

# NanoJava

David von Oheimb

Tobias Nipkow

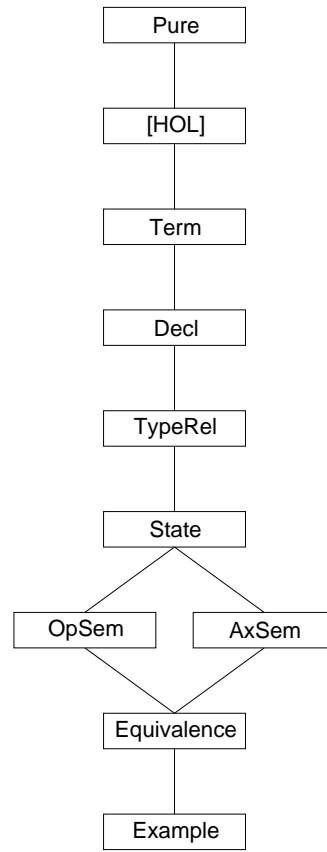
April 19, 2009

## 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>6</b>
4.1	Properties not used in the meta theory . . . . .	8
<b>5</b>	<b>Operational Evaluation Semantics</b>	<b>9</b>
<b>6</b>	<b>Axiomatic Semantics</b>	<b>11</b>
6.1	Hoare Logic Rules . . . . .	11
6.2	Fully polymorphic variants, required for Example only . . . . .	12
6.3	Derived Rules . . . . .	13
<b>7</b>	<b>Equivalence of Operational and Axiomatic Semantics</b>	<b>13</b>
7.1	Validity . . . . .	13
7.2	Soundness . . . . .	14
7.3	(Relative) Completeness . . . . .	15
<b>8</b>	<b>Example</b>	<b>16</b>
8.1	Program representation . . . . .	17
8.2	“atleast” relation for interpretation of Nat “values” . . . . .	17
8.3	Proof(s) using the Hoare logic . . . . .	18



## 1 Statements and expression emulations

theory *Term* imports *Main* begin

typedekl *cname* — class name  
 typedekl *mname* — method name  
 typedekl *fname* — field name  
 typedekl *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

```

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

```

consts

```

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

```

constdefs

```

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

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

```

```

lemma finite_is_class: "finite {C. is_class C}"
<proof>

```

end

### 3 Type relations

theory TypeRel imports Decl begin

consts

```

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

```

syntax (xsymbols)

```

subcls1 :: "[cname, cname] => bool" ("_ <C1 _" [71,71] 70)
subcls  :: "[cname, cname] => bool" ("_ <=C _" [71,71] 70)

```

syntax

```

subcls1 :: "[cname, cname] => bool" ("_ <=C1 _" [71,71] 70)
subcls  :: "[cname, cname] => bool" ("_ <=C _" [71,71] 70)

```

translations

```

"C <C1 D" == "(C,D) ∈ subcls1"
"C <=C D" == "(C,D) ∈ subcls1^*"

```

consts

```

method :: "cname => (mname → 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 $\neq$ Object  $\wedge$  ( $\exists$  c. class C = Some c  $\wedge$  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. class C = Some c  $\wedge$  super c=D)"
<proof>
```

```
lemma subcls1I: "[class C = Some m; super m = D; C  $\neq$  Object]  $\implies$  C $\prec$ C1D"
<proof>
```

**lemma subcls1\_def2:**

```
"subcls1 =
  (SIGMA C: {C. is_class C} . {D. C $\neq$ Object  $\wedge$  super (the (class C)) = D})"
<proof>
```

