

NanoJava

David von Oheimb

Tobias Nipkow

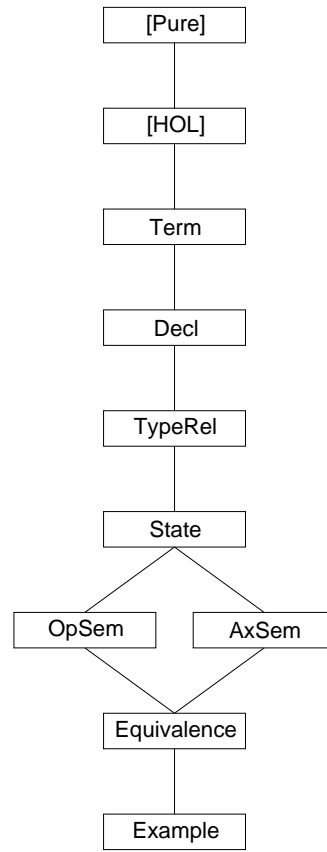
June 8, 2008

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

1	Statements and expression emulations	3
2	Types, class Declarations, and whole programs	3
3	Type relations	4
3.1	Declarations and properties not used in the meta theory	5
4	Program State	7
4.1	Properties not used in the meta theory	8
5	Operational Evaluation Semantics	10
6	Axiomatic Semantics	12
6.1	Hoare Logic Rules	12
6.2	Fully polymorphic variants, required for Example only	13
6.3	Derived Rules	13
7	Equivalence of Operational and Axiomatic Semantics	15
7.1	Validity	15
7.2	Soundness	16
7.3	(Relative) Completeness	18
8	Example	21
8.1	Program representation	21
8.2	“atleast” relation for interpretation of Nat “values”	22
8.3	Proof(s) using the Hoare logic	23



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}"

```

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

```

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 → ty)"
```

3.1 Declarations and properties not used in the meta theory

Direct subclass relation

defs

```
subcls1_def: "subcls1 ≡ {(C,D). C≠Object ∧ (∃c. class C = Some c ∧ super c=D)}"
```

Widening, viz. method invocation conversion

inductive

```
widen :: "ty => ty => bool"  ("_ ≤_" [71,71] 70)
```

where

```
refl [intro!, simp]: "T ≤ T"
| subcls: "C ≤C D ⇒ Class C ≤ Class D"
| null [intro!]: "NT ≤ R"
```

lemma subcls1D:

```
"C < C1D ⇒ C ≠ Object ∧ (∃c. class C = Some c ∧ super c=D)"
```

apply (unfold subcls1_def)

apply auto

done

lemma subcls1I: "[class C = Some m; super m = D; C ≠ Object] ⇒ C < C1D"

apply (unfold subcls1_def)

apply auto

done

lemma subcls1_def2:

```
"subcls1 =
  (SIGMA C: {C. is_class C} . {D. C≠Object ∧ super (the (class C)) = D})"
```

apply (unfold subcls1_def is_class_def)

apply auto

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

constdefs

```
ws_prog :: "bool"
```

```
"ws_prog ≡ ∀ (C,c)∈set Prog. C≠Object →
  is_class (super c) ∧ (super c,C)∉subcls1^+"
```

lemma ws_progD: "[class C = Some c; C≠Object; ws_prog] ⇒

```
is_class (super c) ∧ (super c,C)∉subcls1^+"
```

apply (unfold ws_prog_def class_def)

apply (drule_tac map_of_SomeD)

apply auto

done

lemma subcls1_irrefl_lemma1: "ws_prog ⇒ subcls1⁻¹ ∩ 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 ⇒ arbitrary
    | Some m ⇒ if wf (subcls1-1)
    then (if C=Object then empty else class_rec (super m) f) ++ map_of (f m)
    else arbitrary)"
(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

constdefs

