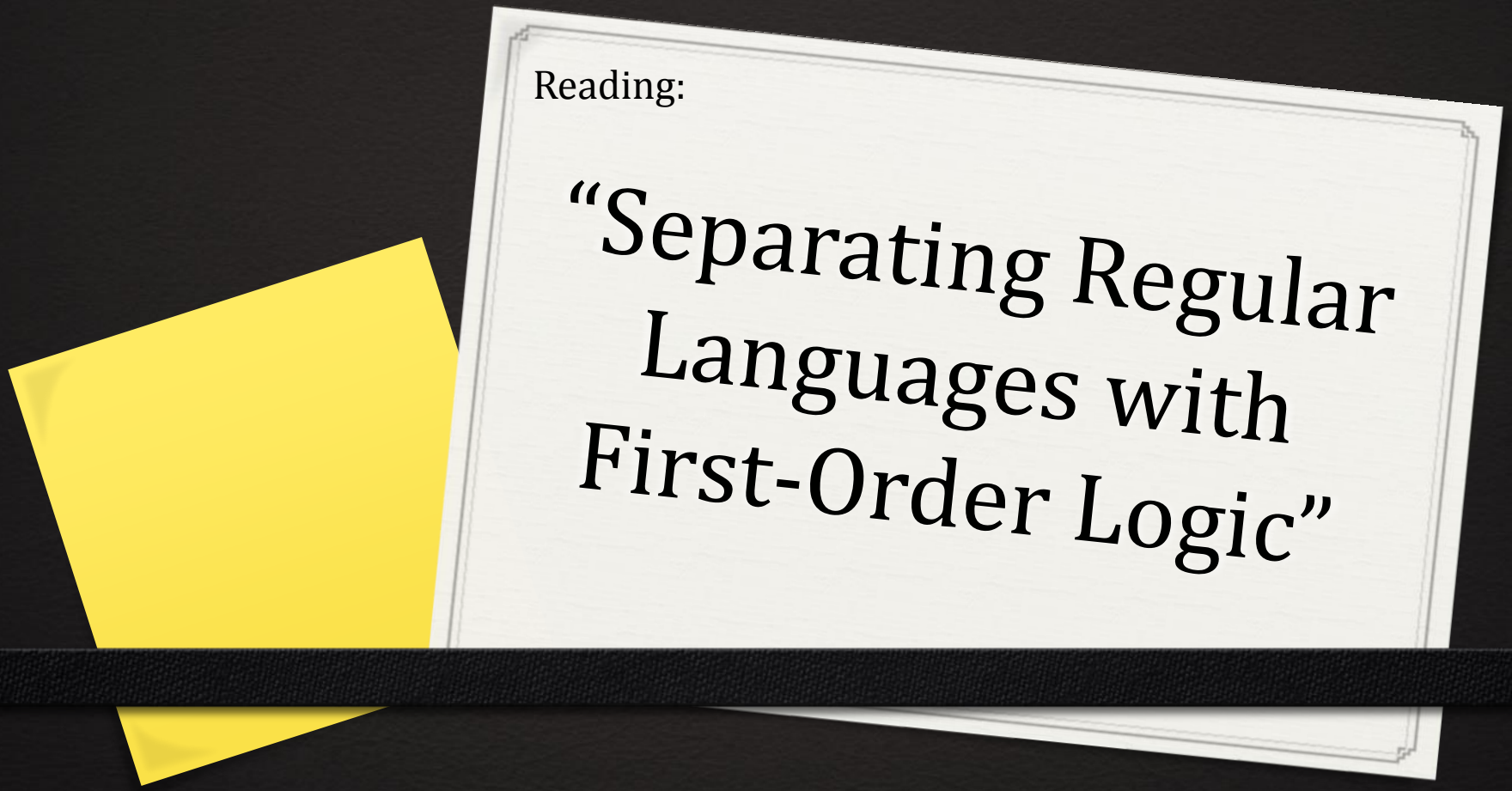
Paper Reading Party 2014
Speaker: @kinaba