**lemma finite\_subcls1:** "finite subcls1"

<proof>

**constdefs**

```
ws_prog :: "bool"
"ws_prog  $\equiv$   $\forall$  (C,c) $\in$ set Prog. C $\neq$ 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 $^+$ "
<proof>
```

```
lemma subcls1_irrefl_lemma1: "ws_prog  $\implies$  subcls1 $^{-1} \cap$  subcls1 $^+ = \{\}$ "
<proof>
```

```
lemma irrefl_trancII': "r $^{-1}$  Int r $^+ = \{\} \implies$  !x. (x, x)  $\sim$ : r $^+$ "
<proof>
```

```
lemmas subcls1_irrefl_lemma2 = subcls1_irrefl_lemma1 [THEN irrefl_trancII']
```

```
lemma subcls1_irrefl: "[ (x, y)  $\in$  subcls1; ws_prog ]  $\implies$  x  $\neq$  y"
<proof>
```

```
lemmas subcls1_acyclic = subcls1_irrefl_lemma2 [THEN acyclicI, standard]
```

```
lemma wf_subcls1: "ws_prog  $\implies$  wf (subcls1 $^{-1}$ )"
<proof>
```

```

consts class_rec :: "cname  $\Rightarrow$  (class  $\Rightarrow$  ('a  $\times$  'b) list)  $\Rightarrow$  ('a  $\rightarrow$  'b)"

recdef (permissive) class_rec "subcls1-1"
  "class_rec C = ( $\lambda$ f. case class C of None  $\Rightarrow$  undefined
    | Some m  $\Rightarrow$  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]  $\Rightarrow$ 
  class_rec C f = (if C = Object then empty else class_rec (super m) f) ++
    map_of (f m)"
  <proof>
defs method_def: "method C  $\equiv$  class_rec C methods"

lemma method_rec: "[class C = Some m; ws_prog]  $\Rightarrow$ 
  method C = (if C=Object then empty else method (super m)) ++ map_of (methods m)"
  <proof>
defs field_def: "field C  $\equiv$  class_rec C flds"

lemma flds_rec: "[class C = Some m; ws_prog]  $\Rightarrow$ 
  field C = (if C=Object then empty else field (super m)) ++ map_of (flds m)"
  <proof>

end

```

## 4 Program State

```

theory State imports TypeRel begin

```

```

constdefs

```

```

  body :: "cname  $\times$  mname  $\Rightarrow$  stmt"
  "body  $\equiv \lambda$ (C,m). bdy (the (method C m))"

```

Locations, i.e. abstract references to objects

```

typedec1 loc

```

```

datatype val

```

```

  = Null          — null reference
  | Addr loc      — address, i.e. location of object

```

```

types   fields
  = "(fname  $\rightarrow$  val)"

```

```

  obj = "cname  $\times$  fields"

```

```

translations

```

```

  "fields"  $\leftarrow$  (type)"fname  $\Rightarrow$  val option"
  "obj"      $\leftarrow$  (type)"cname  $\times$  fields"

```

```

constdefs

```

```

  init_vars:: "('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  val)"

```

```

"init_vars m == Option.map (λT. Null) o m"

private:
types   heap   = "loc   ↦ obj"
        locals = "vname ↦ val"

private:
record state
  = heap   :: heap
    locals :: locals

translations

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

```

#### constdefs

```

del_locs      :: "state => state"
"del_locs s ≡ s (| locals := empty |)"

init_locs     :: "cname => mname => state => state"
"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.

#### constdefs

```

set_locs     :: "state => state => state"
"set_locs s s' ≡ s' (| locals := locals s |)"

get_local    :: "state => vname => val" ("_<_" [99,0] 99)
"get_local s x ≡ the (locals s x)"

```

— local function:

```

get_obj      :: "state => loc => obj"
"get_obj s a ≡ the (heap s a)"

obj_class    :: "state => loc => cname"
"obj_class s a ≡ fst (get_obj s a)"

get_field    :: "state => loc => fname => val"
"get_field s a f ≡ the (snd (get_obj s a) f)"

```

— local function:

```

hupd         :: "loc => obj => state => state" ("hupd'(_|->_)" [10,10] 1000)
"hupd a obj s ≡ s (| heap := ((heap s)(a↦obj))|)"

lupd         :: "vname => val => state => state" ("lupd'(_|->_)" [10,10] 1000)
"lupd x v s   ≡ s (| locals := ((locals s)(x↦v ))|)"

```

#### syntax (xsymbols)

```

hupd         :: "loc => obj => state => state" ("hupd'(_↦_)" [10,10] 1000)
lupd         :: "vname => val => state => state" ("lupd'(_↦_)" [10,10] 1000)

```

#### constdefs

