

コンパイラ

Jacques Garrigue, 2022 年 1 月 12 日

1 電卓とコンパイラ

```
Require Import ZArith Extraction.  
From mathcomp Require Import all_ssreflect.
```

```
(* Simple Calculator *)
```

```
Module Simple.
```

```
Inductive expr : Set :=  
  | Cst of Z  
  | Var of nat  
  | Add of expr & expr  
  | Min of expr  
  | Mul of expr & expr. (* 変数を含む整数式 *)
```

```
Fixpoint eval (env : list Z) (e : expr) : Z := (* 評価関数 *)  
  match e with  
  | Cst x => x  
  | Var n => nth 0 env n (* 変数の値は env で与えられる *)  
  | Add e1 e2 => eval env e1 + eval env e2  
  | Min e1 => 0 - eval env e1  
  | Mul e1 e2 => eval env e1 * eval env e2  
  end%Z.
```

```
Inductive code : Set := (* 逆ポーランド記法による計算譜 *)  
  | Cimm of Z  
  | Cget of nat  
  | Cadd  
  | Cmin  
  | Cmul.
```

```
(* 各コマンドがスタックを変える *)
```

```
Definition step (stack : list Z) c :=  
  match c, stack with  
  | Cimm x, _ => x :: stack  
  | Cget n, _ => nth 0 stack n :: stack  
  | Cadd, x :: y :: st => x+y :: st  
  | Cmin, x :: st => 0-x :: st  
  | Cmul, x :: y :: st => x*y :: st  
  | _, _ => nil  
  end%Z.
```

```
(* foldl (+) [:: x1; ..; xn] x0 = ((x0 (+) x1) (+) .. (+)) xn *)
```

```
Definition eval_code := foldl step.
```

```
Fixpoint compile d (e : expr) : list code :=  
  match e with  
  | Cst x => [:: Cimm x]
```

```

| Var n => [:: Cget (d+n)]
| Add e1 e2 => compile d e2 ++ compile (S d) e1 ++ [:: Cadd]
| Min e1 => compile d e1 ++ [:: Cmin]
| Mul e1 e2 => compile d e2 ++ compile (S d) e1 ++ [:: Cmul]
end.

```

Theorem compile_correct e d stack : (* コンパイラの正しさ *)
 eval_code stack (compile d e) = eval (drop d stack) e :: stack.

Proof.

```

rewrite /eval_code.
elim: e d stack => //=[n|e1 IHe1 e2 IHe2|e IHe|e1 IHe1 e2 IHe2] d stack.
- by rewrite nth_drop.
- by rewrite foldl_cat IHe2 foldl_cat IHe1.

```

Admitted.

End Simple.

(* Iterating calculator *)

Module Iterator.

Inductive expr : Set :=

```

| Cst of Z
| Var of nat
| Add of expr & expr
| Min of expr
| Mul of expr & expr.

```

Fixpoint eval (env : list Z) (e : expr) : Z :=

```

match e with
| Cst x => x
| Var n => nth 0 env n
| Add e1 e2 => eval env e1 + eval env e2
| Min e1 => 0 - eval env e1
| Mul e1 e2 => eval env e1 * eval env e2
end%Z.

```

Inductive cmd : Set :=

```

| Assign of nat & expr (* env[n] に結果を入れる *)
| Seq of cmd & cmd (* 順番に実行 *)
| Repeat of expr & cmd. (* n 回繰り返す *)

```

Fixpoint eval_cmd (env : list Z) (c : cmd) : list Z :=

```

match c with
| Assign n e => set_nth 0%Z env n (eval env e)
| Seq c1 c2 => eval_cmd (eval_cmd env c1) c2
| Repeat e c =>
  if eval env e is Zpos n (* seq の iter を使う *)
  then iter (Pos.to_nat n) (fun e => eval_cmd e c) env
  else env
end.