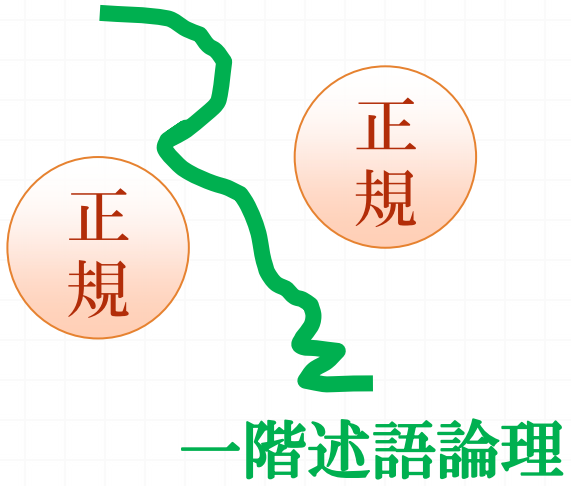
1: 読み終われなかった論文



論文概要

- **“Separating Regular Languages with First-Order Logic”**
  - LICS (Logic in Computer Science) 2014

- キーワード:
  - Ehrenfeucht–Fraïssé ゲーム
  - 正規言語
  - 冪等元



2: 読んで楽しんだ論文

Reading:

"Zombie Swarms: An Investigation of the Behavior of Your Undead Relatives"

論文概要

- **“Zombie Swarms: An Investigation of the Behavior of Your Undead Relatives”**
  - FUN (Fun with Algorithms) 2014

- キーワード:
  - ゾンビ

3: 読んだ紹介論文

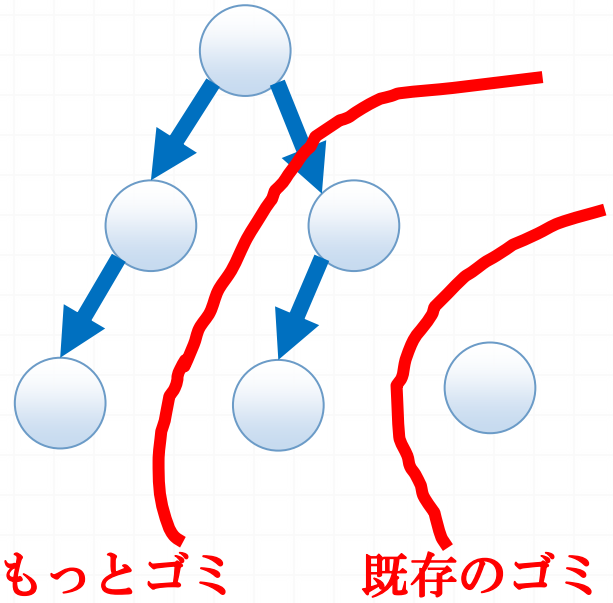
Reading:

"Liveness-Based Garbage Collection"

論文概要

- **“Liveness-Based Garbage Collection”**
  - CC (Compiler Construction) 2014

- キーワード:
  - ガベージコレクション
  - 文脈自由言語
  - 逆元



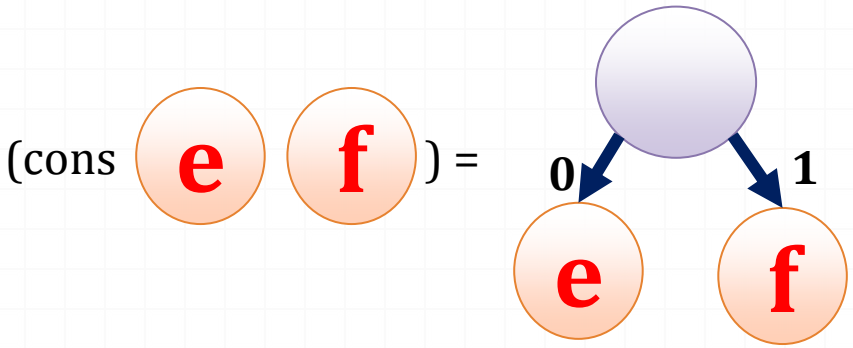
Garbage Collection

- プログラムの実行中に、「もう不必要になった」メモリ領域を解放して再利用する機構

- 「もう不必要になった」とは？
  - 既存手法：今のスタック変数(など)からポインタをどれだけ辿ってもたどり着けない領域のこと
  - 提案手法：プログラムを解析して、**今後のプログラム実行で決して触らないと判断できる領域は到達可能でも容赦なく解放する**

