IMP — A WHILE-language and two semantics

Heiko Loetzbeyer and Robert Sandner

September 11, 2023

Abstract

The formalization of the denotational and operational semantics of a simple while-language together with an equivalence proof between the two semantics. The whole development essentially formalizes/transcribes chapters 2 and 5 of [1]. A much extended version of this development is found in HOL/IMP of the Isabelle distribution.

Contents

1 Arithmetic expressions, boolean expressions, commands

theory *Com* **imports** *ZF* **begin**

1.1 Arithmetic expressions

consts
  loc :: i
  aexp :: i

datatype ⊆ *"univ(loc ∪ (nat -> nat) ∪ ((nat × nat) -> nat))"*
  aexp = N ("n ∈ nat")

```
      | X ("x ∈ loc")
      | Op1 ("f ∈ nat -> nat", "a ∈ aexp")
      | Op2 ("f ∈ (nat × nat) -> nat", "a0 ∈ aexp", "a1 ∈ aexp")


consts evala :: i

abbreviation
  evala_syntax :: "[i, i] ⇒ o"    (infixl <-a->> 50)
  where "p -a-> n ≡ ⟨p,n⟩ ∈ evala"

inductive
  domains "evala" ⊆ "(aexp × (loc -> nat)) × nat"
  intros
    N:   "⟦n ∈ nat;  sigma ∈ loc->nat⟧ ⟹ <N(n),sigma> -a-> n"
    X:   "⟦x ∈ loc;  sigma ∈ loc->nat⟧ ⟹ <X(x),sigma> -a-> sigma`x"
    Op1: "⟦⟨e,sigma⟩ -a-> n; f ∈ nat -> nat⟧ ⟹ <Op1(f,e),sigma> -a-> f`n"
    Op2: "⟦⟨e0,sigma⟩ -a-> n0;  ⟨e1,sigma⟩  -a-> n1; f ∈ (nat×nat) -> nat⟧
           ⟹ <Op2(f,e0,e1),sigma> -a-> f`⟨n0,n1⟩"
  type_intros aexp.intros apply_funtype
```

1.2 Boolean expressions

```
consts bexp :: i

datatype ⊆ "univ(aexp ∪ ((nat × nat)->bool))"
  bexp = true
       | false
       | ROp  ("f ∈ (nat × nat)->bool", "a0 ∈ aexp", "a1 ∈ aexp")
       | noti ("b ∈ bexp")
       | andi ("b0 ∈ bexp", "b1 ∈ bexp")      (infixl <andi> 60)
       | ori  ("b0 ∈ bexp", "b1 ∈ bexp")      (infixl <ori> 60)


consts evalb :: i

abbreviation
  evalb_syntax :: "[i,i] ⇒ o"    (infixl <-b->> 50)
  where "p -b-> b ≡ ⟨p,b⟩ ∈ evalb"

inductive
  domains "evalb" ⊆ "(bexp × (loc -> nat)) × bool"
  intros
    true:  "⟦sigma ∈ loc -> nat⟧ ⟹ ⟨true,sigma⟩ -b-> 1"
    false: "⟦sigma ∈ loc -> nat⟧ ⟹ ⟨false,sigma⟩ -b-> 0"
    ROp:   "⟦⟨a0,sigma⟩ -a-> n0; ⟨a1,sigma⟩ -a-> n1; f ∈ (nat*nat)->bool⟧
            ⟹ <ROp(f,a0,a1),sigma> -b-> f`⟨n0,n1⟩ "
    noti:  "⟦⟨b,sigma⟩ -b-> w⟧ ⟹ <noti(b),sigma> -b-> not(w)"
    andi:  "⟦⟨b0,sigma⟩ -b-> w0; ⟨b1,sigma⟩ -b-> w1⟧
           ⟹ <b0 andi b1,sigma> -b-> (w0 and w1)"
```

```
  ori:   "⟦⟨b0,sigma⟩ -b-> w0; ⟨b1,sigma⟩ -b-> w1⟧
         ⟹ <b0 ori b1,sigma> -b-> (w0 or w1)"
type_intros  bexp.intros
             apply_funtype and_type or_type bool_1I bool_0I not_type
type_elims   evala.dom_subset [THEN subsetD, elim_format]
```

