

# NanoJava

David von Oheimb

Tobias Nipkow

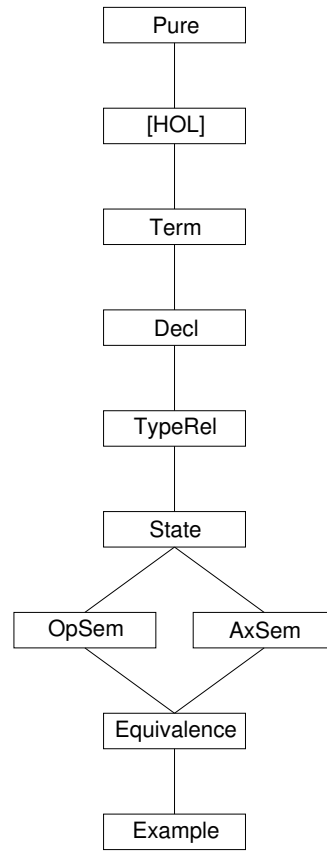
June 21, 2010

## Abstract

These theories define *NanoJava*, a very small fragment of the programming language Java (with essentially just classes) derived from the one given in [1]. For *NanoJava*, an operational semantics is given as well as a Hoare logic, which is proved both sound and (relatively) complete. The Hoare logic supports side-effecting expressions and implements a new approach for handling auxiliary variables. A more complex Hoare logic covering a much larger subset of Java is described in [3]. See also the homepage of project Bali at <http://isabelle.in.tum.de/Bali/> and the conference version of this document [2].

## Contents

<b>1</b>	<b>Statements and expression emulations</b>	<b>3</b>
<b>2</b>	<b>Types, class Declarations, and whole programs</b>	<b>3</b>
<b>3</b>	<b>Type relations</b>	<b>4</b>
3.1	Declarations and properties not used in the meta theory . . . . .	5
<b>4</b>	<b>Program State</b>	<b>7</b>
4.1	Properties not used in the meta theory . . . . .	8
<b>5</b>	<b>Operational Evaluation Semantics</b>	<b>10</b>
<b>6</b>	<b>Axiomatic Semantics</b>	<b>11</b>
6.1	Hoare Logic Rules . . . . .	12
6.2	Fully polymorphic variants, required for Example only . . . . .	13
6.3	Derived Rules . . . . .	13
<b>7</b>	<b>Equivalence of Operational and Axiomatic Semantics</b>	<b>15</b>
7.1	Validity . . . . .	15
7.2	Soundness . . . . .	16
7.3	(Relative) Completeness . . . . .	17
<b>8</b>	<b>Example</b>	<b>20</b>
8.1	Program representation . . . . .	21
8.2	“atleast” relation for interpretation of Nat “values” . . . . .	22
8.3	Proof(s) using the Hoare logic . . . . .	22



## 1 Statements and expression emulations

theory *Term* imports *Main* begin

typedecl *cname* — class name  
 typedecl *mname* — method name  
 typedecl *fname* — field name  
 typedecl *vname* — variable name

consts

*This* :: *vname* — This pointer  
*Par* :: *vname* — method parameter  
*Res* :: *vname* — method result

Inequality axioms are not required for the meta theory.

datatype *stmt*

= *Skip* — empty statement  
 | *Comp* *stmt stmt* (";; \_" [91,90 ] 90)  
 | *Cond* *expr stmt stmt* ("If '(\_)' \_ Else \_" [ 3,91,91] 91)  
 | *Loop* *vname stmt* ("While '(\_)' \_" [ 3,91 ] 91)  
 | *LAss* *vname expr* ("\_ := \_" [99, 95] 94) — local assignment  
 | *FAss* *expr fname expr* ("\_..\_:=\_" [95,99,95] 94) — field assignment  
 | *Meth* "*cname* × *mname*" — virtual method  
 | *Impl* "*cname* × *mname*" — method implementation

and *expr*

= *NewC* *cname* ("new \_" [ 99] 95) — object creation  
 | *Cast* *cname expr* — type cast  
 | *LAcc* *vname* — local access  
 | *FAcc* *expr fname* ("\_..\_" [95,99] 95) — field access  
 | *Call* *cname expr mname expr* ("{\_}\_..\_'(\_)" [99,95,99,95] 95) — method call

end

## 2 Types, class Declarations, and whole programs

theory *Decl* imports *Term* begin

datatype *ty*

= *NT* — null type  
 | *Class* *cname* — class type

Field declaration

types *fdecl*  
 = "*fname* × *ty*"

record *methd*

= *par* :: *ty*  
*res* :: *ty*  
*lcl* :: "(*vname* × *ty*) list"  
*bdy* :: *stmt*

Method declaration

types *mdecl*  
 = "*mname* × *methd*"

```

record "class"
  = super    :: cname
    flds     :: "fdecl list"
    methods  :: "mdecl list"

```

Class declaration

```

types cdecl
  = "cname × class"

```

```

types prog
  = "cdecl list"

```

translations

```

(type) "fdecl" ← (type) "fname × ty"
(type) "mdecl" ← (type) "mname × ty × ty × stmt"
(type) "class" ← (type) "cname × fdecl list × mdecl list"
(type) "cdecl" ← (type) "cname × class"
(type) "prog " ← (type) "cdecl list"

```

consts

```

Prog    :: prog      — program as a global value
Object  :: cname     — name of root class

```

```

definition "class" :: "cname → class" where
  "class      ≡ map_of Prog"

```

```

definition is_class  :: "cname => bool" where
  "is_class C ≡ class C ≠ None"

```

```

lemma finite_is_class: "finite {C. is_class C}"
apply (unfold is_class_def class_def)
apply (fold dom_def)
apply (rule finite_dom_map_of)
done

```

end

### 3 Type relations

theory TypeRel imports Decl begin

consts

```

subcls1 :: "(cname × cname) set" — subclass

```

abbreviation

```

subcls1_syntax :: "[cname, cname] => bool" ("_ <=C1 _" [71,71] 70)
where "C <=C1 D == (C,D) ∈ subcls1"

```

abbreviation

```

subcls_syntax  :: "[cname, cname] => bool" ("_ <=C _" [71,71] 70)
where "C <=C D == (C,D) ∈ subcls1^*"

```

notation (xsymbols)

```

subcls1_syntax ("_ <C1 _" [71,71] 70) and
subcls_syntax  ("_ ≤C _" [71,71] 70)

```

```

consts
  method :: "cname => (mname  $\rightarrow$  methd)"
  field  :: "cname => (fname  $\rightarrow$  ty)"

```

### 3.1 Declarations and properties not used in the meta theory

Direct subclass relation

```

defs
  subcls1_def: "subcls1  $\equiv$   $\{(C,D). C \neq \text{Object} \wedge (\exists c. \text{class } C = \text{Some } c \wedge \text{super } c = D)\}$ "

```

Widening, viz. method invocation conversion

```

inductive
  widen :: "ty => ty => bool"  ("_  $\preceq$  _" [71,71] 70)
where
  refl [intro!, simp]: "T  $\preceq$  T"
| subcls: "C  $\preceq$  C D  $\implies$  Class C  $\preceq$  Class D"
| null [intro!]: "NT  $\preceq$  R"

```

```

lemma subcls1D:
  "C  $\prec$  C1D  $\implies$  C  $\neq$  Object  $\wedge$   $(\exists c. \text{class } C = \text{Some } c \wedge \text{super } c = D)$ "
apply (unfold subcls1_def)
apply auto
done

```

```

lemma subcls1I: "[class C = Some m; super m = D; C  $\neq$  Object]  $\implies$  C  $\prec$  C1D"
apply (unfold subcls1_def)
apply auto
done

