プログラム言語論-第 4 週-

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類 講義, 2009 年 5 月 11 日

1. 意味論からインタープリタへ
2. スタックを用いたインタープリタ
3. 関数型プログラム言語

前回と今回の概要

手続き型言語 (miniC)

- (前回) ブロック構造，変数のスコープ，静的束縛と動的束縛 etc.
- (前回) miniC 言語インタープリタを用いた演習.
- (今日の前半) 意味論からインタープリタの仕組みへ

関数型言語 (miniML)

- (今日の後半) 関数型言語の基本; 単一代入, 高階関数.
- (今日の後半) 関数型言語の意味論とインタープリタ, 関数クロージャ.
- (次回以降) 制御構造, miniML を用いた演習.
- (次回以降) 型 (型検査，型推論，抽象データ型)

microC の意味論 (第 2 週の復習)

microC ... 第 2 週の時点の miniC 言語 (関数呼出しなし, ブロック構造なし)
$\sigma(e)$: 状態 σ における式 e の計算結果 (値).

- e が変数 x のとき: $\sigma(e)$ は，σ における x の値とする.
- e が定数 c のとき: $\sigma(e) = c$.
- e が $(e_1 + e_2)$ のとき: $\sigma(e) = \sigma(e_1) + \sigma(e_2)$. ただし，$\sigma(e_i)$ の一方が自然数でないとき，$\sigma(e)$ は値を持たない.
- ...

例: $\sigma = (x = 1, y = 2)$ のとき，$\sigma((x + y) + 1) = 4$.

文の意味

$\sigma[\![s]\!]$: 状態 σ において文 s を実行した後の状態.

- s が代入文 $(x = e)$ のとき： $\sigma[\![s]\!] = \sigma[x \to \sigma(e)]$.
- s が $(\mathit{if}\ e\ s_1\ \mathit{else}\ s_2)$ のとき:
  - $\sigma(e) = \texttt{true}$ のとき： $\sigma[\![s]\!] = \sigma[\![s_1]\!]$
  - $\sigma(e) = \texttt{false}$ のとき： $\sigma[\![s]\!] = \sigma[\![s_2]\!]$
  - ...
- ...

miniC の意味論

microC に比べて：

- 関数呼び出しなどのブロック構造がある.
- 環境 (状態) に含まれる変数が，実行時に変化する．(静的束縛, 動的束縛)
- miniC の意味論を理解するためには，環境の変化を記述する仕組みが必要.

今日は，フォーマルな意味論ではなく，環境を表すスタックを用いた素朴なインタープリタに基づいた意味論を，miniC に対して展開する.
Reference implementation: 実行効率は良くないが，正確な意味論を知るための (処理系の) 実装.

プログラム実行時のメモリ

(絵は板書する)

- Register (CPU のレジスタ)
- Program Counter (コード領域を指す変数)
- Code (プログラムのコードを格納する領域)
- Environment Pointer (スタックを指す変数)
- Data:
  - Stack (スタック)
  - Heap (ヒープ)

Register は，以下では無視する．(reference implementation では処理性能を考慮しない)　Heap は後述.

プログラムスタック (あるいは，環境スタック)

- ブロック構造を持つプログラム言語の処理系で使用.
- ブロックに局所的な変数たちの値を格納.

```
int f (int y) {                  ----
 int z = 0;                      z=10
 return y+z;}                    y=11
main () {                 ----   ----   ----
  int x = 10;             x=10   x=10   x=10
  x = f(x+1);       ---   ----   ----   ----
}
```

スタックフレーム

スタックフレーム (stack frame, activation record)

- スタックに積まれるデータたちのひとまとまり.
- 1 つのブロックは 1 つのスタックフレーム
- スタック全体は，いくつかのスタックフレームから構成される.
- スタックフレームの中身は？ (関数に対応するブロックの場合)
  - 局所変数 (関数の引数，関数で定義された変数) の値
  - 計算の途中結果
  - 関数の戻り先アドレス (コード領域の番地)
  - 関数が返す値
  - 1 つ前のスタックフレームへのポインタ (Control link)
  - 値を参照する変数を探すためのリンク (Access link)

問題: なぜ，Control link のほかに Access link が必要か?

関数呼出しの意味論

`int f(int x, bool y) {int z; ...}`
$f(e_1, e_2)$ の処理の終了; `return e;` が実行されたとき:

- e を計算し，その値をスタックフレーム内の返り値を格納するスペースにいれる.
- スタックフレームに保存しておいた戻りアドレスに飛ぶ. (Program Counter にそのアドレスをいれる.)
- 現在のスタックフレームをはずす. (Control link をたどり，Environment Pointer が 1 つ前のスタックフレームを指すようにする.)
- (局所変数はすべて失われる.)

演習課題

演習用の miniC 言語 (version 3 以降) には，show という隠しコマンドがあり，その時点での，Active なスタックフレームたちを表示することができる.

```
int f (int x) {
  ...
  show (0);
  ...
}
```

その瞬間に Access 可能な変数の値を，スタックフレームごとに表示する. (スタック全体ではなく，Access link でつながっているもののみ.)
この機能を使うと，プログラムスタックが増減する様子や，静的束縛と動的束縛の差が，より明確にわかるので，試してみなさい.

