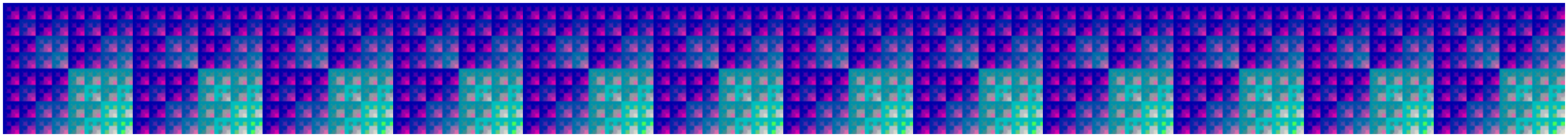
コンパイラ演習
第12回



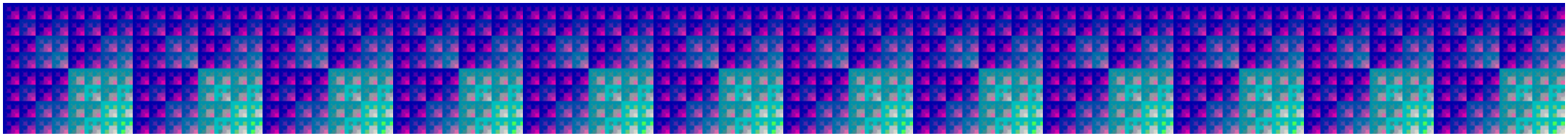
2006/1/26

大山 恵弘　　佐藤 秀明

今回の内容

- エスケープ解析
  - メモリに置かれる値のうち、ヒープではなくスタックにallocateできるものを発見
  - Garbage collectionの負荷を軽減
  - Java SE 6が採用
    - 2006年夏にリリース予定
- [参考]リージョン推論
  - 静的メモリ管理の一般的枠組み
  - 本講義ではML Kit[Tofte et al.]をもとに説明します
    - 全ての値をスタック(の一般形)に確保

背景



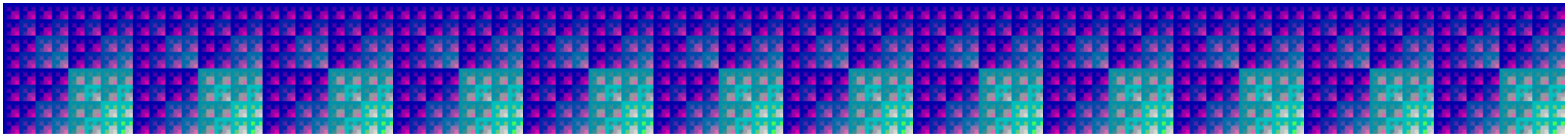
型システムの広範な応用

- プログラム解析に対する要求の高まり
  - プログラムの安全性を実行前に保証したい
    - 動的解析は面倒
  - 様々な静的解析を統一的に定式化したい
    - 理論的基盤の強化
- 解法: 型システムの採用
  - コンパイル時に型チェックを実行
    - 型チェックを通ったプログラムは必ず安全
  - 解析アルゴリズムを型付け規則で記述
    - アルゴリズムの実装/数学的証明が共通な枠組みの下で可能

型を用いた静的解析の例

- Dependent types
  - 静的な配列境界検査
    - http://cs-www.bu.edu/fac/hwxi/
- Resource usage analysis
  - 計算資源 (ファイル、ソケット等) が正しく使用されているかチェック
    - http://www.kb.ecei.tohoku.ac.jp/~koba/publications.html
- Information flow analysis
  - 機密情報が外部に漏れないことを保証
    - http://www.cs.cornell.edu/Info/People/jgm/lang-based-security/