```

```

lemma subcls1_def2:
  "subcls1 =
    (SIGMA C: {C. is_class C} . {D. C  $\neq$  Object  $\wedge$  super (the (class C)) = D})"
apply (unfold subcls1_def is_class_def)
apply (auto split:split_if_asm)
done

```

```

lemma finite_subcls1: "finite subcls1"
apply(subst subcls1_def2)
apply(rule finite_SigmaI [OF finite_is_class])
apply(rule_tac B = "{super (the (class C))}" in finite_subset)
apply auto
done

```

```

definition ws_prog :: "bool" where
  "ws_prog  $\equiv$   $\forall (C,c) \in \text{set Prog}. C \neq \text{Object} \longrightarrow$ 
    is_class (super c)  $\wedge$  (super c, C)  $\notin$  subcls1 $^+$ "

```

```

lemma ws_progD: "[class C = Some c; C  $\neq$  Object; ws_prog]  $\implies$ 
  is_class (super c)  $\wedge$  (super c, C)  $\notin$  subcls1 $^+$ "
apply (unfold ws_prog_def class_def)
apply (drule_tac map_of_SomeD)
apply auto
done

```

```

lemma subcls1_irrefl_lemma1: "ws_prog  $\implies$  subcls1 $^{-1} \cap$  subcls1 $^+ = \{\}$ "
by (fast dest: subcls1D ws_progD)

```

```

lemma irrefl_tranclI': "r-1 Int r+ = {} ==> !x. (x, x) ~: r+"
by(blast elim: tranclE dest: trancl_into_rtrancl)

lemmas subcls1_irrefl_lemma2 = subcls1_irrefl_lemma1 [THEN irrefl_tranclI']

lemma subcls1_irrefl: "[[ (x, y) ∈ subcls1; ws_prog ] ⇒ x ≠ y"
apply (rule irrefl_trancl_rD)
apply (rule subcls1_irrefl_lemma2)
apply auto
done

lemmas subcls1_acyclic = subcls1_irrefl_lemma2 [THEN acyclicI, standard]

lemma wf_subcls1: "ws_prog ⇒ wf (subcls1-1)"
by (auto intro: finite_acyclic_wf_converse finite_subcls1 subcls1_acyclic)

consts class_rec :: "cname ⇒ (class ⇒ ('a × 'b) list) ⇒ ('a → 'b)"

recdef (permissive) class_rec "subcls1-1"
  "class_rec C = (λf. case class C of None ⇒ undefined
    | Some m ⇒ if wf (subcls1-1)
    then (if C=Object then empty else class_rec (super m) f) ++ map_of (f m)
    else undefined)"
(hints intro: subcls1I)

lemma class_rec: "[[class C = Some m; ws_prog] ⇒
  class_rec C f = (if C = Object then empty else class_rec (super m) f) ++
    map_of (f m)"
apply (drule wf_subcls1)
apply (rule class_rec.simps [THEN trans [THEN fun_cong]])
apply assumption
apply simp
done

— Methods of a class, with inheritance and hiding
defs method_def: "method C ≡ class_rec C methods"

lemma method_rec: "[[class C = Some m; ws_prog] ⇒
  method C = (if C=Object then empty else method (super m)) ++ map_of (methods m)"
apply (unfold method_def)
apply (erule (1) class_rec [THEN trans])
apply simp
done

— Fields of a class, with inheritance and hiding
defs field_def: "field C ≡ class_rec C flds"

lemma flds_rec: "[[class C = Some m; ws_prog] ⇒
  field C = (if C=Object then empty else field (super m)) ++ map_of (flds m)"
apply (unfold field_def)
apply (erule (1) class_rec [THEN trans])
apply simp
done

end

```

## 4 Program State

**theory** *State* **imports** *TypeRel* **begin**

**definition** *body* :: "*cname* × *mname* => *stmt*" **where**  
*"body* ≡ λ(*C,m*). *bdy* (*the* (*method C m*))"

Locations, i.e. abstract references to objects

**typeddecl** *loc*

**datatype** *val*  
 = *Null* — null reference  
 / *Addr loc* — address, i.e. location of object

**types** *fields*  
 = "*fname* → *val*"  
  
*obj* = "*cname* × *fields*"

**translations**  
 (type) "*fields*" ← (type) "*fname* => *val option*"  
 (type) "*obj*" ← (type) "*cname* × *fields*"

**definition** *init\_vars* :: "('a → 'b) => ('a → *val*)" **where**  
*"init\_vars m* == *Option.map* (λ*T*. *Null*) o *m*"

private:

**types** *heap* = "*loc* → *obj*"  
*locals* = "*vname* → *val*"

private:

**record** *state*  
 = *heap* :: *heap*  
*locals* :: *locals*

**translations**  
 (type) "*heap*" ← (type) "*loc* => *obj option*"  
 (type) "*locals*" ← (type) "*vname* => *val option*"  
 (type) "*state*" ← (type) "(|*heap* :: *heap*, *locals* :: *locals*|)"

**definition** *del\_locs* :: "*state* => *state*" **where**  
*"del\_locs s* ≡ *s* (| *locals* := *empty* |)"

**definition** *init\_locs* :: "*cname* => *mname* => *state* => *state*" **where**  
*"init\_locs C m s* ≡ *s* (| *locals* := *locals s* ++  
*init\_vars* (*map\_of* (*lcl* (*the* (*method C m*)))) |)"

The first parameter of *set\_locs* is of type *state* rather than *locals* in order to keep *locals* private.

**definition** *set\_locs* :: "*state* => *state* => *state*" **where**  
*"set\_locs s s'* ≡ *s'* (| *locals* := *locals s* |)"

**definition** *get\_local* :: "*state* => *vname* => *val*" ("*\_<\_>*" [99,0] 99) **where**  
*"get\_local s x* ≡ *the* (*locals s x*)"

— local function:

**definition** *get\_obj* :: "*state* => *loc* => *obj*" **where**  
*"get\_obj s a* ≡ *the* (*heap s a*)"

```

definition obj_class      :: "state => loc => cname" where
  "obj_class s a  $\equiv$  fst (get_obj s a)"

definition get_field      :: "state => loc => fname => val" where
  "get_field s a f  $\equiv$  the (snd (get_obj s a) f)"

— local function:
definition hupd           :: "loc => obj => state => state" ("hupd'(_|->_)" [10,10] 1000) where
  "hupd a obj s  $\equiv$  s (| heap := ((heap s)(a $\mapsto$ obj))|)"

definition lupd          :: "vname => val => state => state" ("lupd'(_|->_)" [10,10] 1000) where
  "lupd x v s  $\equiv$  s (| locals := ((locals s)(x $\mapsto$ v))|)"

notation (xsymbols)
  hupd ("hupd'(_ $\mapsto$ _)" [10,10] 1000) and
  lupd ("lupd'(_ $\mapsto$ _)" [10,10] 1000)

definition new_obj :: "loc => cname => state => state" where
  "new_obj a C  $\equiv$  hupd(a $\mapsto$ (C,init_vars (field C)))"

definition upd_obj      :: "loc => fname => val => state => state" where
  "upd_obj a f v s  $\equiv$  let (C,fs) = the (heap s a) in hupd(a $\mapsto$ (C,fs(f $\mapsto$ v))) s"

definition new_Addr     :: "state => val" where
  "new_Addr s == SOME v. ( $\exists$  a. v = Addr a  $\wedge$  (heap s) a = None) | v = Null"

