispatch のコストを少なく
  - 間接分岐の分散により分岐予測のヒット率が上がる。
  - 分岐に必要なコードが少ないので I-Cache の利用効率が上がる。
  - すべての Scheme コードに効果がある。

# Direct Threading

- 実装
  - Ypsilon は高速化のため AST に VM 命令処理コードのアドレスを直接書き込んでいる。
  - VM 命令処理コードはアラインメントの調整により他の Scheme オブジェクトとの識別できるようにしている。

# Direct Threading

- YpsilonVM の AST

```
((close (2 0 . filter)
   (extend.enclose+
     (1 0 . loop)
     (iloc.0 . 0)
     (if.null?.ret.const)
     (call
       (push.car.iloc (0 . 0))
       (apply.iloc (2 . 0)))
     (if.true
       (push.car.iloc (0 . 0))
       (call
         (push.cdr.iloc (0 . 0))
         (apply.iloc+ (1 . 0)))
       (ret.cons))
     (push.cdr.iloc (0 . 0))
     (apply.iloc+ (1 . 0)))
   (push.iloc.1 . 1)
   (apply.iloc+ (0 . 0)))
 (set.gloc.of filter)
 (ret.const.unspec))
```

# Direct Threading

- コードアドレスの調整
  - 次のマクロが一連のコードを挿入することにより各 VM 命令処理コードのアドレスの下位 3 ビットが 100 となる。これで他のオブジェクトとの識別を行っている。

```
#define CASE(code) M_##code: \
                   __asm__ ("ud2"); \
                   __asm__ (".p2align 3"); \
                   L_##code: \
                   __asm__ ("nop"); \
                   __asm__ ("nop"); \
                   __asm__ ("nop"); \
                   __asm__ ("nop"); \
                   __asm__ ("/* "#code" */");
```

# Direct Threading

- prefech 抑制
  - “ud2” は x86 が未定義命令例外を発生する命令。この命令を読むと CPU はそれ以降の命令の Prefetch を停止する。これにより無駄な Bus Traffic と I-Cache の使用を抑制する。

```
#define CASE(code) M_##code: \
                     __asm__ ("ud2"); \
                     __asm__ (".p2align 3"); \
                     L_##code: \
                     __asm__ ("nop"); \
                     __asm__ ("nop"); \
                     __asm__ ("nop"); \
                     __asm__ ("nop"); \
                     __asm__ ("/* "#code" */");
```

# Direct Threading

- その効果は?
  - tak で約 7.8% 、takl で約 11.8% の高速化

```
;; tak    (x200)
;; 0.813437 real    0.812050 user    0.000000 sys
;; tak    (x200)
;; 0.750093 real    0.752047 user    0.000000 sys

;; takl   (x35)
;; 0.833171 real    0.832052 user    0.000000 sys
;; takl   (x35)
;; 0.734454 real    0.732046 user    0.000000 sys
```

# Super Instruction

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Super Instruction

- プログラムに必要な VM 命令数を少なくすることにより VM で一番コストの高い Dispatch の回数を少なくすることができる。
- D-Cache の利用効率が上がる。
- 各 VM 命令の処理コードが長くなることにより C/C++ コンパイラによる最適化を期待できる。

# Super Instruction

- Ypsilon では
  - 現在 93 個ある VM 命令はコンパイラでの最適化も考えて手動で作成されている。
  - コンパイラは Super Instruction を直接出力する。命令結合ステージは存在しない。
  - コンパイラがすべての VM 命令を知っているため効率のよいコードを高速に生成することができる。

# Super Instruction

- Super Instruction のトレードオフ
  - かなり頻繁に使用されるコードでなければ実行コードの局所性が失われることになり逆効果をもたらす。
  - 例えば fib の Super Instruction を作成して VM に組み込めば fib の計算はとても速くなる。しかしそのコードは VM 実行時のコードの局所性を下げるので fib 以外のすべての処理で VM の性能を下げることになる。
  - 良い命令の組み合わせを得るのはとても難しい。単純な統計情報により組み合わせを決定することはできない。