対象言語

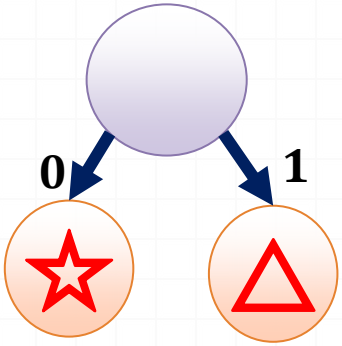
- (cons e f)
  - メモリを確保して、
    ポインタ0でeを、
    ポインタ1でfを指す
- (car e)
  - eのポインタ0をたどった先の値を返す
- (cdr e)
  - eのポインタ1をたどった先の値を返す
- if 文, 関数定義, 再帰
- メモリの書き換え（破壊的代入）は無し



例

.... (car (cons ☆ △)) ...

- このプログラムを実行中。☆を実行。△を実行。そして cons を実行。
- メモリ状態 =



- **△ 部分は今後絶対触らない !!!捨てろ!**

もう少しフォーマルに例

- 今注目している式

  ....(car ☆)....

- の返値には、今後プログラムの実行中にポインタ

  $L \subseteq \{0,1\}^*$

- でしかアクセスしないことがわかっているとする

➔ car の引数 ☆ の値には、ポインタ {0w | w∈L} でしかアクセスしない。 (ポインタ1側は捨てて良い)

式での「逆算」

- (car ☆) に L でアクセス ➔ ☆には {0w | w∈L} で。
- (cdr ☆) に L でアクセス ➔ ☆には {1w | w∈L} で。
- (cons ☆ △) に L でアクセス ➔
  - ☆ には {w | 0w ∈ L} でアクセス。
  - △ には {w | 1w ∈ L} でアクセス。
- (if ☆ then △ else ◇) に L でアクセス ➔
  - ☆ には {ε} でアクセス
  - △ には L でアクセス
  - ◇ には L でアクセス

関数定義

0 f(x) :=

..... if ..x.. then ... x ....

f(x) の返値に L でアクセス

➔ 引数xには、

$\cup_{\text{xの出現}}$ “関数定義全体にLでアクセス”から逆算したxへのアクセス

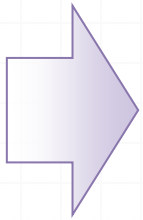
※ 再帰関数がある場合この定義も再帰的になる

文脈自由文法

- f(x) := car(cdr(x))
- g(x) := f(f(x))

f と g の「逆算」関数は

- [f](L) = {10w | w∈L}
- [g](L) = [f]([f](L))



F ::= 10

G ::= FF

どちらの処理も
L を後ろにappendする形なので、
前に足すパス文字列だけ見る

めんどくさいケース: cons

- (cons ☆ △) に L でアクセス ➔
  - ☆ には {w | 0w ∈ L} でアクセス。
  - △ には {w | 1w ∈ L} でアクセス。

- L の前に何かをprependという規則になっていない
  - さっきのやり方で文法に変換するのが難しい

- (cons ☆ △) に L でアクセス ➔
  - ☆ には {$\underline{0}$w | w ∈ L} でアクセス。
  - △ には {$\underline{1}$w | w ∈ L} でアクセス。

$\underline{0}0$ -> ε

$\underline{1}1$ -> ε

パス文字列を"削る"操作を表す左逆元 $\underline{0}$ $\underline{1}$ を導入して無理矢理文脈自由文法として扱えるようにする

めんどくさいケース: if

- (if ☆ then △ else ◇) に L でアクセス ➔
  - **☆ には {ε} でアクセス**
  - △ には L でアクセス
  - ◇ には L でアクセス