```

new_obj      :: "loc => cname => state => state"

```

```

"new_obj a C  ≡ hupd(a↦(C,init_vars (field C)))"

upd_obj      :: "loc => fname => val => state => state"
"upd_obj a f v s ≡ let (C,fs) = the (heap s a) in hupd(a↦(C,fs(f↦v))) s"

new_Addr     :: "state => val"
"new_Addr s == SOME v. (∃ a. v = Addr a ∧ (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"
⟨proof⟩

lemma lupd_get_local_same [simp]: "lupd(x↦v) s<x> = v"
⟨proof⟩

lemma lupd_get_local_other [simp]: "x ≠ y ⟹ lupd(x↦v) s<y> = s<y>"
⟨proof⟩

lemma get_field_lupd [simp]:
  "get_field (lupd(x↦y) s) a f = get_field s a f"
⟨proof⟩

lemma get_field_set_locs [simp]:
  "get_field (set_locs l s) a f = get_field s a f"
⟨proof⟩

lemma get_field_del_locs [simp]:
  "get_field (del_locs s) a f = get_field s a f"
⟨proof⟩

lemma new_obj_get_local [simp]: "new_obj a C s <x> = s<x>"
⟨proof⟩

lemma heap_lupd [simp]: "heap (lupd(x↦y) s) = heap s"
⟨proof⟩

lemma heap_hupd_same [simp]: "heap (hupd(a↦obj) s) a = Some obj"
⟨proof⟩

lemma heap_hupd_other [simp]: "aa ≠ a ⟹ heap (hupd(aa↦obj) s) a = heap s a"
⟨proof⟩

lemma hupd_hupd [simp]: "hupd(a↦obj) (hupd(a↦obj') s) = hupd(a↦obj) s"
⟨proof⟩

lemma heap_del_locs [simp]: "heap (del_locs s) = heap s"
⟨proof⟩

lemma heap_set_locs [simp]: "heap (set_locs l s) = heap s"
⟨proof⟩

lemma hupd_lupd [simp]:
  "hupd(a↦obj) (lupd(x↦y) s) = lupd(x↦y) (hupd(a↦obj) s)"
⟨proof⟩

lemma hupd_del_locs [simp]:
  "hupd(a↦obj) (del_locs s) = del_locs (hupd(a↦obj) s)"
⟨proof⟩

```



```

lemma new_obj_lupd [simp]:
  "new_obj a C (lupd(x↦y) s) = lupd(x↦y) (new_obj a C s)"
  <proof>

lemma new_obj_del_locs [simp]:
  "new_obj a C (del_locs s) = del_locs (new_obj a C s)"
  <proof>

lemma upd_obj_lupd [simp]:
  "upd_obj a f v (lupd(x↦y) s) = lupd(x↦y) (upd_obj a f v s)"
  <proof>

lemma upd_obj_del_locs [simp]:
  "upd_obj a f v (del_locs s) = del_locs (upd_obj a f v s)"
  <proof>

lemma get_field_hupd_same [simp]:
  "get_field (hupd(a↦(C, fs)) s) a = the ∘ fs"
  <proof>

lemma get_field_hupd_other [simp]:
  "aa ≠ a ⇒ get_field (hupd(aa↦obj) s) a = get_field s a"
  <proof>

lemma new_AddrD:
  "new_Addr s = v ⇒ (∃ a. v = Addr a ∧ heap s a = None) | v = Null"
  <proof>

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))"

<proof>

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"

<proof>

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"

<proof>

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"

$\langle proof \rangle$

lemma eval\_eval\_exec\_max:

" $\llbracket s1 \text{ -}e1 \succ v1 \text{ -}n1 \rightarrow t1; s2 \text{ -}e2 \succ v2 \text{ -}n2 \rightarrow t2; s3 \text{ -}c \text{ -}n3 \rightarrow t3 \rrbracket \implies$   
 $s1 \text{ -}e1 \succ v1 \text{ -}max \ (max \ n1 \ n2) \ n3 \rightarrow t1 \ \wedge$   
 $s2 \text{ -}e2 \succ v2 \text{ -}max \ (max \ n1 \ n2) \ n3 \rightarrow t2 \ \wedge$   
 $s3 \text{ -}c \text{ -}max \ (max \ n1 \ n2) \ n3 \rightarrow t3$ "

$\langle proof \rangle$

lemma Impl\_body\_eq: " $(\lambda t. \exists n. Z \text{ -}Impl \ M \text{ -}n \rightarrow t) = (\lambda t. \exists n. Z \text{ -}body \ M \text{ -}n \rightarrow t)$ "

$\langle proof \rangle$

end

## 6 Axiomatic Semantics

theory AxSem imports State begin

types assn = "state => bool"

vassn = "val => assn"

triple = "assn  $\times$  stmt  $\times$  assn"

etriples = "assn  $\times$  expr  $\times$  vassn"

translations

"assn"  $\leftarrow$  (type)"state => bool"