関数型言語

C 言語などの手続き型 (あるいは命令型) 言語と比較したときの関数型言語の特徴:

- 数学的な「関数」概念が基本にある. (が，数学関数そのものではない.)
- 単一代入が基本. 参照透明性 (referential transparency)
- 意味論が明快，簡潔
- 従って，インタープリタも明快，簡潔
- 簡単な割に実は強力; 高階関数，データ型
- 得意な分野: 種々のアルゴリズムの記述，プログラム言語処理系，記号処理システム (不定長データの複雑な処理)
- 不得意な分野: 数値計算 (固定長データの高速計算)

プログラム言語研究では,「関数型言語＋命令型への拡張」の形で種々のプログラム言語をとらえると，都合がよいことが多い.

関数呼出しの意味論

`int f(int x, bool y) {int z; ...}`
環境 σ のもとでの $f(e_1, e_2)$ の処理：

- 引数 e_1, e_2 を現在の環境 σ で計算する.
- それらの結果を v_1, v_2 とする.
- 環境スタックに新しいスタックフレームを追加する.
- Environment Pointer が新しいスタックフレームを指すようにする.
- 新しいスタックフレームに以下の値を格納:
  - Control link: 1 つ前のスタックフレームへのポインタ.
  - Access link: 値を参照する変数を探すためのリンク.
  - 戻り先アドレス: 関数の計算終了後に戻ってくるべきコード領域の番地.
  - 戻り値を格納するスペース.
  - 関数の実引数 v_1, v_2
  - 関数の局所変数 z を格納するスペース

Access link について

Control link: 1 つ手前のスタックフレームへのポインタ.
すなわち，現在呼ばれている関数呼出しの 1 つ前に呼ばれた関数呼出し，あるいは，現在の関数呼出しを呼んだ場所を指す. 実行時の関数呼出し順序の逆順.

- 動的束縛: 変数の値を知りたい (参照する) とき，Control link を順番にたどっていけばよい.
- 静的束縛: 変数の値を知りたいとき，Control link では役に立たない.
  「現在のブロックの 1 つ外にあるブロック」(プログラムの文面上での位置関係) を知りたい.
  これは,「現在のスタックフレームの 1 つ前のスタックフレーム」とは必ずしも一致しない.
- Access link: 1 つ外のブロックに対応するスタックフレームを指す.

まとめ

学習したこと

- ブロック構造をもつプログラム言語の意味論
- スタックを用いた Reference Implementation

関数型言語

- Lisp: 古くからある関数型言語，人工知能システムや数式処理システムなど.
- Scheme: Lisp の意味論を洗練したもの.
- ML (meta language): 関数型言語のグループの名前，SML, OCaml などがある. 最も成功した関数型言語.
- ほかには，ErLang (商業的に使用), F#(MicroSoft の ML-like な言語), Haskell (研究者が作った言語) など.

miniML は OCaml のサブセットとして設計. Scheme や Haskell などとは構文は異なるが，それらのサブセットと思うことも可能.

miniML の構文と意味

資料を参照のこと.

miniML のプログラム例

```
(let x = 1 in x + x) + 5
let f = (fun x -> x + 1) in f (f 3)
(fun f -> f (f 3)) (fun x -> x + 1)
(fun f -> fun x -> f (f x)) (fun y -> y + 1) 3
let x=5 in let f=(fun y->x+y) in let x=10 in (f 20)
```


miniML の特徴

- 式しかない (文がない).
- 変数への値の代入 (assignment) がない.
- 1 引数関数しかない (対を使って，多引数関数を模倣できる)
- 型を書かない.
- 関数もデータ: 関数の引数や計算結果が関数でもよい (高階関数)
- ブロック構造, 静的束縛: クロージャを使った実装.

関数クロージャ (closure)

(5/11 の講義では，クロージャについては話せなかったので，次回にやります．)
関数クロージャ(fun x -> e, σ)

- 関数と環境のセット (対).
- 関数を得たとき (式を計算して，その結果が関数になったとき) の環境を保存している.
- 静的束縛を実現することができる．(Access link のかわり)

なお，動的束縛のときは，クロージャを作る必要はない.

値呼びと名前呼び

関数呼出し e_1 e_2 における実行順序の違い:

- 値呼び
  - まず，e_1 と e_2 を計算して，値 v_1 と v_2 を得る．(ここで，e_1 が先かどうかで 2 通りの順序がある．)
  - v_1 が関数でないとき，エラー.
  - v_1 が関数クロージャ(fun x -> e_0, σ') のとき，環境 σ' に $x = v_2$ を追加.
  - その環境で e_0 を計算して，その結果を全体の答えとする.
- 名前呼び
  - e_1 を計算して，値 v_1 を得る.
  - v_1 が関数でないとき，エラー.
  - v_1 が関数クロージャー (fun x -> e_0, σ') のとき，環境 σ' に $x = e_2$ を追加.
  - その環境で e_0 を計算して，その結果を全体の答えとする.

問題: 2 つの方法により，計算結果が異なることがあるだろうか？また，どちらの方法が，より効率的だろうか？

まとめ

- 関数型プログラム言語の導入.
- プログラム言語における種々の概念を説明するのに適切.
- たとえば，値呼びと名前呼び，高階関数.
- 積み残し: ヒープ，末尾再帰，制御構造，データ型 (データ構造).

short quiz (3限用)

名前呼びと値呼びを比較しなさい (実行性能その他，思いつく観点で).
もし思いつくものがなければ，自分の知っているプログラム言語が名前呼びと値呼びのどちらであるか，なるべくたくさんの言語について答えなさい.