コンパイラ

Jacques Garrigue, 2017 年 1 月 10 日

1　電卓とコンパイラ

```
Require Import ZArith.
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.

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 code : Set :=
  | Cimm of Z
  | Cget of nat
  | Cadd
  | Cmin
  | Cmul.

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
        | _, _ => nil
        end%Z
    in eval_code stack' l'
  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.

Lemma eval_code_cat stack l1 l2 :
  eval_code stack (l1 ++ l2) = eval_code (eval_code stack l1) l2.
Proof. by elim: l1 stack => //=. Qed.

Theorem compile_correct e d stack :
  eval_code stack (compile d e) = eval (drop d stack) e :: stack.
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 repeat_n A : Type (n : nat) (f : A -> A) (x : A) :=
  if n is S n' then repeat_n n' f (f x) else x.

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
```

```
    then repeat_n (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
  | Crep
  | Cnext.

Definition next_skip c n :=
  match c with
  | Crep => S n
  | Cnext => n - 1
  | _ => n
  end.

Fixpoint eval_code (skip : nat) (stack : list Z) (l : list code) :=
  match l with
  | nil => stack
  | c :: l' =>
    if skip is S _ then eval_code (next_skip c skip) stack l' else
    if c is Cnext then stack else
    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 =>
          repeat_n (Pos.to_nat n) (fun st => eval_code 0 st l') st
        | Crep, _ :: st => st
        | _, _ => nil
        end%Z
    in
    eval_code (next_skip c skip) stack' l'
  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 => compile 0 e ++ [:: Crep] ++ compile_cmd c ++ [:: Cnext]
  end.

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

Inductive balanced : list code -> Prop :=
  | 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 :: l ++ [:: Cnext]).

Hint Constructors balanced.

Lemma repeat_n_eq A:Type n f1 f2  (st : A) :
  (forall st, f1 st = f2 st) -> repeat_n n f1 st = repeat_n n f2 st.
Admitted.

Lemma eval_code_cat k stack (l1 l2 : seq code) :
  balanced l1 ->
  eval_code k stack (l1 ++ l2) =
  eval_code k (eval_code k 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 0 stack (compile d e) = eval (drop d stack) e :: stack.
Admitted.

Lemma compile_cmd_balanced c : balanced (compile_cmd c).
Admitted.

Lemma compile_cmd_S n c stack :
  balanced c -> eval_code (S n) stack c = stack.
Admitted.

Hint Resolve compile_balanced compile_cmd_balanced.

Theorem compile_cmd_correct c stack :
  eval_code 0 stack (compile_cmd c) = eval_cmd stack c.
Admitted.

End Iterator.
```