"vassn"  $\leftarrow$  (type)"val => assn"

"triple"  $\leftarrow$  (type)"assn  $\times$  stmt  $\times$  assn"

"etriples"  $\leftarrow$  (type)"assn  $\times$  expr  $\times$  vassn"

### 6.1 Hoare Logic Rules

inductive

hoare :: "[triple set, triple set] => bool" ("\_  $\vdash$  / \_" [61, 61] 60)

and ehoare :: "[triple set, etriples] => bool" ("\_  $\vdash_e$  / \_" [61, 61] 60)

and hoare1 :: "[triple set, assn,stmt,assn] => bool"

("\_  $\vdash$  / ({(1\_)} / (\_) / {(1\_)} )" [61, 3, 90, 3] 60)

and ehoare1 :: "[triple set, assn,expr,vassn] => 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} |]  $\implies$  A  $\vdash$  {P} c1;;c2 {R}"

| Cond: "[| A  $\vdash_e$  {P} e {Q};

$\forall v. A \vdash \{Q \ v\} \ (if \ v \neq \text{Null} \ \text{then} \ c1 \ \text{else} \ c2) \ \{R\} \ |] \implies$

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}  $\implies$

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 { $\lambda v \ s. Q \ (\text{lupd}(x \mapsto v) \ s)$ }  $\implies$

A  $\vdash$  {P} x ::= e {Q}"

```

| FAcc: "A ⊢e {P} e {λv s. ∀a. v=Addr a --> Q (get_field s a f) s} ==>
  A ⊢e {P} e..f {Q}"

| FAss: "[| A ⊢e {P} e1 {λv s. ∀a. v=Addr a --> Q a s};
  ∀a. A ⊢e {Q a} e2 {λv s. R (upd_obj a f v s)} |] ==>
  A ⊢ {P} e1..f==e2 {R}"

| NewC: "A ⊢e {λs. ∀a. new_Addr s = Addr a --> P (Addr a) (new_obj a C s)}
  new C {P}"

| Cast: "A ⊢e {P} e {λv s. (case v of Null => True
  | Addr a => obj_class s a <=C C) --> Q v s} ==>
  A ⊢e {P} Cast C e {Q}"

| Call: "[| A ⊢e {P} e1 {Q}; ∀a. A ⊢e {Q a} e2 {R a};
  ∀a p ls. A ⊢ {λs'. ∃s. R a p s ∧ ls = s ∧
    s' = lupd(This↦a)(lupd(Par↦p)(del_locs s))}
  Meth (C,m) {λs. S (s<Res>) (set_locs ls s)} |] ==>
  A ⊢e {P} {C}e1..m(e2) {S}"

| Meth: "∀D. A ⊢ {λs'. ∃s a. s<This> = Addr a ∧ D = obj_class s a ∧ D <=C C ∧
  P s ∧ s' = init_locs D m s}
  Impl (D,m) {Q} ==>
  A ⊢ {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) |] ==>

```

$$A \vdash_e \{P\} e \{Q\}"$$

```

Impl: "∀ Z. A ∪ (⋃ Z. (λ Cm. (P Z Cm, Impl Cm, Q Z Cm)) 'Ms) ⊢
      (λ Cm. (P Z Cm, body Cm, Q Z Cm)) 'Ms ==>
      A ⊢ (λ Cm. (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}"
<proof>

```

```

lemma Conseq2: "⊢ A ⊢ {P} c {Q'}; ∀ t. Q' t → Q t ⊢ A ⊢ {P} c {Q}"
<proof>

```

```

lemma eConseq1: "⊢ A ⊢_e {P'} e {Q'}; ∀ s. P s → P' s ⊢ A ⊢_e {P} e {Q}"
<proof>

```

```

lemma eConseq2: "⊢ A ⊢_e {P} e {Q'}; ∀ v t. Q' v t → Q v t ⊢ A ⊢_e {P} e {Q}"
<proof>

```

```

lemma Weaken: "⊢ A ⊢ C'; C ⊆ C' ⊢ A ⊢ C"
<proof>

```

```

lemma Thin_lemma:
  "(A' ⊢ C → (∀ A. A' ⊆ A → A ⊢ C)) ∧
  (A' ⊢_e {P} e {Q} → (∀ A. A' ⊆ A → A ⊢_e {P} e {Q}))"
<proof>