# Operand Fusion

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Operand Fusion

- Operand の固定された特化命令。
  - 動的言語では Operand のタイプチェックが必要なくなるというボーナスがある。
  - Super Instruction と同じような効果とトレードオフがある。
  - Super Instruction と Operand Fusion を単純に組み合わせると VM 命令数が爆発するので使いどころは限られる。

# Cache Access

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Cache Access

- 近年の CPU ではメモリアクセスがボトルネック。
- メモリアクセスが絡むと極端な性能低下が発生することがある。
- その低下の度合いは先の最適化の効果を帳消しにするほどのものがある。

# Cache Access

- 例えば vm.h にあるコードの下記の部分を一行だけ移動して開発用マシンでベンチマーク。

変更前

```
int         m_stack_size;
int         m_stack_busy;
void*       m_dispatch_table[VMOP_INSTRUCTION_COUNT];
```

変更後

```
void*       m_dispatch_table[VMOP_INSTRUCTION_COUNT];
int         m_stack_size;
int         m_stack_busy;
```

# Cache Access

- takl の実行時間が 1.34 倍とかなり遅くなる。

**変更前**

```
;;  takl    (x35)
;;  0.734454 real    0.732046 user    0.000000 sys
;;  ------------------------------------------------------------
```

**変更後**

```
;;  takl    (x35)
;;  0.985471 real    0.984061 user    0.000000 sys
;;  ------------------------------------------------------------
```

# Cache Access

- この変更はすべてのベンチマークで速度を低下させるものではない。速度の上がるものも存在する。
- 原因は D-Cache  の Cache Line コンフリクト。
- Intel Core などの L2 Cache は Code と Data が共通なのでこの問題が起きやすい。
- すべての局面でこれの発生を抑えるのは VM の規模を考えると無理。
- どうするのか?

# Cache Access

- 多様なベンチマークを使用して Cache Line コンフリクトが広範囲に発生して VM 全体の性能を低下させているのか、それとも限定的な発生で一部のプログラムだけに発生するのかを判定する。
- 広範囲に発生する場合は優先的に解消を図る。
- 限定的な場合は基本的に放置する。これは CPU 、コンパイラ、メモリー構成、将来のコード変更などで状況が変化するものであるからである。

# Cache Access

- VM では I-Cache が一般プログラム以上に重要。
  - VM 命令の Dispatch や SUBR の呼び出しなどは間接分岐で実装されるため分岐予測が効きにくい。
  - そのため投機的 Prefetch が有効に働かない。
  - したがって CPU は I-Cache に直接命令を読みに行くことが多くなる。
  - 命令が I-Cache になければ大きなペナルティが発生。

# YpsilonVM の開発

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Scheme 専用の最適化

- YpsilonVM の最適化コンセプト
  - ポータブルであること。
  - 十分なメンテナンス性があること。
  - デバッグ機能が充実していること。
  - 不得意とするコードが無いこと。
  - コンカレント GC が効果的に動くこと。
  - その上で出来るだけ速く。

# Scheme 専用の最適化

- デバッグ機能の充実
  - 末尾呼び出しの先でエラーが発生した場合にも最小限のコストで有効な Backtrace を可能にする。
  - Scheme は末尾呼び出しで継続を保存しない。したがって何か工夫をしないとスタックから末尾呼び出しの情報が欠落してしまう。
  - 末尾呼び出しが重要なスタイルである Scheme においてその情報が得られなければデバッグが難しくなる。

# Scheme 専用の最適化

- Guile のデバッグモードは Backtrace で末尾呼び出しの記録を表示するがソースコードの情報は失われている。

```
guile> (define wrong-sum
         (lambda (lst)
           (let loop ((lst lst) (acc 0))
             (if (null? lst)
                 (car acc) ; acc
                 (loop (cdr lst) (+ acc (car lst)))))))

guile> (wrong-sum '(1 2 3 4))

Backtrace:
In standard input:
  10: 0* [wrong-sum (1 2 3 4)]
   3: 1  [loop (1 2 3 4) 0]
   ...
   5: 2  [car {10}]

standard input:5:13: In procedure car in expression (car acc):
standard input:5:13: Wrong type (expecting pair): 10
ABORT: (wrong-type-arg)
```