```

#### 4.1 Properties not used in the meta theory

```

lemma locals_upd_id [simp]: "s(|locals := locals s|) = s"
by simp

lemma lupd_get_local_same [simp]: "lupd(x $\mapsto$ v) s<x> = v"
by (simp add: lupd_def get_local_def)

lemma lupd_get_local_other [simp]: "x  $\neq$  y  $\implies$  lupd(x $\mapsto$ v) s<y> = s<y>"
apply (drule not_sym)
by (simp add: lupd_def get_local_def)

lemma get_field_lupd [simp]:
  "get_field (lupd(x $\mapsto$ y) s) a f = get_field s a f"
by (simp add: lupd_def get_field_def get_obj_def)

lemma get_field_set_locs [simp]:
  "get_field (set_locs l s) a f = get_field s a f"
by (simp add: lupd_def get_field_def set_locs_def get_obj_def)

lemma get_field_del_locs [simp]:
  "get_field (del_locs s) a f = get_field s a f"
by (simp add: lupd_def get_field_def del_locs_def get_obj_def)

lemma new_obj_get_local [simp]: "new_obj a C s <x> = s<x>"
by (simp add: new_obj_def hupd_def get_local_def)

lemma heap_lupd [simp]: "heap (lupd(x $\mapsto$ y) s) = heap s"
by (simp add: lupd_def)

lemma heap_hupd_same [simp]: "heap (hupd(a $\mapsto$ obj) s) a = Some obj"
by (simp add: hupd_def)

```



```
lemma heap_hupd_other [simp]: "aa ≠ a ⇒ heap (hupd(aa↦obj) s) a = heap s a"
apply (drule not_sym)
by (simp add: hupd_def)
```

```
lemma hupd_hupd [simp]: "hupd(a↦obj) (hupd(a↦obj') s) = hupd(a↦obj) s"
by (simp add: hupd_def)
```

```
lemma heap_del_locs [simp]: "heap (del_locs s) = heap s"
by (simp add: del_locs_def)
```

```
lemma heap_set_locs [simp]: "heap (set_locs l s) = heap s"
by (simp add: set_locs_def)
```

```
lemma hupd_lupd [simp]:
  "hupd(a↦obj) (lupd(x↦y) s) = lupd(x↦y) (hupd(a↦obj) s)"
by (simp add: hupd_def lupd_def)
```

```
lemma hupd_del_locs [simp]:
  "hupd(a↦obj) (del_locs s) = del_locs (hupd(a↦obj) s)"
by (simp add: hupd_def del_locs_def)
```

```
lemma new_obj_lupd [simp]:
  "new_obj a C (lupd(x↦y) s) = lupd(x↦y) (new_obj a C s)"
by (simp add: new_obj_def)
```

```
lemma new_obj_del_locs [simp]:
  "new_obj a C (del_locs s) = del_locs (new_obj a C s)"
by (simp add: new_obj_def)
```

```
lemma upd_obj_lupd [simp]:
  "upd_obj a f v (lupd(x↦y) s) = lupd(x↦y) (upd_obj a f v s)"
by (simp add: upd_obj_def Let_def split_beta)
```

```
lemma upd_obj_del_locs [simp]:
  "upd_obj a f v (del_locs s) = del_locs (upd_obj a f v s)"
by (simp add: upd_obj_def Let_def split_beta)
```

```
lemma get_field_hupd_same [simp]:
  "get_field (hupd(a↦(C, fs)) s) a = the ∘ fs"
apply (rule ext)
by (simp add: get_field_def get_obj_def)
```

```
lemma get_field_hupd_other [simp]:
  "aa ≠ a ⇒ get_field (hupd(aa↦obj) s) a = get_field s a"
apply (rule ext)
by (simp add: get_field_def get_obj_def)
```

```
lemma new_AddrD:
  "new_Addr s = v ⇒ (∃ a. v = Addr a ∧ heap s a = None) | v = Null"
apply (unfold new_Addr_def)
apply (erule subst)
apply (rule someI)
apply (rule disjI2)
apply (rule HOL.refl)
done
```

```
end
```

## 5 Operational Evaluation Semantics

theory OpSem imports State begin

inductive

exec :: "[state,stmt, nat,state] => bool" ("\_ ->-> \_" [98,90, 65,98] 89)  
 and eval :: "[state,expr,val,nat,state] => bool" ("\_ ->->-> \_" [98,95,99,65,98] 89)

where

Skip: " s -Skip-n-> s "

| Comp: "[| s0 -c1-n-> s1; s1 -c2-n-> s2 |] ==>  
 s0 -c1;; c2-n-> s2"

| Cond: "[| s0 -e>v-n-> s1; s1 -(if v≠Null then c1 else c2)-n-> s2 |] ==>  
 s0 -If(e) c1 Else c2-n-> s2"

| LoopF: " s0<x> = Null ==>  
 s0 -While(x) c-n-> s0 "

| LoopT: "[| s0<x> ≠ Null; s0 -c-n-> s1; s1 -While(x) c-n-> s2 |] ==>  
 s0 -While(x) c-n-> s2 "

| LAcc: " s -LAcc x>s<x>-n-> s "

| LAss: " s -e>v-n-> s' ==>  
 s -x:=e-n-> lupd(x↦v) s' "

| FAcc: " s -e>Addr a-n-> s' ==>  
 s -e..f>get\_field s' a f-n-> s' "

| FAss: "[| s0 -e1>Addr a-n-> s1; s1 -e2>v-n-> s2 |] ==>  
 s0 -e1..f:=e2-n-> upd\_obj a f v s2 "

| NewC: " new\_Addr s = Addr a ==>  
 s -new C>Addr a-n-> new\_obj a C s "

| Cast: "[| s -e>v-n-> s';  
 case v of Null => True | Addr a => obj\_class s' a ≤C C |] ==>  
 s -Cast C e>v-n-> s' "

| Call: "[| s0 -e1>a-n-> s1; s1 -e2>p-n-> s2;  
 lupd(This↦a)(lupd(Par↦p)(del\_locs s2)) -Meth (C,m)-n-> s3  
 |] ==> s0 -{C}e1..m(e2)>s3<Res>-n-> set\_locs s2 s3 "

| Meth: "[| s<This> = Addr a; D = obj\_class s a; D ≤C C;  
 init\_locs D m s -Impl (D,m)-n-> s' |] ==>  
 s -Meth (C,m)-n-> s' "

| Impl: " s -body Cm- n-> s' ==>  
 s -Impl Cm-Suc n-> s' "

inductive\_cases exec\_elim\_cases':

"s -Skip -n-> t"  
 "s -c1;; c2 -n-> t"  
 "s -If(e) c1 Else c2-n-> t"  
 "s -While(x) c -n-> t"  
 "s -x:=e -n-> t"  
 "s -e1..f:=e2 -n-> t"

