# Shibuya.lisp Tech Talk #2

- 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。

# 前回の続き

- コンカレント GC において一番の難題はメモリスタベーション。これを解決できなければ実用にはならない。

# メモリスタベーションとは?

- メモリの消費に GC による回収が追いつかなくなること。
- VM を改良していくと、時間あたりのメモリ消費量がどんどん増えていく。
- マルチスレッドになればメモリ消費は簡単に数倍になる。ネイティブコードになれば、さらに数倍になると予想される。

# メモリスタベーションとは?

- しかし、コレクターは C++ で書かれている。なので単純なコードの最適化によるメモリ回収能力の上昇余地はあまりない。
- プロファイルを取ってみるとミューテクスで激しい競合が起きている。
- そこでスピンロックを試してみることにした。

# スピンロック

- ユーザースペースで空回りして、システムに一切制御を移さないタイプを実装。
- システムコール関連のオーバーヘッド削減を期待。
- が、ぜんぜん役にたたなかった・・・ orz
- なぜか?

# スピンロック

- 考えてみれば当たり前のことなのだが、元々激しい競合を起こしているわけだから。スピンロックだろうがミューテックスだろうがロックは取れないということ。
- つまりシステムに待たされるか、ユーザースペースで空回りして待っているか程度の違いしかない。
- しかし、ここで不思議なことが・・・

# スピンロック

- 同じくらいということなら納得できるが、ミューテックスよりもスピンロックの方が明らかに遅くなるテストプログラムがある。
- よく調べてみるとフリーリストの作成に時間が掛かるようになっている。

# フリーリスト

- Ypsilon では新しいメモリブロックを確保するタイミングでそのブロック内にフリーリストを作成している。
- フリーリストの作成はコレクターの仕事。ミューテーターはフリーリストが完成するまでそのブロックにはアクセスしないのでロックは必要ない。
- ここでは競合は起こらないはずなのに・・・

# バスのロック

- スピンロックは繰り返しコンペア&スワップ命令でバスをLOCKしている。ミューテクスも内部でLOCKするけれど、その頻度は違う。
- 最初はバスのLOCKが重いのかと考えた。
- しかし、それならばすべての同期でオーバヘッドがあるはず。フリーリストの作成時が目立つのは別の理由があるはずと考えた。

# メモリのシリアライズ

- x86 ではバスの LOCK は保留中のすべてのメモリ書き込みが終了するまで次の命令を実行しないという強力な副作用をもっている。
- フリーリストの作成とはリンクリストの作成。この時には千個単位の書き込みが発生する。CPU はストアバッファとその空きを待つ命令を大量に抱えた状態になるのではないか?
- だとすれば・・・

# メモリのシリアライズ

- その状態でバスの LOCK によるメモリのシリアライズが発生すれば被害は甚大。目に見えてパフォーマンスが落ちるのも想像できる。
- つまり、このケースではアトミック命令を使ったロックフリーな同期アルゴリズムも役に立たず、メモリーバリアですら影響が大きい可能性があると考えられる。
- それなら、どうする?

# 同期しなければいい?

- 共有変数が同期していなくてもミューテーターとコレクタが協調動作できれば問題は解決。
- すべての局面でそうするのはさすがに大変。でも、ボトルネックとなる部分だけを頑張ればなんとかなるかもしれない。
- その方針でベースの部分からメモリ管理部分の設計を見直して作り直し。
- どうなったか?

# ベンチマーク

- Intel Core2 でメモリアロケーションの大量発生する paraffins(gambit-benchmarks) を測定。

```
Linux 2.6.24-23-generic #1 SMP x86_64 GNU/Linux

;;  paraffins (x100)
;;  0.487823 real    0.928058 user    0.000000 sys
```

- 2 コアで CPU 利用率 190% を達成。
- 同時に極めて短い GC 停止時間も。
- 本当にそんなに上手くいっているのか?