```

Inductive code : Set :=

```

| Cimm of Z
| Cget of nat

```

```

| Cadd
| Cmin
| Cmul
| Cset of nat (* スタックの上を n 番目に書き込む *)
| Crep of nat (* 次の n 個の命令ををスタックの上分繰り返す *)
| Cnext. (* 終わったら Cnext の後ろに跳ぶ *)

```

```

Fixpoint eval_code (stack : list Z) (l : list code) :=
  match l with
  | nil => stack
  | c :: l' =>
    let stack' :=
      match c, stack with
      | Cimm x, _ => x :: stack
      | Cget n, _ => nth 0 stack n :: stack
      | Cadd, x :: y :: st => x+y :: st
      | Cmin, x :: st => 0-x :: st
      | Cmul, x :: y :: st => x*y :: st
      | Cset n, x :: st => set_nth 0%Z st n x
      | Crep _, Zpos n :: st =>
        iter (Pos.to_nat n) (fun st => eval_code st l') st
      | Crep _, _ :: st => st
      | Cnext, _ => stack
      | _, _ => nil
      end%Z
    in
    match c with
    | Crep n => eval_drop n stack' l' (* Crep の後はコードを飛ばす *)
    | Cnext => stack' (* Cnext は評価を止める *)
    | _ => eval_code stack' l' (* 他の場合は続ける *)
    end
  end
with eval_drop n st (l : list code) := (* 相互再帰 *)
  match l, n with
  | _ :: l', 0 => eval_code st l'
  | _ :: l', S n' => eval_drop n' st l'
  | [::], _ => st
  end.

```

```

Fixpoint compile d (e : expr) : list code :=
  match e with
  | Cst x => [:: Cimm x]
  | Var n => [:: Cget (d+n)]
  | Add e1 e2 => compile d e2 ++ compile (S d) e1 ++ [:: Cadd]
  | Min e1 => compile d e1 ++ [:: Cmin]
  | Mul e1 e2 => compile d e2 ++ compile (S d) e1 ++ [:: Cmul]
  end.

```

```

Fixpoint compile_cmd (c : cmd) : list code :=
  match c with
  | Assign n e => compile 0 e ++ [:: Cset n]
  | Seq c1 c2 => compile_cmd c1 ++ compile_cmd c2
  | Repeat e c =>

```

```

    let l := compile_cmd c in
    compile 0 e ++ [:: Crep (length l)] ++ l ++ [:: Cnext]
end.

```

```

Definition neutral c :=
  match c with Cnext | Crep _ => false | _ => true end.

```

```

Inductive balanced : list code -> Prop := (* Crep と Cnext の対応が取れている *)
  | Bneutral : forall c, neutral c = true -> balanced [:: c]
  | Bcat : forall l1 l2, balanced l1 -> balanced l2 -> balanced (l1 ++ l2)
  | Brep : forall l, balanced l ->
    balanced (Crep (length l) :: l ++ [:: Cnext]).

```

```
Hint Constructors balanced.
```

```

Lemma eval_drop_cat st l1 l2 :
  eval_drop (length l1) st (l1 ++ Cnext :: l2) = eval_code st l2.
Admitted.

```

```
Check eq_iter. (* 証明に使える *)
```

```

Lemma eval_code_cat stack (l1 l2 : seq code) :
  balanced l1 ->
  eval_code stack (l1 ++ l2) =
  eval_code (eval_code stack l1) l2.
Admitted.

```

```

Lemma compile_balanced n e : balanced (compile n e).
Proof. by elim: e n => /=: auto. Qed.

```

```

Theorem compile_correct e d stack :
  eval_code stack (compile d e) = eval (drop d stack) e :: stack.
Admitted.

```

```

Lemma compile_cmd_balanced c : balanced (compile_cmd c).
Proof. by elim: c => /=: auto using compile_balanced. Qed.

```

```
Hint Resolve compile_balanced compile_cmd_balanced.
```

```

Theorem compile_cmd_correct c stack :
  eval_code stack (compile_cmd c) = eval_cmd stack c.
Admitted.
End Iterator.

```

```
Extraction Iterator.eval_code.
```

練習問題 1.1 1. 上記の証明の admit と Admitted をなくせ.

2. 引数が 0 より大きければコマンドを一回だけ行うコマンド `If of expr & cmd` を定義せよ. コンパイルの仕方も `Repeat` とほぼ同じ. それを使って整数の商を定義せよ.