inductive\_cases Meth\_elim\_cases: "s -Meth Cm -n-> t"

```

inductive_cases Impl_elim_cases: "s -Impl Cm          -n→ t"
lemmas exec_elim_cases = exec_elim_cases' Meth_elim_cases Impl_elim_cases
inductive_cases eval_elim_cases:
    "s -new C          >v-n→ t"
    "s -Cast C e       >v-n→ t"
    "s -LAcc x         >v-n→ t"
    "s -e..f           >v-n→ t"
    "s -{C}e1..m(e2)   >v-n→ t"

lemma exec_eval_mono [rule_format]:
  "(s -c -n→ t → (∀m. n ≤ m → s -c -m→ t)) ∧
   (s -e>v-n→ t → (∀m. n ≤ m → s -e>v-m→ t))"
apply (rule exec_eval.induct)
prefer 14
apply clarify
apply (rename_tac n)
apply (case_tac n)
apply (blast intro:exec_eval.intros)+
done
lemmas exec_mono = exec_eval_mono [THEN conjunct1, rule_format]
lemmas eval_mono = exec_eval_mono [THEN conjunct2, rule_format]

lemma exec_exec_max: "[[s1 -c1-    n1    → t1 ; s2 -c2-    n2→ t2]] ⇒
  s1 -c1-max n1 n2→ t1 ∧ s2 -c2-max n1 n2→ t2"
by (fast intro: exec_mono le_maxI1 le_maxI2)

lemma eval_exec_max: "[[s1 -c-    n1    → t1 ; s2 -e>v-    n2→ t2]] ⇒
  s1 -c-max n1 n2→ t1 ∧ s2 -e>v-max n1 n2→ t2"
by (fast intro: eval_mono exec_mono le_maxI1 le_maxI2)

lemma eval_eval_max: "[[s1 -e1>v1-    n1    → t1 ; s2 -e2>v2-    n2→ t2]] ⇒
  s1 -e1>v1-max n1 n2→ t1 ∧ s2 -e2>v2-max n1 n2→ t2"
by (fast intro: eval_mono le_maxI1 le_maxI2)

lemma eval_eval_exec_max:
  "[[s1 -e1>v1-n1→ t1; s2 -e2>v2-n2→ t2; s3 -c-n3→ t3]] ⇒
   s1 -e1>v1-max (max n1 n2) n3→ t1 ∧
   s2 -e2>v2-max (max n1 n2) n3→ t2 ∧
   s3 -c -max (max n1 n2) n3→ t3"
apply (drule (1) eval_eval_max, erule thin_rl)
by (fast intro: exec_mono eval_mono le_maxI1 le_maxI2)

lemma Impl_body_eq: "(λt. ∃n. Z -Impl M-n→ t) = (λt. ∃n. Z -body M-n→ t)"
apply (rule ext)
apply (fast elim: exec_elim_cases intro: exec_eval.Impl)
done

end

```

## 6 Axiomatic Semantics

```
theory AxSem imports State begin
```

```

types assn    = "state => bool"
  vassn    = "val => assn"
  triple = "assn × stmt × assn"
  etriple = "assn × expr × vassn"
translations

```

```

(type) "assn"  $\leftarrow$  (type) "state  $\Rightarrow$  bool"
(type) "vassn"  $\leftarrow$  (type) "val  $\Rightarrow$  assn"
(type) "triple"  $\leftarrow$  (type) "assn  $\times$  stmt  $\times$  assn"
(type) "etriples"  $\leftarrow$  (type) "assn  $\times$  expr  $\times$  vassn"

```

## 6.1 Hoare Logic Rules

inductive

```

hoare :: "[triple set, triple set]  $\Rightarrow$  bool" ("_  $\vdash$  / _" [61, 61] 60)
and ehoare :: "[triple set, etriples]  $\Rightarrow$  bool" ("_  $\vdash_e$  / _" [61, 61] 60)
and hoare1 :: "[triple set, assn, stmt, assn]  $\Rightarrow$  bool"
  ("_  $\vdash$  / ({(1_)} / (_) / {(1_)} )" [61, 3, 90, 3] 60)
and ehoare1 :: "[triple set, assn, expr, vassn]  $\Rightarrow$  bool"
  ("_  $\vdash_e$  / ({(1_)} / (_) / {(1_)} )" [61, 3, 90, 3] 60)
where

  "A  $\vdash$  {P} c {Q}  $\equiv$  A  $\vdash$  {(P, c, Q)}"
| "A  $\vdash_e$  {P} e {Q}  $\equiv$  A  $\vdash_e$  (P, e, Q)"

| Skip: "A  $\vdash$  {P} Skip {P}"

| Comp: "[| A  $\vdash$  {P} c1 {Q}; A  $\vdash$  {Q} c2 {R} |]  $\Rightarrow$  A  $\vdash$  {P} c1;;c2 {R}"

| Cond: "[| A  $\vdash_e$  {P} e {Q};
   $\forall v. A \vdash \{Q\ v\}$  (if v  $\neq$  Null then c1 else c2) {R} |]  $\Rightarrow$ 
  A  $\vdash$  {P} If(e) c1 Else c2 {R}"

| Loop: "A  $\vdash$  { $\lambda s. P\ s \wedge s\langle x \rangle \neq \text{Null}$ } c {P}  $\Rightarrow$ 
  A  $\vdash$  {P} While(x) c { $\lambda s. P\ s \wedge s\langle x \rangle = \text{Null}$ }"

| LAcc: "A  $\vdash_e$  { $\lambda s. P\ (s\langle x \rangle)\ s$ } LAcc x {P}"

| LAss: "A  $\vdash_e$  {P} e { $\lambda v\ s. Q\ (\text{lupd}(x \mapsto v)\ s)$ }  $\Rightarrow$ 
  A  $\vdash$  {P} x ::= e {Q}"

| FAcc: "A  $\vdash_e$  {P} e { $\lambda v\ s. \forall a. v = \text{Addr}\ a \rightarrow Q\ (\text{get\_field}\ s\ a\ f)\ s$ }  $\Rightarrow$ 
  A  $\vdash_e$  {P} e..f {Q}"

| FAss: "[| A  $\vdash_e$  {P} e1 { $\lambda v\ s. \forall a. v = \text{Addr}\ a \rightarrow Q\ a\ s$ };
   $\forall a. A \vdash_e \{Q\ a\}\ e2\ \{\lambda v\ s. R\ (\text{upd\_obj}\ a\ f\ v\ s)\}$  |]  $\Rightarrow$ 
  A  $\vdash$  {P} e1..f::=e2 {R}"

| NewC: "A  $\vdash_e$  { $\lambda s. \forall a. \text{new\_Addr}\ s = \text{Addr}\ a \rightarrow P\ (\text{Addr}\ a)\ (\text{new\_obj}\ a\ C\ s)$ }
  new C {P}"

| Cast: "A  $\vdash_e$  {P} e { $\lambda v\ s. (\text{case } v \text{ of Null} \Rightarrow \text{True}$ 
  | Addr a  $\Rightarrow$  obj\_class s a  $\leq_C$  C)  $\rightarrow Q\ v\ s$ }  $\Rightarrow$ 
  A  $\vdash_e$  {P} Cast C e {Q}"

| Call: "[| A  $\vdash_e$  {P} e1 {Q};  $\forall a. A \vdash_e \{Q\ a\}\ e2\ \{R\ a\}$ ;
   $\forall a\ p\ ls. A \vdash \{\lambda s'. \exists s. R\ a\ p\ s \wedge ls = s \wedge$ 
    s' =  $\text{lupd}(\text{This} \mapsto a)(\text{lupd}(\text{Par} \mapsto p)(\text{del\_locs}\ s))\}$ 
    Meth (C,m) { $\lambda s. S\ (s\langle \text{Res} \rangle)\ (\text{set\_locs}\ ls\ s)\}$  |]  $\Rightarrow$ 
  A  $\vdash_e$  {P} {C}e1..m(e2) {S}"

| Meth: " $\forall D. A \vdash \{\lambda s'. \exists s\ a. s\langle \text{This} \rangle = \text{Addr}\ a \wedge D = \text{obj\_class}\ s\ a \wedge D \leq_C C \wedge$ 
  P s  $\wedge s' = \text{init\_locs}\ D\ m\ s\}$ 
  Impl (D,m) {Q}  $\Rightarrow$ 
  A  $\vdash$  {P} Meth (C,m) {Q}"

```

—  $\bigcup Z$  instead of  $\forall Z$  in the conclusion and  
 $Z$  restricted to type state due to limitations of the inductive package

```
| Impl: "∀Z::state. A ∪ (⋃Z. (λCm. (P Z Cm, Impl Cm, Q Z Cm))'Ms) ⊢
      (λCm. (P Z Cm, body Cm, Q Z Cm))'Ms ==>
      A ⊢ (λCm. (P Z Cm, Impl Cm, Q Z Cm))'Ms"
```