# Jello.scm

- Jello.scm はコンカレント GC の性能測定のために作ったベンチマークプログラム。
- Ypsilon の開発目的であるゲーム組み込みを意識して SDL と OpenGL を組み合わせて作成。

# これで十分なのか?

- Jello.scm ではコレクターが作業を終えるのに 20ms ちょっとの時間が必要。複雑なゲームでライブオブジェクトが増えればこの時間はどんどん長くなる。メニーコア・マルチスレッド・ネイティブコードの組み合わせでスタベーションを回避できるのかどうか不安。
- そこで、次の一手も用意することにした。

# VM のマルチスレッド対応

- Jello-spawn.scm は VM のマルチスレッド機能を使った Jello.scm 。
- コレクター部分の変更は一切なしで、活動時間が8分の1くらいに減少。
- これだけ短くなるならコンカレント GC はいらない?・・・とも考えられるのですが。
- それについては・・・

# コンカレント GC

^ ^;

# R6RS について

- R6RS の大きな特徴の一つはライブラリシステム。
- 基本的にプログラムはすべてライブラリ。
- トップレベルプログラムと呼ばれるものもライブラリの変種。内部の処理はほとんど一緒。
- 当然ライブラリも REPL を使ってインタラクティブに開発したい。
- しかし・・・

# R6RS の REPL について

- R6RS では REPL については何も決まっていない。
- これは、例えば emacs で入力した式を "C-x C-e" で評価しながら開発をするという従来の手法については考えられていないということ。
- したがって REPL については、各処理系で独自の拡張として実装されている。（注：plt-r6rs は REPL のサポートなし）

# REPL の現状

```
(library (foo)
  (export bar)
  (import (rnrs))
  (define (bar x) (list-sort < x)))
(import (foo))
(bar '(1 4 2 5 3))
```

- 例えば Ikarus では (library (foo) ...) を "C-x C-e" などで再評価することを許していない。Larceny は ERR5RS モードでのみ可能。
- もっと柔軟に REPL でライブラリを使えるようにしたい。
- 自前のライブラリシステムを開発することに。

# ところが

- コアで定義されているライブラリシステムが、標準ライブラリで定義されている syntax-case と事実上切り離せなくなっていることに気がつく。
- どういうことなのか?

# たとえば

```
(library (access)
  (export foo)
  (import (rnrs))
  (define login-name 'shibuya)
  (define login-password 'tokyo)
  (define-syntax foo
    (lambda (x)
      (syntax-case x ()
        ((_ y)
         (datum->syntax #'z (syntax->datum #'y)))))))
```

- ここで export されている foo はどんな機能を提供するものだと思いますか?

# Ikarus, Larceny では

```
(library (access)
  (export foo)
  (import (rnrs))
  (define login-name 'shibuya)
  (define login-password 'tokyo)
  (define-syntax foo
    (lambda (x)
      (syntax-case x ()
        ((_ y)
         (datum->syntax #'z (syntax->datum #'y)))))))

(import (access))
(foo login-name) ;=> shibuya
(foo login-password) ;=> tokyo
(foo list) ;=> <#procedure list>
```

# Syntax-Case とライブラリ

- foo の動作については未定義かと思われる。
- でも、Ikarus, Larceny の動作は‥‥微妙 ^^;
- Ypsilon は syntax-case も自前の実装にして、ライブラリのフルコントロールを確保することに。

# Syntax-Case の実装

- しかし syntax-case を使ったプログラムを書いたことがないのに、いきなり実装しようというのは、無謀だったかもしれません・・・
- 実際の所、えらい目にあいました・・・ orz
- 一番の問題はドキュメントが少ないこと。
- 詳細は長くなりますので、またの機会に :)

# Syntax-Case の実装

- でも、システムの内部をすべて掌握したことにより、コンパイル結果を動的にキャッシュするといった仕組みなども簡単に実装して試すことができるようになりました。
- そして、さらにこんなことも・・・

# Dead object elimination

- プログラムの実行に必要のないオブジェクトを判別して自動的にそれらを解放する。
- 通常の GC と違い、ライブラリやトップレベルの束縛も必要なければすべて解放してしまう。
- コンパイラなども必要なければ解放してしまうので、メモリの使用量がかなり少なくなる。
- どのくらい?

# Dead object elimination

- 効果
  - ライブオブジェクトが少なくなることにより、GC の負荷が低減され、パフォーマンスが上がる。
  - スクリプトのロード時間を短縮できる。
  - つまり、完成したスクリプトの実行環境を作るのにとても向いている。組み込み時に最適。（注：まだヒープをフリーズする機能がないので、実用にはなっていません）

# Syntax-Case の実装

かなり時間使ったけど・・・

元は取ったと思いたい　^^;

# Shibuya.lisp Tech Talk #2

- 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/