```

```

lemma cThin: "⊢ A' ⊢ C; A' ⊆ A ⊢ A ⊢ C"
<proof>

```

```

lemma eThin: "⊢ A' ⊢_e {P} e {Q}; A' ⊆ A ⊢ A ⊢_e {P} e {Q}"
<proof>

```

```

lemma Union: "A ⊢ (⋃ Z. C Z) = (∀ Z. A ⊢ C Z)"
<proof>

```

```

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}"
<proof>

```

```

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

```

constdefs
  valid :: "[assn, stmt, assn] => bool" ("|= {(1_)} / ( _ ) / {(1_)}" [3,90,3] 60)
  "|= {P} c {Q} ≡ ∀ s t. P s --> (∃ n. s -c -n → t) --> Q t"

```

```

evalid    :: "[assn,expr,vassn] => bool" ("|=e {(1_)} / (_)/ {(1_)}" [3,90,3] 60)
"|=e {P} e {Q} ≡ ∀ s v t. P s --> (∃ n. s -e>v-n→ t) --> Q v t"

nvalid    :: "[nat, triple    ] => bool" ("|=_: _" [61,61] 60)
"|=n: t ≡ let (P,c,Q) = t in ∀ s    t. s -c    -n→ t --> P s --> Q    t"

envalid    :: "[nat,etriples    ] => bool" ("|=e: _" [61,61] 60)
"|=n:e t ≡ let (P,e,Q) = t in ∀ s v t. s -e>v-n→ t --> P s --> Q v t"

nvalids   :: "[nat,          triple set] => bool" ("||=_: _" [61,61] 60)
"||=n: T ≡ ∀ t∈T. |=n: t"

cnvalids   :: "[triple set,triple set] => bool" ("_ ||=/ _" [61,61] 60)
"A ||= C ≡ ∀ n. ||=n: A --> ||=n: C"

cenvalid   :: "[triple set,etriples    ] => bool" ("_ ||=e/ _" [61,61] 60)
"A ||=e t ≡ ∀ n. ||=n: A --> |=n:e t"

syntax (xsymbols)
  valid    :: "[assn,stmt, assn] => bool" ( "|= {(1_)} / (_)/ {(1_)}" [3,90,3] 60)
  evalid   :: "[assn,expr,vassn] => bool" ("|=e {(1_)} / (_)/ {(1_)}" [3,90,3] 60)
  nvalid    :: "[nat, triple    ] => bool" ("|=_: _" [61,61] 60)
  envalid   :: "[nat,etriples    ] => bool" ("|=e: _" [61,61] 60)
  nvalids   :: "[nat,          triple set] => bool" ("||=_: _" [61,61] 60)
  cnvalids   :: "[triple set,triple set] => bool" ("_ ||=/ _" [61,61] 60)
  cenvalid   :: "[triple set,etriples    ] => bool" ("_ ||=e/ _" [61,61] 60)

lemma nvalid_def2: "|=n: (P,c,Q) ≡ ∀ s t. s -c-n→ t → P s → Q t"
<proof>

lemma valid_def2: "|= {P} c {Q} = (∀ n. |=n: (P,c,Q))"
<proof>

lemma envalid_def2: "|=n:e (P,e,Q) ≡ ∀ s v t. s -e>v-n→ t → P s → Q v t"
<proof>

lemma evalid_def2: "|=e {P} e {Q} = (∀ n. |=n:e (P,e,Q))"
<proof>

lemma cenvalid_def2:
  "A ||=e (P,e,Q) = (∀ n. ||=n: A → (∀ s v t. s -e>v-n→ t → P s → Q v t))"
<proof>



## 7.2 Soundness



declare exec_elim_cases [elim!] eval_elim_cases [elim!]