# Scheme 専用の最適化

- Ypsilon は標準でソースコードの情報を含めた Backtrace を可能にする。

```
> (define wrong-sum
    (lambda (lst)
      (let loop ((lst lst) (acc 0))
        (if (null? lst)
            (car acc) ; acc
            (loop (cdr lst) (+ acc (car lst)))))))

> (wrong-sum '(1 2 3 4))

error in car: expected pair, but got 10

backtrace:
  0  (car acc)
  ..."/dev/stdin" line 5
  1  (loop (cdr lst) (+ acc (car lst)))
  ..."/dev/stdin" line 6
  2  (wrong-sum '(1 2 3 4))
  ..."/dev/stdin" line 1
```

# Scheme 専用の最適化

- しかし末尾呼び出しの Backtrace を可能とするには追加のデバッグ情報が必要になる。
- エラーが発生しなければ使われないデバッグ情報に大きなコストを支払いたくない。
- どうするのか?

# Stack GC

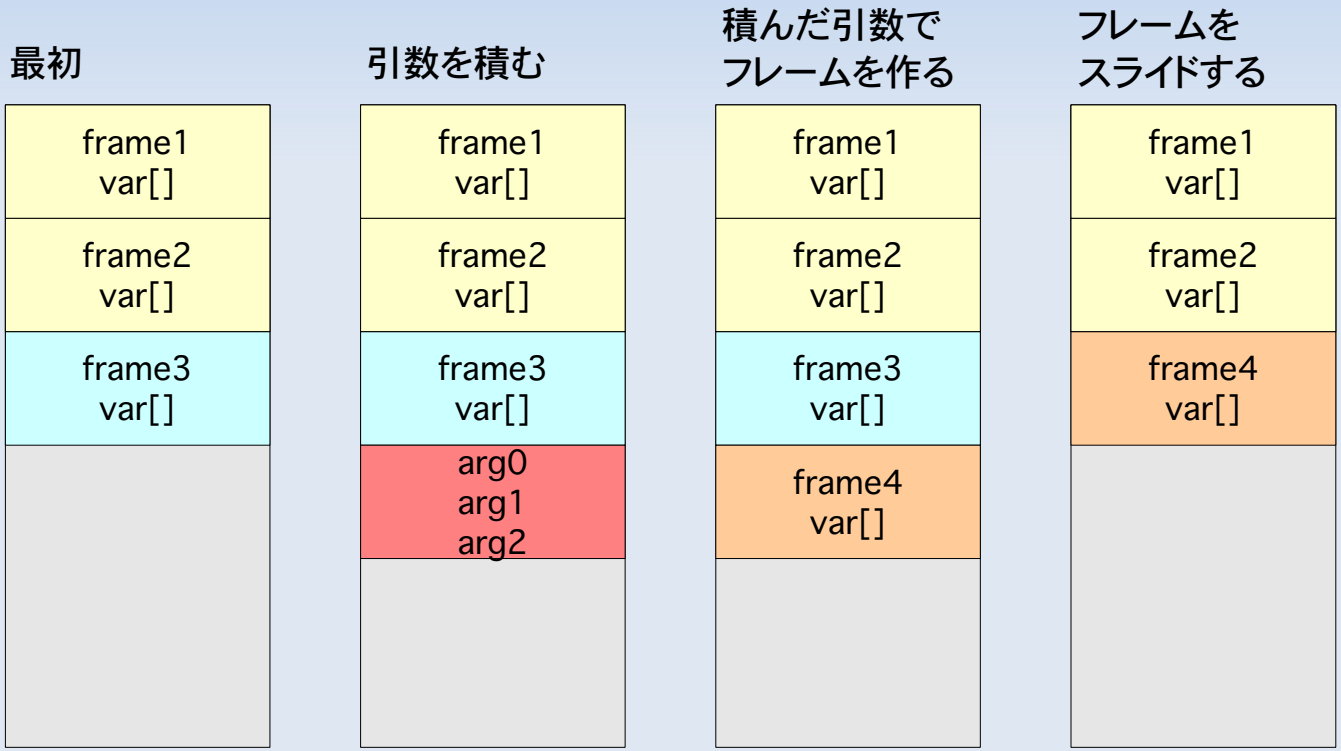
- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Stack GC