```
body :: "cname × mname => stmt"
"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

```
"fields" ↦ (type)"fname => val option"
"obj"    ↦ (type)"cname × fields"
```

constdefs

```
init_vars:: "('a → 'b) => ('a → 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"`
by `simp`

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

lemma `lupd_get_local_other [simp]: "x ≠ y ⟹ lupd(x↦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↦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↦y) s) = heap s"
by (simp add: lupd_def)

lemma heap_hupd_same [simp]: "heap (hupd(a↦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  $\implies$  ( $\exists a. v = \text{Addr } a \wedge \text{heap } s \ a = \text{None}$ )  $\mid v = \text{Null}$ "
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" ("_  $\xrightarrow{-}$  _" [98,90, 65,98] 89)
  and eval :: "[state, expr, val, nat, state] => bool" ("_  $\xrightarrow{-}$  _" [98,95,99,65,98] 89)
where
  Skip: " s  $\xrightarrow{-}$  Skip  $\rightarrow$  s "

  | Comp: "[| s0  $\xrightarrow{-}$  c1  $\rightarrow$  s1; s1  $\xrightarrow{-}$  c2  $\rightarrow$  s2 |] ==>
    s0  $\xrightarrow{-}$  c1;; c2  $\rightarrow$  s2 "

  | Cond: "[| s0  $\xrightarrow{-}$  e  $\rightarrow$  v  $\rightarrow$  s1; s1  $\xrightarrow{-}$  (if v  $\neq$  Null then c1 else c2)  $\rightarrow$  s2 |] ==>
    s0  $\xrightarrow{-}$  If(e) c1 Else c2  $\rightarrow$  s2 "

  | LoopF: " s0 <x> = Null ==>
    s0  $\xrightarrow{-}$  While(x) c  $\rightarrow$  s0 "
  | LoopT: "[| s0 <x>  $\neq$  Null; s0  $\xrightarrow{-}$  c  $\rightarrow$  s1; s1  $\xrightarrow{-}$  While(x) c  $\rightarrow$  s2 |] ==>
    s0  $\xrightarrow{-}$  While(x) c  $\rightarrow$  s2 "

  | LAcc: " s  $\xrightarrow{-}$  LAcc x  $\rightarrow$  s <x>  $\rightarrow$  s "

  | LAss: " s  $\xrightarrow{-}$  e  $\rightarrow$  v  $\rightarrow$  s' ==>
    s  $\xrightarrow{-}$  x ::= e  $\rightarrow$  lupd(x  $\mapsto$  v) s' "

  | FAcc: " s  $\xrightarrow{-}$  e  $\rightarrow$  Addr a  $\rightarrow$  s' ==>
    s  $\xrightarrow{-}$  e..f  $\rightarrow$  get_field s' a f  $\rightarrow$  s' "

  | FAss: "[| s0  $\xrightarrow{-}$  e1  $\rightarrow$  Addr a  $\rightarrow$  s1; s1  $\xrightarrow{-}$  e2  $\rightarrow$  v  $\rightarrow$  s2 |] ==>
    s0  $\xrightarrow{-}$  e1..f ::= e2  $\rightarrow$  upd_obj a f v s2 "

  | NewC: " new_Addr s = Addr a ==>
    s  $\xrightarrow{-}$  new C  $\rightarrow$  Addr a  $\rightarrow$  new_obj a C s "

  | Cast: "[| s  $\xrightarrow{-}$  e  $\rightarrow$  v  $\rightarrow$  s';
    case v of Null => True  $\mid$  Addr a => obj_class s' a  $\preceq$  C C |] ==>
    s  $\xrightarrow{-}$  Cast C e  $\rightarrow$  v  $\rightarrow$  s' "

  | Call: "[| s0  $\xrightarrow{-}$  e1  $\rightarrow$  a  $\rightarrow$  s1; s1  $\xrightarrow{-}$  e2  $\rightarrow$  p  $\rightarrow$  s2;
    lupd(This  $\mapsto$  a)(lupd(Par  $\mapsto$  p)(del_locs s2))  $\xrightarrow{-}$  Meth (C,m)  $\rightarrow$  s3
    |] ==> s0  $\xrightarrow{-}$  {C}e1..m(e2)  $\rightarrow$  s3 <Res>  $\rightarrow$  set_locs s2 s3 "

  | Meth: "[| s <This> = Addr a; D = obj_class s a; D  $\preceq$  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
  "assn"      ↦ (type)"state => bool"
  "vassn"     ↦ (type)"val => assn"
  "triple"    ↦ (type)"assn × stmt × assn"
  "etriple"   ↦ (type)"assn × expr × vassn"