lemma Impl_nvalid_0: "|=0: (P,Impl M,Q)"
<proof>

lemma Impl_nvalid_Suc: "|=n: (P,body M,Q) ⇒ |=Suc n: (P,Impl M,Q)"
<proof>

lemma nvalid_SucD: "∧ t. |=Suc n:t ⇒ |=n:t"
<proof>

```

**lemma** *nvalids\_SucD*: "Ball A (nvalid (Suc n))  $\implies$  Ball A (nvalid n)"  
 <proof>

**lemma** *Loop\_sound\_lemma* [rule\_format (no\_asm)]:  
 " $\forall s\ t. s \text{ -c-n} \rightarrow t \rightarrow P\ s \wedge s\langle x \rangle \neq \text{Null} \rightarrow P\ t \implies$   
 ( $s \text{ -c0-n0} \rightarrow t \rightarrow P\ s \rightarrow c0 = \text{While}\ (x)\ c \rightarrow n0 = n \rightarrow P\ t \wedge t\langle x \rangle = \text{Null}$ )"

**lemma** *Impl\_sound\_lemma*:  
 " $\llbracket \forall z\ n. \text{Ball}\ (A \cup B)\ (\text{nvalid}\ n) \rightarrow \text{Ball}\ (f\ z\ 'Ms)\ (\text{nvalid}\ n);$   
 $Cm \in Ms; \text{Ball}\ A\ (\text{nvalid}\ na); \text{Ball}\ B\ (\text{nvalid}\ na) \rrbracket \implies \text{nvalid}\ na\ (f\ z\ Cm)$ "  
 <proof>

**lemma** *all\_conjunct2*: " $\forall l. P'\ l \wedge P\ l \implies \forall l. P\ l$ "  
 <proof>

**lemma** *all3\_conjunct2*:  
 " $\forall a\ p\ l. (P'\ a\ p\ l \wedge P\ a\ p\ l) \implies \forall a\ p\ l. P\ a\ p\ l$ "  
 <proof>

**lemma** *cnvalid1\_eq*:  
 " $A \models \{(P, c, Q)\} \equiv \forall n. \models n: A \rightarrow (\forall s\ t. s \text{ -c-n} \rightarrow t \rightarrow P\ s \rightarrow Q\ t)$ "  
 <proof>

**lemma** *hoare\_sound\_main*: " $\bigwedge t. (A \vdash C \rightarrow A \models C) \wedge (A \vdash_e t \rightarrow A \models_e t)$ "  
 <proof>

**theorem** *hoare\_sound*: " $\{\} \vdash \{P\}\ c\ \{Q\} \implies \models \{P\}\ c\ \{Q\}$ "  
 <proof>

**theorem** *ehoare\_sound*: " $\{\} \vdash_e \{P\}\ e\ \{Q\} \implies \models_e \{P\}\ e\ \{Q\}$ "  
 <proof>

### 7.3 (Relative) Completeness

**constdefs** *MGT* :: "stmt  $\Rightarrow$  state  $\Rightarrow$  triple"  
 "*MGT* c Z  $\equiv (\lambda s. Z = s, c, \lambda t. \exists n. Z \text{ -c- } n \rightarrow t)$ "  
*MGT<sub>e</sub>* :: "expr  $\Rightarrow$  state  $\Rightarrow$  etriple"  
 "*MGT<sub>e</sub>* e Z  $\equiv (\lambda s. Z = s, e, \lambda v\ t. \exists n. Z \text{ -e>-v-n} \rightarrow t)$ "  
**syntax** (xsymbols)  
*MGT<sub>e</sub>* :: "expr  $\Rightarrow$  state  $\Rightarrow$  etriple" ("MGT<sub>e</sub>")  
**syntax** (HTML output)  
*MGT<sub>e</sub>* :: "expr  $\Rightarrow$  state  $\Rightarrow$  etriple" ("MGT<sub>e</sub>")

**lemma** *MGF\_implies\_complete*:  
 " $\forall Z. \{\} \vdash \{MGT\ c\ Z\} \implies \models \{P\}\ c\ \{Q\} \implies \{\} \vdash \{P\}\ c\ \{Q\}$ "  
 <proof>

**lemma** *eMGF\_implies\_complete*:  
 " $\forall Z. \{\} \vdash_e MGT_e\ e\ Z \implies \models_e \{P\}\ e\ \{Q\} \implies \{\} \vdash_e \{P\}\ e\ \{Q\}$ "  
 <proof>

**declare** *exec\_eval.intros*[intro!]

**lemma** *MGF\_Loop*: " $\forall Z. A \vdash \{\text{op} = Z\}\ c\ \{\lambda t. \exists n. Z \text{ -c-n} \rightarrow t\} \implies$   
 $A \vdash \{\text{op} = Z\}\ \text{While}\ (x)\ c\ \{\lambda t. \exists n. Z \text{ -While}\ (x)\ c\text{-n} \rightarrow t\}$ "  
 <proof>

**lemma** *MGF\_lemma*: " $\forall M\ Z. A \vdash \{MGT\ (Impl\ M)\ Z\} \implies$

$(\forall Z. A \vdash \{MGT\ c\ Z\}) \wedge (\forall Z. A \vdash_e MGT_e\ e\ Z)$ "  
 $\langle proof \rangle$

**lemma** *MGF\_Impl*: " $\{\} \vdash \{MGT\ (Impl\ M)\ Z\}$ "  
 $\langle proof \rangle$

**theorem** *hoare\_relative\_complete*: " $\models \{P\}\ c\ \{Q\} \implies \{\} \vdash \{P\}\ c\ \{Q\}$ "  
 $\langle proof \rangle$

**theorem** *ehoare\_relative\_complete*: " $\models_e \{P\}\ e\ \{Q\} \implies \{\} \vdash_e \{P\}\ e\ \{Q\}$ "  
 $\langle proof \rangle$

**lemma** *cFalse*: " $A \vdash \{\lambda s. False\}\ c\ \{Q\}$ "  
 $\langle proof \rangle$

**lemma** *eFalse*: " $A \vdash_e \{\lambda s. False\}\ e\ \{Q\}$ "  
 $\langle proof \rangle$

**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*  $\neq$  *Par*"  
       *Res\_neq\_This* [*simp*]: "*Res*  $\neq$  *This*"



## 8.1 Program representation