- L の前に何かをprependという規則になっていない

- 関数定義の逆算のところでそもそも分けて考えることにする (返値に漏れるアクセスとそうでないアクセスを分けて考える)

めんどくさいケース: if

0 f(x) :=

..... if ..x.. then ... x ....

f(x) の返値に L でアクセス

➔ 引数xには、***返値経由では、***

∪ifの外のxの出現 Lから逆算したxへのアクセス

➔ 引数xには、***関数内では、***

∪ifの中のxの出現 Lから逆算したxへのアクセス

例

- append(x,y) :=

  if nil?(x) then y else cons(car(x), append(cdr(x),y))

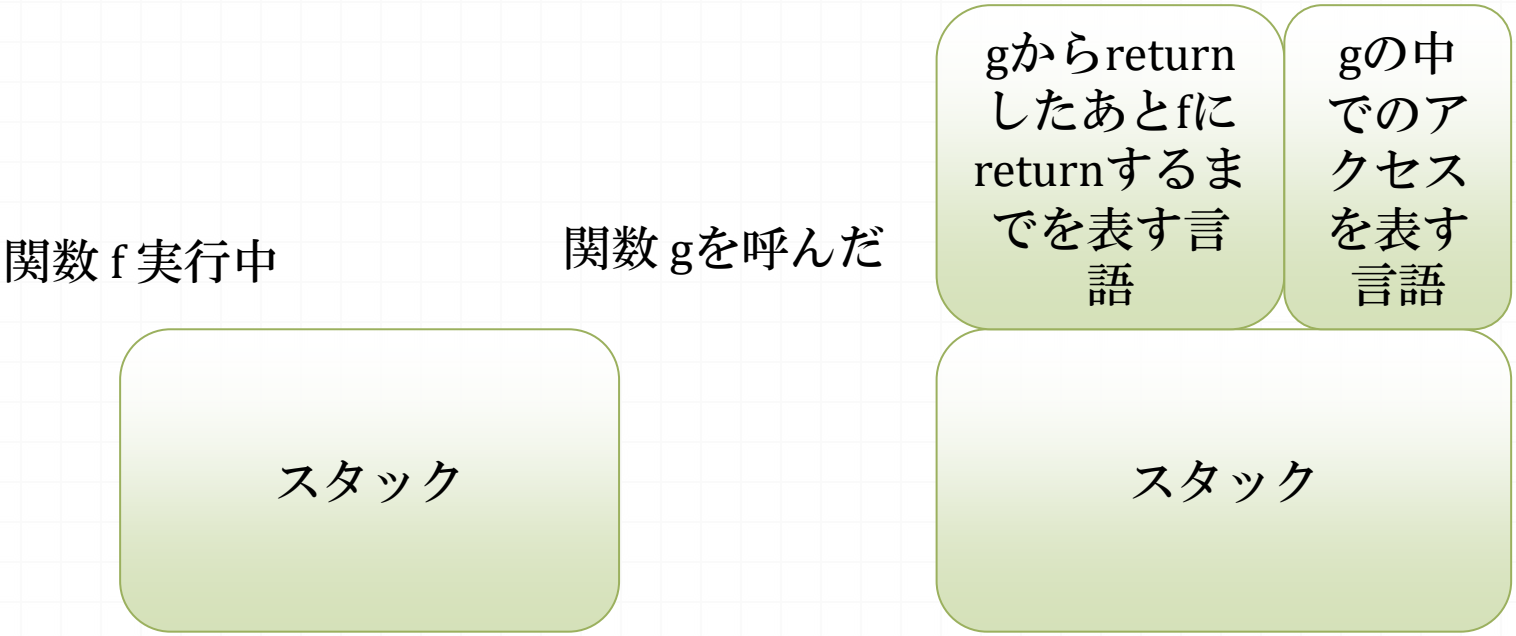
- [append_in_x] ::= ε
- [aapend_in_y] ::= ∅
- [append_out_x] ::= 0$\underline{0}$ | 1 [append_out_x] $\underline{1}$
- [append_out_y] ::= ε | [append_out_y] $\underline{1}$