エスケープ解析

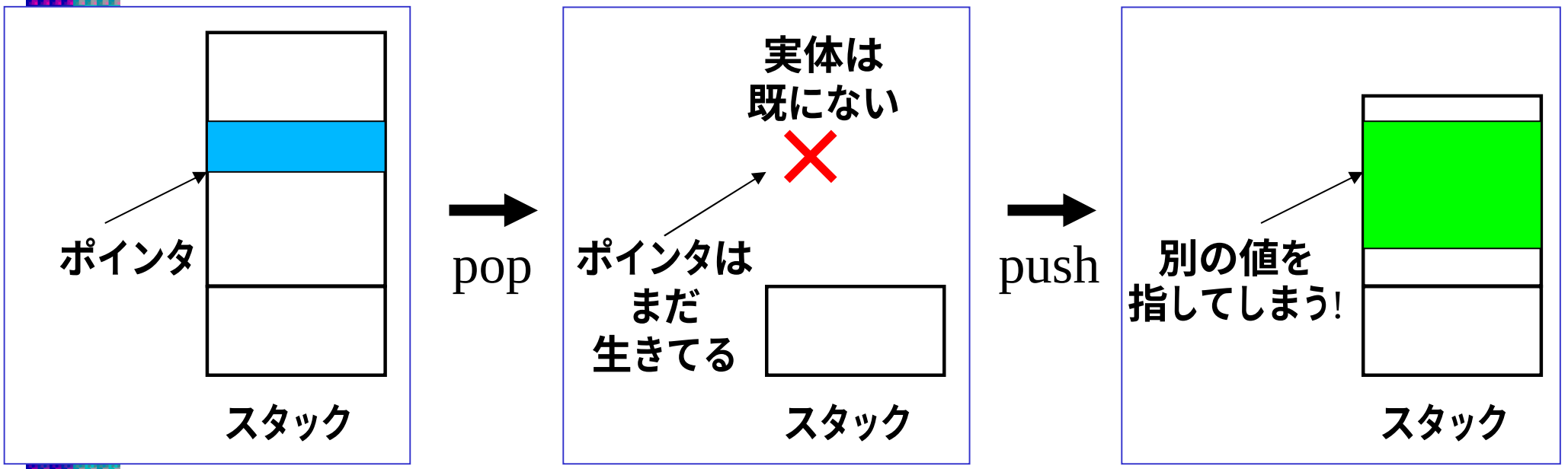


「エスケープ」するデータ構造

- 「関数からエスケープする」＝現在実行中の関数を抜けた後もアクセスできる
  - ポインタが返り値としてreturnされる
  - ポインタがグローバル変数に代入される
- エスケープするか否かを表すフラグ ρ を tuple/closure/arrayの型情報に追加
  - 決してエスケープしない→スタックに確保可能
  - エスケープするかも→ヒープに確保する必要性

エスケープ解析の正当性

- エスケープするものが誤ってスタックに置かれるとまずい
  - dangling pointerを用いたスタック領域破壊の危険性
- 有効な策: アルゴリズムを型付け規則で表現
  - 安全性を型システム上で証明
    - dangling pointerが発生しない　または
    - dangling pointerを用いたアクセスが起こらない



直感的な例(1/4)

```
let f x =
  let p = (3, 4) in  (* p : (int × int)false *)
  let (a, b) = p in
  a + b
in ...
```

- 組pは関数fから戻った後はアクセスできないので、型$(int \times int)_{false}$が与えられる

直感的な例(2/4)

```
let f y =
  let g x = x + y in  (* g : (int → int)true *)
  g
in ...
```

- 関数gのclosureは関数fから返されるので、型$(int \rightarrow int)_{true}$が与えられる

直感的な例(3/4)

```
let f y =
  let a = Array.create 5 0 in
                              (* a : (int array)true *)
  let g x = x + a.(2) in  (* g : (int → int)true *)
  g
in ...
```

- 関数gのclosureがエスケープするので、gの自由変数である配列aもエスケープする

直感的な例(4/4)

```
let t = ref (0, 0) in
let f () =
  let p = (1, 2) in  (* p : (int × int)true *)
  t := p
in ...
```

- 組pは関数fの内部から見てグローバルな領域に代入されるので、型$(int \times int)_{true}$となる

型解析の一般的な手順(1/2)

1. 対象言語の型を、必要な情報で拡張
   - エスケープするか否かを表すフラグなど
2. 型付け規則を書き下す
   - エスケープ情報の生成/伝搬ルールを定義
3. 目的のプログラムに対して型付けを実行
   - 型付け規則を「下から上に」当てはめる
   - この時点でフラグの値はまだ確定していない

型解析の一般的な手順(2/2)

4. 前項の型付けから制約を抽出
   - 拡張したフラグに関する論理式の集合
   - フラグの情報が伝搬される条件を表現
5. 反復法等によりフラグ間の制約を解消
   - 各フラグの値を確定させる

   i. とりあえずすべてのフラグをfalseとする

   ii. 制約に矛盾する部分からフラグをtrueに変更

   iii. 矛盾がなくなるまでii.を繰り返す

解析戦略の例(1/3)

- (副作用を考慮しなくてもわかる)制約の例
  - 関数の返り値はエスケープする
  - 関数closureがエスケープするなら
    関数の自由変数もエスケープする
  - tupleがエスケープするなら
    その各要素もエスケープする
  - 配列がエスケープするなら
    その各要素もエスケープする

解析戦略の例(2/3)