— structural rules

```
| Asm: "  a ∈ A ==> A ⊢ {a}"

| ConjI: " ∀c ∈ C. A ⊢ {c} ==> A ⊢ C"

| ConjE: "[| A ⊢ C; c ∈ C |] ==> A ⊢ {c}"

— Z restricted to type state due to limitations of the inductive package
| Conseq: "[| ∀Z::state. A ⊢ {P' Z} c {Q' Z};
      ∀s t. (∀Z. P' Z s --> Q' Z t) --> (P s --> Q t) |] ==>
      A ⊢ {P} c {Q }"
```

—  $Z$  restricted to type state due to limitations of the inductive package

```
| eConseq: "[| ∀Z::state. A ⊢e {P' Z} e {Q' Z};
      ∀s v t. (∀Z. P' Z s --> Q' Z v t) --> (P s --> Q v t) |] ==>
      A ⊢e {P} e {Q }"
```

## 6.2 Fully polymorphic variants, required for Example only

axioms

```
Conseq: "[| ∀Z. A ⊢ {P' Z} c {Q' Z};
      ∀s t. (∀Z. P' Z s --> Q' Z t) --> (P s --> Q t) |] ==>
      A ⊢ {P} c {Q }"

eConseq: "[| ∀Z. A ⊢e {P' Z} e {Q' Z};
      ∀s v t. (∀Z. P' Z s --> Q' Z v t) --> (P s --> Q v t) |] ==>
      A ⊢e {P} e {Q }"

Impl: "∀Z. A ∪ (⋃Z. (λCm. (P Z Cm, Impl Cm, Q Z Cm))'Ms) ⊢
      (λCm. (P Z Cm, body Cm, Q Z Cm))'Ms ==>
      A ⊢ (λCm. (P Z Cm, Impl Cm, Q Z Cm))'Ms"
```

## 6.3 Derived Rules

```
lemma Conseq1: "[| A ⊢ {P'} c {Q}; ∀s. P s → P' s |] ==> A ⊢ {P} c {Q}"
apply (rule hoare_ehoare.Conseq)
apply (rule allI, assumption)
apply fast
done
```

```
lemma Conseq2: "[| A ⊢ {P} c {Q'}; ∀t. Q' t → Q t |] ==> A ⊢ {P} c {Q}"
apply (rule hoare_ehoare.Conseq)
apply (rule allI, assumption)
apply fast
done
```

```
lemma eConseq1: "[| A ⊢e {P'} e {Q}; ∀s. P s → P' s |] ==> A ⊢e {P} e {Q}"
apply (rule hoare_ehoare.eConseq)
apply (rule allI, assumption)
apply fast
```

done

```
lemma eConseq2: "[[A ⊢e {P} e {Q'}]; ∀ v t. Q' v t → Q v t] ⇒ A ⊢e {P} e {Q}"
apply (rule hoare_ehoare.eConseq)
apply (rule allI, assumption)
apply fast
done
```

```
lemma Weaken: "[[A ⊢ C'; C ⊆ C']] ⇒ A ⊢ C"
apply (rule hoare_ehoare.ConjI)
apply clarify
apply (drule hoare_ehoare.ConjE)
apply fast
apply assumption
done
```

```
lemma Thin_lemma:
  "(A' ⊢ C → (∀ A. A' ⊆ A → A ⊢ C)) ∧
  (A' ⊢e {P} e {Q} → (∀ A. A' ⊆ A → A ⊢e {P} e {Q}))"
apply (rule hoare_ehoare.induct)
apply (tactic "ALLGOALS(EVERY'[clarify_tac @ {claset}, REPEAT o smp_tac 1])")
apply (blast intro: hoare_ehoare.Skip)
apply (blast intro: hoare_ehoare.Comp)
apply (blast intro: hoare_ehoare.Cond)
apply (blast intro: hoare_ehoare.Loop)
apply (blast intro: hoare_ehoare.LAcc)
apply (blast intro: hoare_ehoare.LAss)
apply (blast intro: hoare_ehoare.FAcc)
apply (blast intro: hoare_ehoare.FAss)
apply (blast intro: hoare_ehoare.NewC)
apply (blast intro: hoare_ehoare.Cast)
apply (erule hoare_ehoare.Call)
apply (rule, drule spec, erule conjE, tactic "smp_tac 1 1", assumption)
apply blast
apply (blast intro!: hoare_ehoare.Meth)
apply (blast intro!: hoare_ehoare.Impl)
apply (blast intro!: hoare_ehoare.Asm)
apply (blast intro: hoare_ehoare.ConjI)
apply (blast intro: hoare_ehoare.ConjE)
apply (rule hoare_ehoare.Conseq)
apply (rule, drule spec, erule conjE, tactic "smp_tac 1 1", assumption+)
apply (rule hoare_ehoare.eConseq)
apply (rule, drule spec, erule conjE, tactic "smp_tac 1 1", assumption+)
done
```

```
lemma cThin: "[[A' ⊢ C; A' ⊆ A]] ⇒ A ⊢ C"
by (erule (1) conjunct1 [OF Thin_lemma, rule_format])
```

```
lemma eThin: "[[A' ⊢e {P} e {Q}; A' ⊆ A]] ⇒ A ⊢e {P} e {Q}"
by (erule (1) conjunct2 [OF Thin_lemma, rule_format])
```

```
lemma Union: "A ⊢ (⋃ Z. C Z) = (∀ Z. A ⊢ C Z)"
by (auto intro: hoare_ehoare.ConjI hoare_ehoare.ConjE)
```

```
lemma Impl1':
  "[[∀ Z::state. A ∪ (⋃ Z. (λ Cm. (P Z Cm, Impl Cm, Q Z Cm)) 'Ms) ⊢
    (λ Cm. (P Z Cm, body Cm, Q Z Cm)) 'Ms;
   Cm ∈ Ms]] ⇒
```

```

      A ⊢ {P Z Cm} Impl Cm {Q Z Cm}"
apply (drule AxSem.Impl)
apply (erule Weaken)
apply (auto del: image_eqI intro: rev_image_eqI)
done

lemmas Impl1 = AxSem.Impl [of _ _ _ "{Cm}", simplified, standard]

end

```

## 7 Equivalence of Operational and Axiomatic Semantics

theory *Equivalence* imports *OpSem* *AxSem* begin

### 7.1 Validity

definition *valid* :: "[assn,stmt, assn] => bool" ("|= {(1\_)} / ( \_ ) / {(1\_)}" [3,90,3] 60) where  

$$|= \{P\} c \{Q\} \equiv \forall s \ t. P \ s \ \rightarrow (\exists n. s \ -c \ -n \rightarrow t) \ \rightarrow Q \ t$$

definition *evalid* :: "[assn,expr,vassn] => bool" ("|=e {(1\_)} / ( \_ ) / {(1\_)}" [3,90,3] 60) where  

$$|=e \{P\} e \{Q\} \equiv \forall s \ v \ t. P \ s \ \rightarrow (\exists n. s \ -e \triangleright v \ -n \rightarrow t) \ \rightarrow Q \ v \ t$$

definition *nvalid* :: "[nat, triple ] => bool" ("|=\_: \_" [61,61] 60) where  

$$|=n: t \equiv \text{let } (P,c,Q) = t \text{ in } \forall s \ t. s \ -c \ -n \rightarrow t \ \rightarrow P \ s \ \rightarrow Q \ t$$

definition *envalid* :: "[nat, etriple ] => bool" ("|=\_:e \_" [61,61] 60) where  

$$|=n:e \ t \equiv \text{let } (P,e,Q) = t \text{ in } \forall s \ v \ t. s \ -e \triangleright v \ -n \rightarrow t \ \rightarrow P \ s \ \rightarrow Q \ v \ t$$

definition *nvalids* :: "[nat, triple set] => bool" ("||=: \_" [61,61] 60) where  

$$||=: T \equiv \forall t \in T. |=n: t$$

definition *cnvalids* :: "[triple set, triple set] => bool" ("\_ ||=/ \_" [61,61] 60) where  