GCの実行

- プログラムの各時点で、今後アクセスする可能性のあるパス集合を求められるようにしておく
- その集合の範囲外のノードを判定して回収しまくる

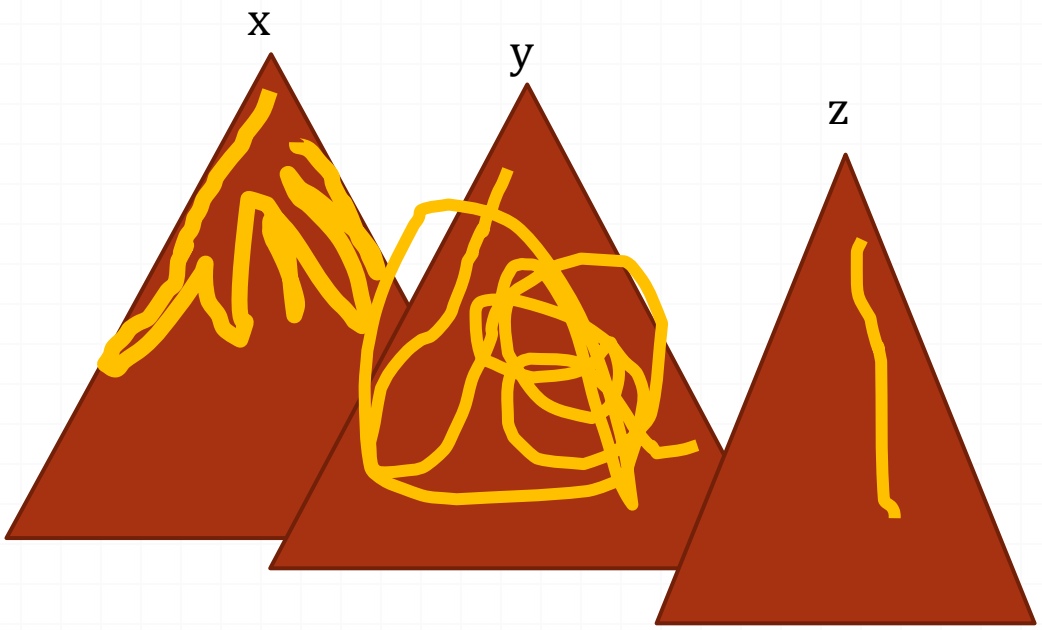
プログラムの各時点で、今後アクセスする可能性のあるパス集合を求められるようにしておく

- 関数を呼び出すたびに、スタックに「今後のアクセス」を表す言語（を表した文法）を積む
  - このスタックの concat と union で求まる



その集合の範囲外のノードを判定して回収しまくる

- さっきのスタックを使うと、各変数からの「今後使う可能性があるアクセスパス」がわかるので
  その範囲をマーキング



注意事項: 0̲0, 1̲1 の扱い

- 作った文法は 0̲ や 1̲ という便宜上の記号が入っていて直接パスに対応しないので、
- 論文では、
  - CFG を正規言語で近似。
  - NFA にして、0̲0 と 1̲1 による遷移にε辺を張りまくる
- としてこれらを消す近似をしていました

- Appendの例 “A ::= 0̲0 | 1 A 1̲” は “1*0̲01̲*” になる

注意事項: traverseしまくり

- 普通のGCは「到達可能な範囲」をマークするので、一度（他の変数経由などで）到達したブロックの先は全部到達可能なので、見る必要がない
- 提案手法のGCは、変数毎にアクセスパス集合が違うのでそれぞれから共有部分もたどらないといけない

議論しましょう

- CFG を正規言語で近似する必要は本当にあるのか？
  - （あるいは逆に、実際のメモリリークを考えると、もっとずっと弱い表現で近似してもよいという可能性はないか？）
- 複数回のtraverseが起きないように何かかっこいい実装できないか？
- その他、これを実用に耐えるものにしようと思ったらどうするといいか？