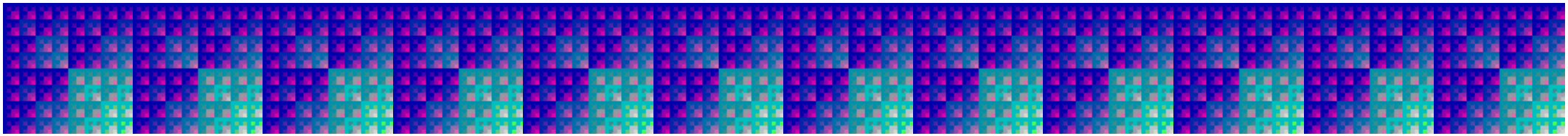
- 副作用を考慮し出すと途端に難しくなる
  - ポインタの解析は面倒
- 「グローバル変数へ代入」を
  「現在のフレームからポインタ漏洩」で近似
  - ローカルで生成した変数以外へのポインタ渡しはすべてエスケープとみなす

解析戦略の例(3/3)

- 配列の型にもう一つフラグGを追加
  - 現在のフレームよりもグローバルなレベルにある配列であることを表現
  - 関数のbodyの型付けは、bodyの実行開始時に参照可能な全変数のGフラグを立ててから行う
- Gの立った配列への代入はエスケープとみなす

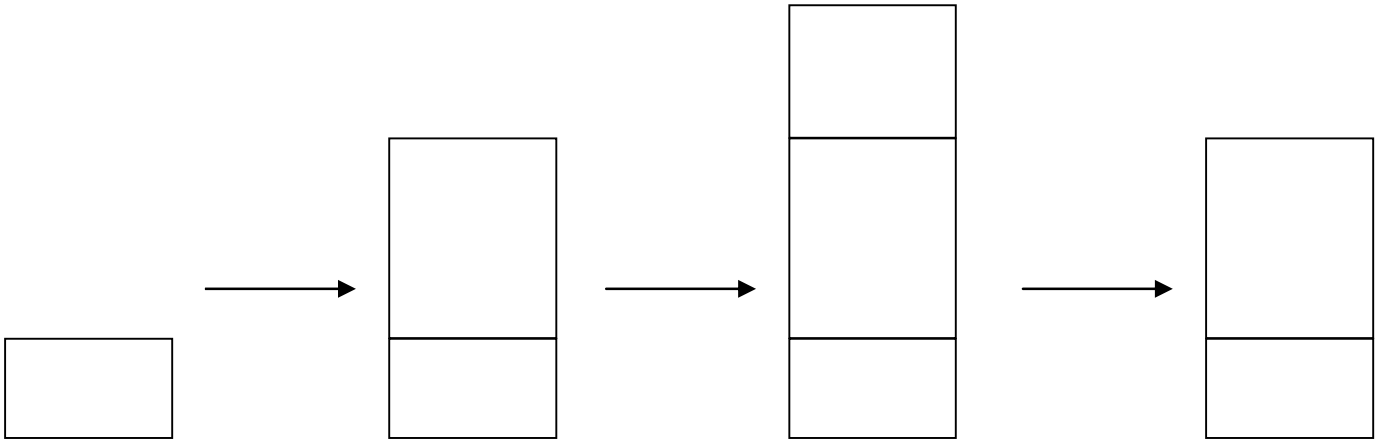
- 実際に式を解析する例は巻末資料を参照

リージョン推論



(ML Kitにおける)リージョンの概念

- 全メモリ空間をスタックのように管理
  - 全ての値をスタックに確保
- リージョン　スタックフレーム
  - 各リージョンのpush/popは
    関数のcall/returnと必ずしも同期しない



静的解析によるリージョン導入

- 各々のリージョンをいつpush/popすべきか
  - 各リージョンの存在期間を極限まで短くしたい
    - 使用する前にできるだけ遅くpush
    - 使用し終わったらできるだけ早くpop
  - 各リージョンをpush/popする順序は入れ子構造に制限される
    - 最初にpushされたリージョンは最後にpopされる
- 適当なpush/popのタイミングをコンパイラが静的解析により推論
  - メモリのallocate/deallocateを自動で管理

リージョンのための構文拡張

- letregion ρ in e end
  - 式eを評価する直前に新規リージョンρをpush
  - eの評価結果を得た直後にρをpop
- e@ρ
  - 式eの値をリージョンρ内に確保
  - eは値を生成する式に限られる
    - タプル生成: (x, y, z)@ρ
    - クロージャ生成: let rec f x @ρ = …
    - int生成: 1@ρ
    - などなど