- Stack GC の導入
  - 末尾呼び出しを最適化すると同時に末尾呼び出し時に行うデバッグ情報のセーブに必要なオーバーヘッドを小さくする。
  - さらに Stack Closure による最適化も可能にする。

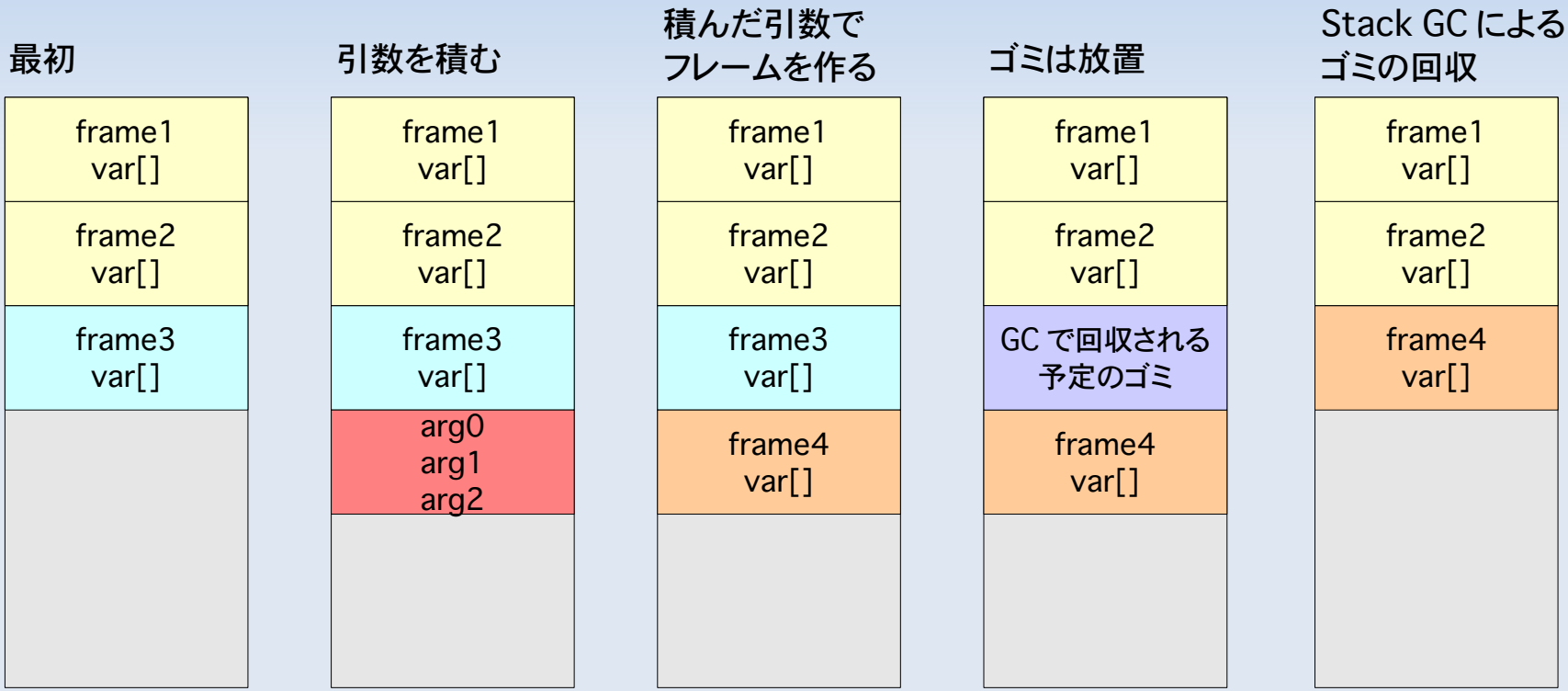
# Stack GC

- フレームのスライドによる末尾呼び出し



# Stack GC

- Stack GC による末尾呼び出し



# Stack GC

- Stack GC 単体で見るトレードオフ
  - 欠点
    - 末尾再帰のループパターンでも GC が発生する。
    - GC の処理に時間がかかる。
  - 利点
    - 末尾呼び出しごとのフレームのコピーを省略できる。
    - 通常のコールと末尾呼び出しが同じコードになる。これによりコールの処理を複雑にしても VM の実効速度への影響を小さくできる。

# Stack GC

- Stack GC の性能評価
  - 末尾呼び出し以外の呼び出しがスタック中に残っていると GC によって回収できるフレームが少なくなる。
  - 一定数以上そのようなフレームが存在すると GC によるオーバーヘッドがフレームのコピーによるオーバーヘッドを上回る。
  - 速度的には単体ではあまり効果があると言えない。次に述べる Stack Closure と組み合わせることによって大きな効果を発揮する。

# Stack Closure

- 基本的な最適化
  - Direct Threading
  - Super Instruction
  - Operand Fusion
  - Cache Access
- Scheme 専用の最適化
  - Stack GC
  - Stack Closure

# Stack Closure

- Stack Closure
  - 自由変数を持つ Closure でも生存期間の解析によりヒープに作成する必要がないと判定できるものがある。その場合は Stack の中だけですべての処理を完結させる。
  - 通常の Closure の作成では Stack からヒープに環境をコピーする必要が出てくる。コピーのコストはもちろんであるが、それ以降ヒープにある環境をアクセスすることによるペナルティーの影響も大きい。
  - それらのコストをすべて取り払うことができる。

# Stack Closure

- 下のリストで loop は自由変数 pred を持っている。
- 解析により loop と pred の生存期間が同じであると確認できるので loop は Stack Closure となる。

```
(define filter
  (lambda (pred lst)
    (define loop
      (lambda (lst)
        (cond ((null? lst) '())
              ((pred (car lst))
               (cons (car lst) (loop (cdr lst))))
              (else
               (loop (cdr lst))))))
    (loop lst)))
```

# Stack Closure

- Stack Closure の実装は生存期間の解析ができれば簡単である。この例では loop が何度どこから呼ばれても pred のスタック上での位置が一定であることを利用している。

```
(define filter
  (lambda (pred lst)
    (define loop
      (lambda (lst)
        (cond ((null? lst) '())
              ((pred (car lst))
               (cons (car lst) (loop (cdr lst))))
              (else
               (loop (cdr lst))))))
    (loop lst)))
```

