# A Coq formalization of an analysis and optimization of While

Fedyukovich Grigory
Andu, 5 of February, 2010

# Introduction

- We present the formalization of the While language developed in Coq.
- Then we develop the type systems for live variable analysis and dead code elimination and checked them in Coq.
- We conducted the proofs constructively: prove the cases for the axioms as induction basis and use induction hypothesis for the compositional rules.

# While language

a ::= x | n | a0 + a1 | a0 * a1 | a0 - a1 |

b ::= a0 = a1 | tt | ff | ¬b |

s ::= x := a | skip | S0; S1 | if b then S0 else S1 |
while b do S.

# Arithmetic expressions

```
Inductive AExp : Set :=
    | num : Num -> AExp
    | var : Var -> AExp
    | add : AExp -> AExp -> AExp
    …

Fixpoint ASemFun (a : AExp) (s : State) {struct a} : nat :=
    match a with
        | num n => NumSemFun n
        | var v1 => s v1
        | add a1 a2 => (ASemFun a1 s) + (ASemFun a2 s)
        …
end.
```

# Boolean expressions

```
Inductive BExp : Set :=
    | ttrue : BExp
    | ffalse : BExp
    | aeq : AExp -> AExp -> BExp
    | neg : BExp -> BExp.

Fixpoint BSemFun (b : BExp) (s : State) {struct b} : bool :=
    match b with
        | ttrue => true
        | ffalse => false
        | aeq a1 a2 => beq_nat (ASemFun a1 s) (ASemFun a2 s)
        | neg b => negb (BSemFun b s)
    end.
```

# Statements

```
Inductive Stm : Set :=
  | ass : Var -> AExp -> Stm
  | skip : Stm
  | comp : Stm -> Stm -> Stm
  | ifcond : BExp -> Stm -> Stm -> Stm
  | whileloop : BExp -> Stm -> Stm.
```

# Natural (big-step) semantics

```
Inductive eval : Stm -> State -> State -> Prop :=
  | eval_ass : forall s x a,
      eval (ass x a) s (update x a s)
      …
  | eval_comp : forall s1 s2 s3 S1 S2,
      eval S1 s1 s2 ->
      eval S2 s2 s3 ->
      eval (comp S1 S2) s1 s3.
```

# Natural (big-step) semantics

```
Inductive eval : Stm -> State -> State -> Prop :=
  | eval_while_tt : forall s1 s2 s3 b S,
      true = (BSemFun b s1) ->
      eval S s1 s2 ->
      eval (whileloop b S) s2 s3 ->
      eval (whileloop b S) s1 s3
  | eval_while_ff : forall s b S,
      false = (BSemFun b s) ->
      eval (whileloop b S) s s.
```

# Hoare logic

```
Definition Cond := State -> Prop.    (*Conditions (Pre- and Post-)*)

                    (*     Hoare Triple     *)
Inductive infer_triple : Cond -> Stm -> Cond -> Prop :=
    | infer_triple_ass : forall P x a,
        infer_triple (condAss P x a) (x <- a) P
        …
    | infer_triple_comp : forall P Q R S1 S2,
        infer_triple P S1 Q ->
        infer_triple Q S2 R ->
        infer_triple P (comp S1 S2) R
```

# Hoare logic

```
Inductive infer_triple : Cond -> Stm -> Cond -> Prop :=
  …
  | infer_triple_while : forall P b S,
      infer_triple (boolConj b P) S P ->
      infer_triple P (whileloop b S) (boolConj (neg b) P)
  …
  (*P – loop invariant*)
```

# Soundness and completeness

```
Definition valid_triple (P : Cond) (S : Stm) (Q : Cond) :=
  forall s s', P s /\ (eval S s s') -> Q s'.

Theorem HoareLogicSoundness :
  forall (P Q : Cond) (S : Stm),
  infer_triple P S Q -> valid_triple P S Q.

Theorem HoareLogicCompleteness :
  forall (P : Cond) (S : Stm) (Q : Cond),
  valid_triple P S Q -> infer_triple P S Q.
```

# Live variables analysis

```
Definition Live := Var -> bool.

Definition add (v : Var) (live : Live) : Live :=
  fun (v' : Var) =>
    if beq_nat v v' then true else live v'.

Definition subset (live0 : Live) (live1 : Live) : Prop :=
  forall (v : Var),
    live0 v = true -> live1 v = true.
```

# Live variables analysis

```
Inductive type_live_aexp : AExp -> Live -> Live -> Prop :=
    | type_live_var : forall live x0,
        type_live_aexp (var x0) (add x0 live) live
    …

Inductive type_live_stm : Stm -> Live -> Live -> Prop
    | type_live_ass_in_live : forall live live' x a0,
        member x live' = true ->
        type_live_aexp a0 live (remove x live') ->
        type_live_stm (ass x a0) live live'

    | type_live_ass_not_live : forall live x a0,
        member x live = false ->
        type_live_stm (ass x a0) live live
    …
```

# Dead code elimination

```
Inductive type_dead_stm : Stm -> Live -> Live -> Stm -> Prop :=

  | type_dead_ass_in_live : forall live live' x a0,
      member x live' = true ->
      type_live_aexp a0 live (remove x live') ->
      type_dead_stm (ass x a0) live live' (ass x a0)

  | type_dead_ass_not_live : forall live x a0,
      member x live = false ->
      type_dead_stm (ass x a0) live live (skip)
      …
```

# Dead code elimination

```
Definition related (s0 s1 : State) (live : Live) :=
        forall (v : Var), member v live = true -> s0 v = s1 v.

Theorem DCE_Soundness_a :
    forall (S : Stm) (live live' : Live) (S' : Stm),
        type_dead_stm S live live' S' ->
                forall s0 s1, related s0 s1 live ->
                forall s0', eval S s0 s0' ->
        exists s1', related s0' s1' live' /\ eval S' s1 s1'.
Theorem DCE_Soundness_b :
    forall (S : Stm) (live live' : Live) (S' : Stm),
        type_dead_stm S live live' S' ->
                forall s0 s1, related s0 s1 live ->
                forall s1', eval S' s1 s1' ->
        exists s0', related s0' s1' live' /\ eval S s0 s0'.
```

# Conclusion

| | | |
|---|---|---|
| **DCE** | **…** | **…** |
| **LVA** | **…** | **…** |
| **Hoare logic** | | |
| **Operational semantics** | | |
| **Syntax** | | |

# Conclusion

- The modularity consists of formalization of syntax as first abstract level, operational semantics as second level and Hoare logic as third level.
- Live variable analysis as the next level. Its type system is sound in the big-step semantics and Hoare logic.
- Basing on the fourth-level type system we developed another one for the dead code elimination.
- We leave the constructed environment for future development of an optimization tools. For instance, the common subexpression elimination based on the available expressions analysis will be added soon.

# Thank you!

???