```

consts N      :: cname ("Nat")
consts pred   :: fname
consts suc    :: mname
      add     :: mname
consts any    :: vname
syntax dummy:: expr ("<>")
      one     :: expr
translations
  "<>" == "LAcc any"
  "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  $\preceq_C$  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 $\mapsto$ Class Nat)"

lemma init_locs_Nat_add [simp]: "init_locs Nat add s = s"
  <proof>

lemma init_locs_Nat_suc [simp]: "init_locs Nat suc s = s"
  <proof>

lemma upd_obj_new_obj_Nat [simp]:
  "upd_obj a pred v (new_obj a Nat s) = hupd(a $\mapsto$ (Nat, empty(pred $\mapsto$ v))) s"
  <proof>

```

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

```

consts Nat_atleast :: "state  $\Rightarrow$  val  $\Rightarrow$  nat  $\Rightarrow$  bool" ("_:_  $\geq$  _" [51, 51, 51] 50)
primrec "s:x $\geq$ 0      = (x $\neq$ Null)"
      "s:x $\geq$ Suc n = ( $\exists$  a. x=Addr a  $\wedge$  heap s a  $\neq$  None  $\wedge$  s:get_field s a pred $\geq$ n)"

lemma Nat_atleast_lupd [rule_format, simp]:
  " $\forall$  s v::val. lupd(x $\mapsto$ y) s:v  $\geq$  n = (s:v  $\geq$  n)"
  <proof>

lemma Nat_atleast_set_locs [rule_format, simp]:
  " $\forall$  s v::val. set_locs l s:v  $\geq$  n = (s:v  $\geq$  n)"
  <proof>

lemma Nat_atleast_del_locs [rule_format, simp]:
  " $\forall$  s v::val. del_locs s:v  $\geq$  n = (s:v  $\geq$  n)"
  <proof>

lemma Nat_atleast_NullD [rule_format]: "s:Null  $\geq$  n  $\longrightarrow$  False"
  <proof>

```

```

lemma Nat_atleast_pred_NullD [rule_format]:
  "Null = get_field s a pred  $\implies$  s:Addr a  $\geq$  n  $\longrightarrow$  n = 0"
  <proof>

lemma Nat_atleast_mono [rule_format]:
  " $\forall$  a. s:get_field s a pred  $\geq$  n  $\longrightarrow$  heap s a  $\neq$  None  $\longrightarrow$  s:Addr a  $\geq$  n"
  <proof>

lemma Nat_atleast_newC [rule_format]:
  "heap s aa = None  $\implies \forall$  v::val. s:v  $\geq$  n  $\longrightarrow$  hupd(aa $\mapsto$ obj) s:v  $\geq$  n"
  <proof>

```

### 8.3 Proof(s) using the Hoare logic

```

theorem add_homomorph_lb:
  "{ }  $\vdash$  {  $\lambda$ s. s:s<This>  $\geq$  X  $\wedge$  s:s<Par>  $\geq$  Y } Meth(Nat,add) {  $\lambda$ s. s:s<Res>  $\geq$  X+Y }"
  <proof>

```

**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.