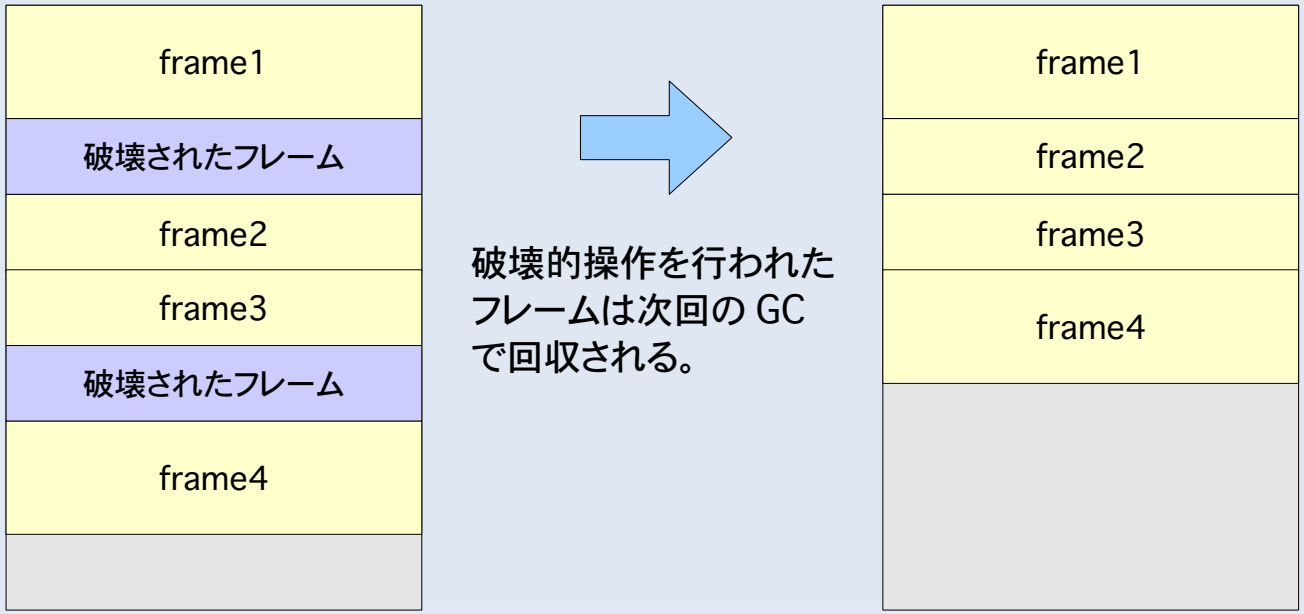
# Stack Closure

- Stack Closure は適応範囲が広い。
  - loop への呼び出しは末尾である必要がない。
  - pred に対する破壊的操作があってもよい。

```
(define filter
  (lambda (pred lst)
    (define loop
      (lambda (lst)
        (cond ((null? lst) '())
              ((pred (car lst))
               (cons (car lst) (loop (cdr lst))))
              (else
               (loop (cdr lst))))))
    (loop lst)))
```

# Stack Closure

- ただし効率の良い実装には Stack GC が必要。
  - これはフレームのリンク構造に修復不能な破壊的操作を行うことにより実装を行うからで、リンク構造が破壊されていてもゴミであれば問題なく回収できる機構が必要となるからである。



# Stack Closure

- その効果は?
  - この例では loop の Stack Closure を作成するコストは (let ((loop 'thing)) expr ...) とほぼ同じ。

```
(define filter
  (lambda (pred lst)
    (define loop
      (lambda (lst)
        (cond ((null? lst) '())
              ((pred (car lst))
               (cons (car lst) (loop (cdr lst))))
              (else
               (loop (cdr lst))))))
    (loop lst)))
```

# Concurrent GC

- Concurrent GC の考え方はとても単純。

# Concurrent GC

- Mark Phase
  - Mutator を止める。
  - ルート集合をとる。
  - Mutator を再開する。
  - Mutator の実行と並列に Collector が Mark を行う。
  - Mutator を止める。
  - Collector が変更のあったルート集合の Mark を行う。
  - Mutator を再開する。

# Concurrent GC

- Write Barrier
  - Mutator と Collector が並列に動作しているときは破壊的操作が行われた時にマーク漏れが発生しないようにする必要がある。
  - 何種類か方法があるが、基本的に上書きされたオブジェクト、または上書きしたオブジェクトのアドレスを覚えておいて後でマークするだけ。

# Concurrent GC

- Sweep Phase
  - ゴミはもう確定している。
  - Mutator はゴミにアクセスしない。
  - Collector はヒープをスキャンしてゴミを回収。

# Concurrent GC

- とても簡単そうに見える。
- 実は Scheme の場合にはいろいろアドバンテージがあるので実装は本当に簡単。
- pthread 関連の方が面倒に思えるくらい。
- 約一週間ほどで Concurrent GC の実装は完了。

# Concurrent GC

で終われると思ったのは
甘かった orz

# Concurrent GC

- 簡単に作った実装はどうなったか?
- 2コアのマシンでベンチマークを行うと普通の GC よりかなり遅い。
- コアを1つ停止してベンチマークを行うと普通の GC よりちょっと遅い。
- つまり2コアより1コアで実行した方が全然速く、しかも普通の GC にも負けるという役に立たない代物。orz

# Concurrent GC

- 調べてみるとMutatorとCollectorが激しい競合を起こしていた。
- どうやらMark Phase中のメモリアロケーションが重い。
- それはなぜか?