$$A ||=: C \equiv \forall n. ||=:n: A \ \rightarrow ||=:n: C$$

definition *cenvalid* :: "[triple set, etriple ] => bool" ("\_ ||=:e \_" [61,61] 60) where  

$$A ||=:e \ t \equiv \forall n. ||=:n: A \ \rightarrow ||=:n:e \ t$$

notation (xsymbols)  

$$\text{valid } ("|= \{(1\_)\} / ( \_ ) / \{(1\_)\}" [3,90,3] 60) \text{ and}$$
  

$$\text{evalid } ("|=e \{(1\_)\} / ( \_ ) / \{(1\_)\}" [3,90,3] 60) \text{ and}$$
  

$$\text{nvalid } ("|=_: \_ " [61,61] 60) \text{ and}$$
  

$$\text{envalid } ("|=_:e \_ " [61,61] 60) \text{ and}$$
  

$$\text{nvalids } ("||=: \_ " [61,61] 60) \text{ and}$$
  

$$\text{cnvalids } ("_ ||=/ \_ " [61,61] 60) \text{ and}$$
  

$$\text{cenvalid } ("_ ||=:e \_ " [61,61] 60)$$

lemma *nvalid\_def2*: " $|=n: (P,c,Q) \equiv \forall s \ t. s \ -c \ -n \rightarrow t \ \rightarrow P \ s \ \rightarrow Q \ t$ "  
 by (simp add: *nvalid\_def* *Let\_def*)

lemma *valid\_def2*: " $|= \{P\} c \{Q\} = (\forall n. |=n: (P,c,Q))$ "  
 apply (simp add: *valid\_def* *nvalid\_def2*)  
 apply blast  
 done

lemma *envalid\_def2*: " $|=n:e (P,e,Q) \equiv \forall s \ v \ t. s \ -e \triangleright v \ -n \rightarrow t \ \rightarrow P \ s \ \rightarrow Q \ v \ t$ "  
 by (simp add: *envalid\_def* *Let\_def*)

```

lemma evalid_def2: " $\models_e \{P\} e \{Q\} = (\forall n. \models_{n:e} (P, e, Q))$ "
apply (simp add: evalid_def envalid_def2)
apply blast
done

```

```

lemma cenvalid_def2:
  " $A \models_e (P, e, Q) = (\forall n. \models_n A \longrightarrow (\forall s \ v \ t. s \text{-}e \text{-}v \text{-}n \longrightarrow t \longrightarrow P \ s \longrightarrow Q \ v \ t))$ "
by (simp add: cenvalid_def envalid_def2)

```

## 7.2 Soundness

```

declare exec_elim_cases [elim!] eval_elim_cases [elim!]

```

```

lemma Impl_nvalid_0: " $\models_0: (P, \text{Impl } M, Q)$ "
by (clarsimp simp add: nvalid_def2)

```

```

lemma Impl_nvalid_Suc: " $\models_n: (P, \text{body } M, Q) \implies \models_{\text{Suc } n}: (P, \text{Impl } M, Q)$ "
by (clarsimp simp add: nvalid_def2)

```

```

lemma nvalid_SucD: " $\bigwedge t. \models_{\text{Suc } n} t \implies \models_n t$ "
by (force simp add: split_paired_all nvalid_def2 intro: exec_mono)

```

```

lemma nvalids_SucD: " $\text{Ball } A \ (\text{nvalid } (\text{Suc } n)) \implies \text{Ball } A \ (\text{nvalid } n)$ "
by (fast intro: nvalid_SucD)

```

```

lemma Loop_sound_lemma [rule_format (no_asm)]:
  " $\forall s \ t. s \text{-}c \text{-}n \longrightarrow t \longrightarrow P \ s \wedge s \langle x \rangle \neq \text{Null} \longrightarrow P \ t \implies$ 
   $(s \text{-}c0 \text{-}n0 \longrightarrow t \longrightarrow P \ s \longrightarrow c0 = \text{While } (x) \ c \longrightarrow n0 = n \longrightarrow P \ t \wedge t \langle x \rangle = \text{Null})$ "
apply (rule_tac ?P2.1="%s e v n t. True" in exec_eval.induct [THEN conjunct1])
apply clarsimp+
done

```

```

lemma Impl_sound_lemma:
  " $\llbracket \forall z \ n. \text{Ball } (A \cup B) \ (\text{nvalid } n) \longrightarrow \text{Ball } (f \ z \ ' \ Ms) \ (\text{nvalid } n);$ 
   $\text{Cm} \in Ms; \text{Ball } A \ (\text{nvalid } na); \text{Ball } B \ (\text{nvalid } na) \rrbracket \implies \text{nvalid } na \ (f \ z \ \text{Cm})$ "
by blast

```

```

lemma all_conjunct2: " $\forall l. P' \ l \wedge P \ l \implies \forall l. P \ l$ "
by fast

```

```

lemma all3_conjunct2:
  " $\forall a \ p \ l. (P' \ a \ p \ l \wedge P \ a \ p \ l) \implies \forall a \ p \ l. P \ a \ p \ l$ "
by fast

```

```

lemma cnvalid1_eq:
  " $A \models \{(P, c, Q)\} \equiv \forall n. \models_n A \longrightarrow (\forall s \ t. s \text{-}c \text{-}n \longrightarrow t \longrightarrow P \ s \longrightarrow Q \ t)$ "
by (simp add: cnvalids_def nvalids_def nvalid_def2)

```

```

lemma hoare_sound_main: " $\bigwedge t. (A \vdash C \longrightarrow A \models C) \wedge (A \vdash_e t \longrightarrow A \models_e t)$ "
apply (tactic "split_all_tac 1", rename_tac P e Q)
apply (rule hoare_ehoare.induct)

```

```

apply (tactic {* ALLGOALS (REPEAT o dresolve_tac [@{thm all_conjunct2}, @{thm all3_conjunct2}])
*})
apply (tactic {* ALLGOALS (REPEAT o thin_tac @{context} "hoare ?x ?y") *})
apply (tactic {* ALLGOALS (REPEAT o thin_tac @{context} "ehoare ?x ?y") *})
apply (simp_all only: cnvalid1_eq cenvalid_def2)
      apply fast

```



```

      apply fast
      apply fast
      apply (clarify,tactic "smp_tac 1 1",erule(2) Loop_sound_lemma,(rule HOL.refl)+)
      apply fast
      apply fast
      apply fast
      apply fast
      apply fast
      apply fast
      apply (clarsimp del: Meth_elim_cases)
      apply (force del: Impl_elim_cases)
      defer
      prefer 4 apply blast
      prefer 4 apply blast
      apply (simp_all (no_asm_use) only: cnvalids_def nvalids_def)
      apply blast
      apply blast
      apply blast
      apply (rule allI)
      apply (rule_tac x=Z in spec)
      apply (induct_tac "n")
      apply (clarify intro!: Impl_nvalid_0)
      apply (clarify intro!: Impl_nvalid_Suc)
      apply (drule nvalids_SucD)
      apply (simp only: all_simps)
      apply (erule (1) impE)
      apply (drule (2) Impl_sound_lemma)
      apply blast
      apply assumption
    done

```

```

theorem hoare_sound: "{ } ⊢ {P} c {Q} ⇒ ⊨ {P} c {Q}"
apply (simp only: valid_def2)
apply (drule hoare_sound_main [THEN conjunct1, rule_format])
apply (unfold cnvalids_def nvalids_def)
apply fast
done

```

```

theorem ehoare_sound: "{ } ⊢e {P} e {Q} ⇒ ⊨e {P} e {Q}"
apply (simp only: evalid_def2)
apply (drule hoare_sound_main [THEN conjunct2, rule_format])
apply (unfold cenvalid_def nvalids_def)
apply fast
done