1.3 Commands

```
consts com :: i
datatype com =
    skip                                    (‹skip› [])
  | assignment ("x ∈ loc", "a ∈ aexp")      (infixl ‹:=› 60)
  | semicolon ("c0 ∈ com", "c1 ∈ com")      (‹_; _›  [60, 60] 10)
  | while ("b ∈ bexp", "c ∈ com")           (‹while _ do _›  60)
  | "if" ("b ∈ bexp", "c0 ∈ com", "c1 ∈ com")   (‹if _ then _ else _› 60)


consts evalc :: i

abbreviation
  evalc_syntax :: "[i, i] ⇒ o"    (infixl ‹-c->› 50)
  where "p -c-> s ≡ ⟨p,s⟩ ∈ evalc"

inductive
  domains "evalc" ⊆ "(com × (loc -> nat)) × (loc -> nat)"
  intros
    skip:    "⟦sigma ∈ loc -> nat⟧ ⟹ <skip,sigma> -c-> sigma"

    assign:  "⟦m ∈ nat; x ∈ loc; ⟨a,sigma⟩ -a-> m⟧
              ⟹ <x := a,sigma> -c-> sigma(x:=m)"

    semi:    "⟦⟨c0,sigma⟩ -c-> sigma2; ⟨c1,sigma2⟩ -c-> sigma1⟧
              ⟹ <c0; c1, sigma> -c-> sigma1"

    if1:     "⟦b ∈ bexp; c1 ∈ com; sigma ∈ loc->nat;
                ⟨b,sigma⟩ -b-> 1; ⟨c0,sigma⟩ -c-> sigma1⟧
              ⟹ <if b then c0 else c1, sigma> -c-> sigma1"

    if0:     "⟦b ∈ bexp; c0 ∈ com; sigma ∈ loc->nat;
                ⟨b,sigma⟩ -b-> 0; ⟨c1,sigma⟩ -c-> sigma1⟧
               ⟹ <if b then c0 else c1, sigma> -c-> sigma1"

    while0:  "⟦c ∈ com; ⟨b, sigma⟩ -b-> 0⟧
               ⟹ <while b do c,sigma> -c-> sigma"

    while1:  "⟦c ∈ com; ⟨b,sigma⟩ -b-> 1; ⟨c,sigma⟩ -c-> sigma2;
                 <while b do c, sigma2> -c-> sigma1⟧
               ⟹ <while b do c, sigma> -c-> sigma1"

  type_intros  com.intros update_type
```

```
  type_elims  evala.dom_subset [THEN subsetD, elim_format]
              evalb.dom_subset [THEN subsetD, elim_format]
```

1.4 Misc lemmas

```
lemmas evala_1 [simp] = evala.dom_subset [THEN subsetD, THEN SigmaD1, THEN SigmaD1]
  and evala_2 [simp] = evala.dom_subset [THEN subsetD, THEN SigmaD1, THEN SigmaD2]
  and evala_3 [simp] = evala.dom_subset [THEN subsetD, THEN SigmaD2]

lemmas evalb_1 [simp] = evalb.dom_subset [THEN subsetD, THEN SigmaD1, THEN SigmaD1]
  and evalb_2 [simp] = evalb.dom_subset [THEN subsetD, THEN SigmaD1, THEN SigmaD2]
  and evalb_3 [simp] = evalb.dom_subset [THEN subsetD, THEN SigmaD2]

lemmas evalc_1 [simp] = evalc.dom_subset [THEN subsetD, THEN SigmaD1, THEN SigmaD1]
  and evalc_2 [simp] = evalc.dom_subset [THEN subsetD, THEN SigmaD1, THEN SigmaD2]
  and evalc_3 [simp] = evalc.dom_subset [THEN subsetD, THEN SigmaD2]

inductive_cases
    evala_N_E [elim!]: "<N(n),sigma> -a-> i"
  and evala_X_E [elim!]: "<X(x),sigma> -a-> i"
  and evala_Op1_E [elim!]: "<Op1(f,e),sigma> -a-> i"
  and evala_Op2_E [elim!]: "<Op2(f,a1,a2),sigma>  -a-> i"

end
```

2 Denotational semantics of expressions and commands

```
theory Denotation imports Com begin
```

2.1 Definitions

```
consts
  A     :: "i ⇒ i ⇒ i"
  B     :: "i ⇒ i ⇒ i"
  C     :: "i ⇒ i"

definition
  Gamma :: "[i,i,i] ⇒ i"  (‹Γ›) where
  "Γ(b,cden) ≡
    (λphi. {io ∈ (phi O cden). B(b,fst(io))=1} ∪
           {io ∈ id(loc->nat). B(b,fst(io))=0})"

primrec
  "A(N(n), sigma) = n"
  "A(X(x), sigma) = sigma‘x"
  "A(Op1(f,a), sigma) = f‘A(a,sigma)"
  "A(Op2(f,a0,a1), sigma) = f‘<A(a0,sigma),A(a1,sigma)>"

primrec
```

```
  "B(true, sigma) = 1"
  "B(false, sigma) = 0"
  "B(ROp(f,a0,a1), sigma) = f`<A(a0,sigma),A(a1,sigma)>"
  "B(noti(b), sigma) = not(B(b,sigma))"
  "B(b0 andi b1, sigma) = B(b0,sigma) and B(b1,sigma)"
  "B(b0 ori b1, sigma) = B(b0,sigma) or B(b1,sigma)"
```

primrec

```
  "C(skip) = id(loc->nat)"
  "C(x := a) =
    {io ∈ (loc->nat) × (loc->nat). snd(io) = fst(io)(x := A(a,fst(io)))}"
  "C(c0; c1) = C(c1) O C(c0)"
  "C(if b then c0 else c1) =
    {io ∈ C(c0). B(b,fst(io)) = 1} ∪ {io ∈ C(c1). B(b,fst(io)) = 0}"
  "C(while b do c) = lfp((loc->nat) × (loc->nat), Γ(b,C(c)))"
```