# Concurrent GC

- Mark Phase 中に Mutator がアロケーションを行うと何が起こるのか考えてみる。
  - Mutator がアロケーションを要求する。
  - Snapshot-at-beginning ではメモリーのアロケーションと同時に Mark 動作も行わなければならない。
  - Mark 動作をしている Collector と Mutator が Mark ビットの取り合いをするので競合が発生する。

# Concurrent GC

- アロケーションレートが高い時は Collector が動き続けている。そこにまた Mutator が大量にアロケーション要求を行うのだから大量に競合が発生する。
- どんどんメモリをアロケートして次々にゴミを作る Scheme にはこれは大きな問題。
- どうするか?

# Concurrent GC

- Incremental Update 方式に変更
  - この方式では新規のアロケーションには特別な処理が必要ない。とても Scheme に向いていると言える。
  - 欠点は GC 終了時の停止でルートをもう一度スキャンしなければならなくなること。
  - Snapshot-at-beginning では VM レジスターや VM スタックくらいしかスキャンする必要がなかったのだが、背に腹はかえられない。

# Concurrent GC

- どうなったか?
  - アロケーションでの競合が少なくなることによって Mutator の動作が軽くなる。
  - ところが、それにより別の問題が発生する。
  - Mutator が速くなるということは、それだけ時間あたりのアロケーションも多くなるということ。
  - これによりスタベーションが発生。

# Concurrent GC

- スタベーションとは?
  - メモリの消費に回収が追いつかなくなること。
  - 調べてみると今度は Sweep Phase での競合が目立つ。
  - つまりメモリの回収が遅すぎるということ。
  - しかし Sweep Phase では Collector はゴミにしか関心がなく Mutator はゴミ以外にしか関心がないはず。
  - なぜなのか?

# Concurrent GC

- Concurrent Sweep では Collector に間違ってメモリが回収されないように処理を行う必要がある。具体的にはまだ Collector の巡回が終わっていない場所からメモリを確保する場合には Mark ビットを立ててやる必要がある。
- Collector は Mark ビットをゴミ判定のために見ているので競合が発生する。
- どうするか?

# Concurrent GC

- Collector が現在 Sweep を行っている領域からはアロケーションが行われないようにする。
- そうすれば Mark  ビットの競合はなくなる。
- どうなったか？

# Concurrent GC

- 競合が少なくなったので Collector の回収速度が上がり、同時に VM 全体の動作も速くなった。
- VM 全体の動作が速くなると VM で実行する Scheme プログラムも速く走るようになり時間あたりのアロケーション要求が増加した。
- そして再びスタベーションが発生。

# Concurrent GC

これって
「イタチごっこ」
って奴ですか！

# Concurrent GC

- とにかくアロケーションの量を減らさないと根本的な解決にならないと目先をかえて VM を最適化。
- アロケーションの量が減るにしたがってスタベーションも解消の方向に。

# Concurrent GC

- いろいろなアイディアで VM をがんがん最適化していると VM の実行性能も上がってくる。
- Scheme プログラムの実行も速くなる。
- 結局またスタベーションが発生することに・・・ orz

# Concurrent GC

このイタチごっこに
終りはくるのでしょうか？

# Concurrent GC

次回予告
「役に立たないスピンロック」

# Shibuya.lisp Tech Talk #1

- fujita-y
  - 藤田善勝
  - Ypsilon の開発者
  - リトルウイング代表取締役プログラマー
  - http://code.google.com/p/ypsilon/
  - http://d.hatena.ne.jp/fujita-y
  - http://www.littlewing.co.jp/

2008/10/18