```

### 7.3 (Relative) Completeness

```

definition MGT :: "stmt => state => triple" where
  "MGT c Z ≡ (λs. Z = s, c, λ t. ∃n. Z -c- n → t)"

```

```

definition MGTe :: "expr => state => etriple" where
  "MGTe e Z ≡ (λs. Z = s, e, λv t. ∃n. Z -e>-v-n → t)"

```

```

notation (xsymbols)
  MGTe ("MGTe")
notation (HTML output)
  MGTe ("MGTe")

```

```

lemma MGF_implies_complete:

```

```

"∀Z. {} ⊢ { MGT c Z } ⇒ ⊢ {P} c {Q} ⇒ {} ⊢ {P} c {Q}"
apply (simp only: valid_def2)
apply (unfold MGT_def)
apply (erule hoare_ehoare.Conseq)
apply (clarsimp simp add: nvalid_def2)
done

lemma eMGF_implies_complete:
  "∀Z. {} ⊢e MGTe e Z ⇒ ⊢e {P} e {Q} ⇒ {} ⊢e {P} e {Q}"
apply (simp only: evalid_def2)
apply (unfold MGTe_def)
apply (erule hoare_ehoare.eConseq)
apply (clarsimp simp add: envalid_def2)
done

declare exec_eval.intros[intro!]

lemma MGF_Loop: "∀Z. A ⊢ {op = Z} c {λt. ∃n. Z -c-n→ t} ⇒
  A ⊢ {op = Z} While (x) c {λt. ∃n. Z -While (x) c-n→ t}"
apply (rule_tac P' = "λZ s. (Z,s) ∈ ({(s,t). ∃n. s<x> ≠ Null ∧ s -c-n→ t})^*"
  in hoare_ehoare.Conseq)
apply (rule allI)
apply (rule hoare_ehoare.Loop)
apply (erule hoare_ehoare.Conseq)
apply clarsimp
apply (blast intro:rtrancl_into_rtrancl)
apply (erule thin_rl)
apply clarsimp
apply (erule_tac x = Z in allE)
apply clarsimp
apply (erule converse_rtrancl_induct)
apply blast
apply clarsimp
apply (drule (1) exec_exec_max)
apply (blast del: exec_elim_cases)
done

lemma MGF_lemma: "∀M Z. A ⊢ {MGT (Impl M) Z} ⇒
  (∀Z. A ⊢ {MGT c Z}) ∧ (∀Z. A ⊢e MGTe e Z)"
apply (simp add: MGT_def MGTe_def)
apply (rule stmt_expr.induct)
apply (rule_tac [!] allI)

apply (rule Conseq1 [OF hoare_ehoare.Skip])
apply blast

apply (rule hoare_ehoare.Comp)
apply (erule spec)
apply (erule hoare_ehoare.Conseq)
apply clarsimp
apply (drule (1) exec_exec_max)
apply blast

apply (erule thin_rl)
apply (rule hoare_ehoare.Cond)
apply (erule spec)
apply (rule allI)
apply (simp)
apply (rule conjI)

```

```

apply (rule impI, erule hoare_ehoare.Conseq, clarsimp, drule (1) eval_exec_max,
      erule thin_rl, erule thin_rl, force)+

apply (erule MGF_Loop)

apply (erule hoare_ehoare.eConseq [THEN hoare_ehoare.LAss])
apply fast

apply (erule thin_rl)
apply (rule_tac Q = " $\lambda a s. \exists n. Z \text{ -expr1 } \succ \text{Addr } a \text{ -n} \rightarrow s$ " in hoare_ehoare.FAss)
apply (drule spec)
apply (erule eConseq2)
apply fast
apply (rule allI)
apply (erule hoare_ehoare.eConseq)
apply clarsimp
apply (drule (1) eval_eval_max)
apply blast

apply (simp only: split_paired_all)
apply (rule hoare_ehoare.Meth)
apply (rule allI)
apply (drule spec, drule spec, erule hoare_ehoare.Conseq)
apply blast

apply (simp add: split_paired_all)

apply (rule eConseq1 [OF hoare_ehoare.NewC])
apply blast

apply (erule hoare_ehoare.eConseq [THEN hoare_ehoare.Cast])
apply fast

apply (rule eConseq1 [OF hoare_ehoare.LAcc])
apply blast

apply (erule hoare_ehoare.eConseq [THEN hoare_ehoare.FAcc])
apply fast

apply (rule_tac R = " $\lambda a v s. \exists n1 n2 t. Z \text{ -expr1 } \succ a \text{ -n1} \rightarrow t \wedge t \text{ -expr2 } \succ v \text{ -n2} \rightarrow s$ " in
      hoare_ehoare.Call)
apply (erule spec)
apply (rule allI)
apply (erule hoare_ehoare.eConseq)
apply clarsimp
apply blast
apply (rule allI)+
apply (rule hoare_ehoare.Meth)
apply (rule allI)
apply (drule spec, drule spec, erule hoare_ehoare.Conseq)
apply (erule thin_rl, erule thin_rl)
apply (clarsimp del: Impl_elim_cases)
apply (drule (2) eval_eval_exec_max)
apply (force del: Impl_elim_cases)
done

lemma MGF_Impl: "{ } | $\vdash$  {MGT (Impl M) Z}"
apply (unfold MGT_def)
apply (rule Impl1')

```

```

apply (rule_tac [2] UNIV_I)
apply clarsimp
apply (rule hoare_ehoare.ConjI)
apply clarsimp
apply (rule ssubst [OF Impl_body_eq])
apply (fold MGT_def)
apply (rule MGF_lemma [THEN conjunct1, rule_format])
apply (rule hoare_ehoare.Asm)
apply force
done

theorem hoare_relative_complete: " $\models \{P\} c \{Q\} \implies \{\} \vdash \{P\} c \{Q\}$ "
apply (rule MGF_implies_complete)
apply (erule_tac [2] asm_rl)
apply (rule allI)
apply (rule MGF_lemma [THEN conjunct1, rule_format])
apply (rule MGF_Impl)
done

theorem ehoare_relative_complete: " $\models_e \{P\} e \{Q\} \implies \{\} \vdash_e \{P\} e \{Q\}$ "
apply (rule eMGF_implies_complete)
apply (erule_tac [2] asm_rl)
apply (rule allI)
apply (rule MGF_lemma [THEN conjunct2, rule_format])
apply (rule MGF_Impl)
done

lemma cFalse: " $A \vdash \{\lambda s. \text{False}\} c \{Q\}$ "
apply (rule cThin)
apply (rule hoare_relative_complete)
apply (auto simp add: valid_def)
done

lemma eFalse: " $A \vdash_e \{\lambda s. \text{False}\} e \{Q\}$ "
apply (rule eThin)
apply (rule ehoare_relative_complete)
apply (auto simp add: evalid_def)
done

end

```

## 8 Example

```
theory Example imports Equivalence begin
```

```

class Nat {

  Nat pred;

  Nat suc()
  { Nat n = new Nat(); n.pred = this; return n; }

  Nat eq(Nat n)
  { if (this.pred != null) if (n.pred != null) return this.pred.eq(n.pred);
    else return n.pred; // false
  }
}

```

```

    else if (n.pred != null) return this.pred; // false
    else return this.suc(); // true
}

Nat add(Nat n)
{ if (this.pred != null) return this.pred.add(n.suc()); else return n; }

public static void main(String[] args) // test x+1=1+x
{
    Nat one = new Nat().suc();
    Nat x    = new Nat().suc().suc().suc().suc();
    Nat ok = x.suc().eq(x.add(one));
    System.out.println(ok != null);
}
}

axioms This_neq_Par [simp]: "This ≠ Par"
    Res_neq_This [simp]: "Res ≠ This"