Region Polymorphism

- 関数の返り値は関数の外側で定義されたリージョンに格納
  - 関数から返った後も使用されるから
- どのリージョンに返り値を格納するかは呼び出し元によって異なる
  - リージョンを(構文上の)引数として指定

let rec f [ρ] x = … in ... f [ρ ’] x’ …

定義　　適用

リージョン推論の例(fac)

```
let rec fac n =
  if n = 0 then 1
  else fac (n-1) * n in
fac 5
```



```
let rec fac [ρret] n =
   letregion ρbool in
      if  (letregion ρ0 in
             (n = (0@ρ0))@ρbool
           end)
       then 1@ρret
       else letregion ρarg, ρret' in
              (fac [ρret'] (letregion ρ1 in
                              (n-(1@ρ1))@ρarg
                            end)
                * n)@ρret
            end
   end in
letregion ρ5 in
   fac [ρans] (5@ρ5)
end
```

ただしρ ans は式全体の返り値を格納するリージョン(式の外側で定義済み)

リージョン推論の改良

- 効率的なリージョン操作
  - 複数の値を同一リージョンに確保するようにまとめる
  - 既存リージョンの内容をリセットして再利用
- リージョン概念の一般化
  - リージョンのサイズを可変にする
  - 任意のポイントでのリージョン確保/解放
    - push/popの順序は入れ子構造でなくともよい
    - ヒープに対するmalloc/freeの自動挿入に近い

共通課題(1/2)

- 次のプログラム中で生成される各tuple/arrayにエスケープフラグを付加せよ。
  - できる限り賢く解析せよ
    - 講義と巻末資料で紹介した戦略よりも賢くできるはず
    - dangling pointerを用いるアクセスを発生させないこと
    - 厳密なアルゴリズム/型付けは考慮せずともよい

```
let a = Array.create 1 (1, 2) in
let b = (3, 4) in
let rec g p = p in
let rec f () =
  let c = if (条件式) then a else Array.create 1 (g (5, 6)) in
  c.(0) <- b;
  c in
f ()
```

共通課題(2/2)

- リージョン推論を前出のfacの例のようにナイーブに行うと、末尾呼び出し最適化が困難になる場合がある。このことについて以下の各項目に答えよ。
  - 困難とは具体的に何か。
  - この問題を引き起こす本質的な原因はどこにあるか。
  - 解決するためにはシステムにどのような変更を加えればよいか。(自由に挙げよ)

コンパイラ係用選択課題

- エスケープ解析を実装せよ。
  - 副作用については保守的に実装してもよい
    - 例: arrayへの代入はすべてエスケープとみなす

課題の提出先と締め切り

- 提出先: compiler-enshu@yl.is.s.u-tokyo.ac.jp
- 共通課題の締め切り:
  2週間後 (2/9) の午後1時
- コンパイラ係用課題の締め切り:
  2006年3月31日
- Subject: report 12 <学籍番号> <アカウント>
- 本文にも氏名と学籍番号を明記のこと

課題の提出についての注意

- プログラムだけでなく、説明・考察・感想なども書くこと
- 基本的にはメールの本文に解答を記述
- 多くのソースを送る必要がある課題では、ソースをtarファイルなどに固めてメールに添付のこと

参考文献

- ML Kit情報
  - http://www.it-c.dk/research/mlkit/
- リージョン(+型) のちゃんとした教科書
  - Benjamin C. Pierce, editor. Advanced Topics in Types and Programming Languages, Chapter 3.
    - 理学部7号館3階の図書室にあります

コンパイラ係のコンパイラ部分の成績評価について(1)

- 各コンパイラ係と大山、佐藤の間で面談をします
  - 基本的にはレイトレ競技会の付近の日または当日
  - compiler-enshu@…にメールしてアポをとって下さい
  - 場所は地下端末室、時間は20分程度
  - やること:
    - 自作コンパイラの特徴・独創的な点などの説明
    - 自作コンパイラによる、プログラム(レイトレ含む)のコンパイル・実行のデモ
      - 自作コンパイラでコンパイルしたレイトレが自作CPUまたはシミュレータ上で動く様子を見せて下さい

コンパイラ係のコンパイラ部分の成績評価について(2)

- 選択課題を必ず一つ以上提出して下さい
  - GC、オブジェクト指向、多相型、例外、パターンマッチ、エスケープ解析
  - これらと同等以上の難度を有する言語機構の実装をもって選択課題の提出とみなすことは可能です
- 選択課題の提出〆切：3月31日

コンパイラでない係のコンパイラ部分の成績評価について

- 共通課題の提出状況と提出内容をもとに評価します