2.2 Misc lemmas

```
lemma A_type [TC]: "⟦a ∈ aexp; sigma ∈ loc->nat⟧ ⟹ A(a,sigma) ∈ nat"
  by (erule aexp.induct) simp_all

lemma B_type [TC]: "⟦b ∈ bexp; sigma ∈ loc->nat⟧ ⟹ B(b,sigma) ∈ bool"
by (erule bexp.induct, simp_all)

lemma C_subset: "c ∈ com ⟹ C(c) ⊆ (loc->nat) × (loc->nat)"
  apply (erule com.induct)
      apply simp_all
      apply (blast dest: lfp_subset [THEN subsetD])+
  done

lemma C_type_D [dest]:
    "⟦⟨x,y⟩ ∈ C(c); c ∈ com⟧ ⟹ x ∈ loc->nat ∧ y ∈ loc->nat"
  by (blast dest: C_subset [THEN subsetD])

lemma C_type_fst [dest]: "⟦x ∈ C(c); c ∈ com⟧ ⟹ fst(x) ∈ loc->nat"
  by (auto dest!: C_subset [THEN subsetD])

lemma Gamma_bnd_mono:
  "cden ⊆ (loc->nat) × (loc->nat)
    ⟹ bnd_mono ((loc->nat) × (loc->nat), Γ(b,cden))"
  by (unfold bnd_mono_def Gamma_def) blast

end
```

3 Equivalence

theory *Equiv* **imports** *Denotation Com* **begin**

lemma *aexp_iff [rule_format]:*

```
  "⟦a ∈ aexp; sigma: loc -> nat⟧
    ⟹ ∀n. ⟨a,sigma⟩ -a-> n ⟷ A(a,sigma) = n"
  apply (erule aexp.induct)
     apply (force intro!: evala.intros)+
  done

declare aexp_iff [THEN iffD1, simp]
        aexp_iff [THEN iffD2, intro!]

inductive_cases [elim!]:
  "⟨true,sigma⟩ -b-> x"
  "⟨false,sigma⟩ -b-> x"
  "<ROp(f,a0,a1),sigma> -b-> x"
  "<noti(b),sigma> -b-> x"
  "<b0 andi b1,sigma> -b-> x"
  "<b0 ori b1,sigma> -b-> x"


lemma bexp_iff [rule_format]:
  "⟦b ∈ bexp; sigma: loc -> nat⟧
    ⟹ ∀w. ⟨b,sigma⟩ -b-> w ⟷ B(b,sigma) = w"
  apply (erule bexp.induct)
  apply (auto intro!: evalb.intros)
  done

declare bexp_iff [THEN iffD1, simp]
        bexp_iff [THEN iffD2, intro!]

lemma com1: "⟨c,sigma⟩ -c-> sigma' ⟹ <sigma,sigma'> ∈ C(c)"
  apply (erule evalc.induct)
        apply (simp_all (no_asm_simp))
```

assign

```
     apply (simp add: update_type)
```

comp

```
    apply fast
```

while

```
   apply (erule Gamma_bnd_mono [THEN lfp_unfold, THEN ssubst, OF C_subset])
   apply (simp add: Gamma_def)
```

recursive case of *while*

```
  apply (erule Gamma_bnd_mono [THEN lfp_unfold, THEN ssubst, OF C_subset])
  apply (auto simp add: Gamma_def)
  done

declare B_type [intro!] A_type [intro!]
declare evalc.intros [intro]
```

```
lemma com2 [rule_format]: "c ∈ com ⟹ ∀x ∈ C(c). <c,fst(x)> -c-> snd(x)"
  apply (erule com.induct)

skip

      apply force

assign

     apply force

comp

    apply force

while

   apply safe
   apply simp_all
   apply (frule Gamma_bnd_mono [OF C_subset], erule Fixedpt.induct, assumption)
     unfolding Gamma_def
   apply force

if

  apply auto
  done
```

3.1 Main theorem

```
theorem com_equivalence:
    "c ∈ com ⟹ C(c) = {io ∈ (loc->nat) × (loc->nat). <c,fst(io)> -c-> snd(io)}"
  by (force intro: C_subset [THEN subsetD] elim: com2 dest: com1)

end
```

References

[1] Glynn Winskel. *The Formal Semantics of Programming Languages.* 1993.