```

## 8.1 Program representation

```

consts N      :: cname ("Nat")
consts pred   :: fname
consts suc    :: mname
    add       :: mname
consts any    :: vname

```

### abbreviation

```

dummy :: expr ("<>")
where "<>" == LAcc any"

```

### abbreviation

```

one :: expr
where "one" == {Nat}new Nat..suc(<>)"

```

The following properties could be derived from a more complete program model, which we leave out for laziness.

```

axioms Nat_no_subclasses [simp]: "D ≤C Nat = (D=Nat)"

axioms method_Nat_add [simp]: "method Nat add = Some
(| par=Class Nat, res=Class Nat, lcl=[],
 bdy= If((LAcc This..pred))
      (Res := {Nat}(LAcc This..pred)..add({Nat}LAcc Par..suc(<>)))
      Else Res := LAcc Par |)"

axioms method_Nat_suc [simp]: "method Nat suc = Some
(| par=NT, res=Class Nat, lcl=[],
 bdy= Res := new Nat;; LAcc Res..pred := LAcc This |)"

axioms field_Nat [simp]: "field Nat = empty(pred↦Class Nat)"

lemma init_locs_Nat_add [simp]: "init_locs Nat add s = s"
by (simp add: init_locs_def init_vars_def)

lemma init_locs_Nat_suc [simp]: "init_locs Nat suc s = s"
by (simp add: init_locs_def init_vars_def)

```

```

lemma upd_obj_new_obj_Nat [simp]:
  "upd_obj a pred v (new_obj a Nat s) = hupd(a↦(Nat, empty(pred↦v))) s"
by (simp add: new_obj_def init_vars_def upd_obj_def Let_def)

```

## 8.2 “atleast” relation for interpretation of Nat “values”

```

consts Nat_atleast :: "state ⇒ val ⇒ nat ⇒ bool" ("_: _ ≥ _" [51, 51, 51] 50)
primrec "s:x≥0      = (x≠Null)"
      "s:x≥Suc n = (∃ a. x=Addr a ∧ heap s a ≠ None ∧ s:get_field s a pred≥n)"

```

```

lemma Nat_atleast_lupd [rule_format, simp]:
  "∀ s v::val. lupd(x↦y) s:v ≥ n = (s:v ≥ n)"
apply (induct n)
by auto

```

```

lemma Nat_atleast_set_locs [rule_format, simp]:
  "∀ s v::val. set_locs l s:v ≥ n = (s:v ≥ n)"
apply (induct n)
by auto

```

```

lemma Nat_atleast_del_locs [rule_format, simp]:
  "∀ s v::val. del_locs s:v ≥ n = (s:v ≥ n)"
apply (induct n)
by auto

```

```

lemma Nat_atleast_NullD [rule_format]: "s:Null ≥ n ⟶ False"
apply (induct n)
by auto

```

```

lemma Nat_atleast_pred_NullD [rule_format]:
  "Null = get_field s a pred ⟹ s:Addr a ≥ n ⟶ n = 0"
apply (induct n)
by (auto dest: Nat_atleast_NullD)

```

```

lemma Nat_atleast_mono [rule_format]:
  "∀ a. s:get_field s a pred ≥ n ⟶ heap s a ≠ None ⟶ s:Addr a ≥ n"
apply (induct n)
by auto

```

```

lemma Nat_atleast_newC [rule_format]:
  "heap s aa = None ⟹ ∀ v::val. s:v ≥ n ⟶ hupd(aa↦obj) s:v ≥ n"
apply (induct n)
apply auto
apply (case_tac "aa=a")
apply auto
apply (tactic "smp_tac 1 1")
apply (case_tac "aa=a")
apply auto
done

```

## 8.3 Proof(s) using the Hoare logic

```

theorem add_homomorph_lb:
  "{ } ⊢ { λs. s:s<This> ≥ X ∧ s:s<Par> ≥ Y } Meth(Nat,add) { λs. s:s<Res> ≥ X+Y }"
apply (rule hoare_ehoare.Meth)
apply clarsimp
apply (rule_tac P' = "λZ s. (s:s<This> ≥ fst Z ∧ s:s<Par> ≥ snd Z) ∧ D=Nat" and
  Q' = "λZ s. s:s<Res> ≥ fst Z+snd Z" in AxSem.Conseq)

```

```

prefer 2
apply (clarsimp simp add: init_locs_def init_vars_def)
apply rule
apply (case_tac "D = Nat", simp_all, rule_tac [2] cFalse)
apply (rule_tac P = " $\lambda Z \text{ Cm } s. s:s\langle\text{This}\rangle \geq \text{fst } Z \wedge s:s\langle\text{Par}\rangle \geq \text{snd } Z$ " in AxSem.Impl1)
apply (clarsimp simp add: body_def)
apply (rename_tac n m)
apply (rule_tac Q = " $\lambda v \text{ s}. (s:s\langle\text{This}\rangle \geq n \wedge s:s\langle\text{Par}\rangle \geq m) \wedge$   

 $(\exists a. s\langle\text{This}\rangle = \text{Addr } a \wedge v = \text{get\_field } s \text{ a pred})$ " in hoare_ehoare.Cond)
apply (rule hoare_ehoare.FAcc)
apply (rule eConseq1)
apply (rule hoare_ehoare.LAcc)
apply fast
apply auto
prefer 2
apply (rule hoare_ehoare.LAss)
apply (rule eConseq1)
apply (rule hoare_ehoare.LAcc)
apply (auto dest: Nat_atleast_pred_NullD)
apply (rule hoare_ehoare.LAss)
apply (rule_tac
  Q = " $\lambda v \text{ s}. (\forall m. n = \text{Suc } m \longrightarrow s:v \geq m) \wedge s:s\langle\text{Par}\rangle \geq m$ " and
  R = " $\lambda T \text{ P } s. (\forall m. n = \text{Suc } m \longrightarrow s:T \geq m) \wedge s:P \geq \text{Suc } m$ "
  in hoare_ehoare.Call)
apply (rule hoare_ehoare.FAcc)
apply (rule eConseq1)
apply (rule hoare_ehoare.LAcc)
apply clarify
apply (drule sym, rotate_tac -1, frule (1) trans)
apply simp
prefer 2
apply clarsimp
apply (rule hoare_ehoare.Meth)
apply clarsimp
apply (case_tac "D = Nat", simp_all, rule_tac [2] cFalse)
apply (rule AxSem.Conseq)
apply rule
apply (rule hoare_ehoare.Asm)
apply (rule_tac a = " $((\text{case } n \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m \Rightarrow m), m+1)$ " in UN_I, rule+)
apply (clarsimp split add: nat.split_asm dest!: Nat_atleast_mono)
apply rule
apply (rule hoare_ehoare.Call)
apply (rule hoare_ehoare.LAcc)
apply rule
apply (rule hoare_ehoare.LAcc)
apply clarify
apply (rule hoare_ehoare.Meth)
apply clarsimp
apply (case_tac "D = Nat", simp_all, rule_tac [2] cFalse)
apply (rule AxSem.Impl1)
apply (clarsimp simp add: body_def)
apply (rule hoare_ehoare.Comp)
prefer 2
apply (rule hoare_ehoare.FAss)
prefer 2
apply rule
apply (rule hoare_ehoare.LAcc)
apply (rule hoare_ehoare.LAcc)
apply (rule hoare_ehoare.LAss)

```

```
apply (rule eConseq1)
apply (rule hoare_ehoare.NewC)
apply (auto dest!: new_AddrD elim: Nat_atleast_newC)
done
```

```
end
```



## References

- [1] T. Nipkow, D. v. Oheimb, and C. Pusch.  $\mu$ Java: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.
- [2] D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited, 2002. Submitted for publication.
- [3] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience*, 598:??–??+43, 2001. <http://isabelle.in.tum.de/Bali/papers/CPE01.html>, to appear.