```

6.1 Hoare Logic Rules

```

inductive

```

```

  hoare :: "[triple set, triple set] => bool"  ("_ |⊢/ _" [61, 61] 60)
  and ehoare :: "[triple set, etriple] => bool" ("_ |⊢e/ _" [61, 61] 60)
  and hoare1 :: "[triple set, assn,stmt,assn] => bool"
    ("_ ⊢/ ({(1_)} / ( _ ) / {(1_)} )" [61, 3, 90, 3] 60)
  and ehoare1 :: "[triple set, assn,expr,vassn] => bool"
    ("_ ⊢e/ ({(1_)} / ( _ ) / {(1_)} )" [61, 3, 90, 3] 60)
where

```

```

  "A ⊢ {P}c{Q} ≡ A |⊢ {P,c,Q}"
| "A ⊢e {P}e{Q} ≡ A |⊢e (P,e,Q)"

| Skip: "A ⊢ {P} Skip {P}"

| Comp: "[| A ⊢ {P} c1 {Q}; A ⊢ {Q} c2 {R} |] ==> A ⊢ {P} c1;;c2 {R}"

| Cond: "[| A ⊢e {P} e {Q};
  ∀ v. A ⊢ {Q v} (if v ≠ Null then c1 else c2) {R} |] ==>
  A ⊢ {P} If(e) c1 Else c2 {R}"

| Loop: "A ⊢ {λs. P s ∧ s<x> ≠ Null} c {P} ==>
  A ⊢ {P} While(x) c {λs. P s ∧ s<x> = Null}"

| LAcc: "A ⊢e {λs. P (s<x>) s} LAcc x {P}"

| LAss: "A ⊢e {P} e {λv s. Q (lupd(x↦v) s)} ==>
  A ⊢ {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 ⊢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: " $\llbracket A \vdash \{P'\} c \{Q\}; \forall s. P s \longrightarrow P' s \rrbracket \implies A \vdash \{P\} c \{Q\}$ "

```

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. (λCm. (P Z Cm, Impl Cm, Q Z Cm)) 'Ms) ⊢
    (λCm. (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

```

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" ("|=n: _" [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" ("|=n: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" ("||=n: _" [61,61] 60)
  "||=n: T ≡ ∀ t∈T. |=n: t"

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

  cenvalid :: "[triple set,etriples   ] => bool" ("_ ||=n:e _" [61,61] 60)
  "A ||=n: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, etriple        ] => 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, etriple    ] => bool" ("⊨- /e_" [61,61] 60)

```

```

lemma nvalid_def2: "⊨n: (P,c,Q) ≡ ∀ s t. s -c-n→ t → P s → Q t"
by (simp add: nvalid_def Let_def)

```

```

lemma valid_def2: "⊨ {P} c {Q} = (∀ n. ⊨n: (P,c,Q))"
apply (simp add: valid_def nvalid_def2)
apply blast
done

```

```

lemma envalid_def2: "⊨n:e (P,e,Q) ≡ ∀ s v t. s -e>v-n→ t → P s → Q v t"
by (simp add: envalid_def Let_def)

```

```

lemma evalid_def2: "⊨e {P} e {Q} = (∀ n. ⊨n:e (P,e,Q))"
apply (simp add: evalid_def envalid_def2)
apply blast
done

```

```

lemma cenvalid_def2:
  "A ⊨e (P,e,Q) = (∀ n. ⊨n: A → (∀ s v t. s -e>v-n→ t → P s → 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: "⊨0: (P,Impl M,Q)"
by (clarsimp simp add: nvalid_def2)

```

```

lemma Impl_nvalid_Suc: "⊨n: (P,body M,Q) ⇒ ⊨Suc n: (P,Impl M,Q)"
by (clarsimp simp add: nvalid_def2)

```

```

lemma nvalid_SucD: "⋀ t. ⊨Suc n: t ⇒ ⊨n: t"
by (force simp add: split_paired_all nvalid_def2 intro: exec_mono)

```

```

lemma nvalids_SucD: "Ball A (nvalid (Suc n)) ⇒ Ball A (nvalid n)"
by (fast intro: nvalid_SucD)

```

```

lemma Loop_sound_lemma [rule_format (no_asm)]:
  "∀ s t. s -c-n→ t → P s ∧ s<x> ≠ Null → P t ⇒
    (s -c0-n0→ t → P s → c0 = While (x) c → n0 = n → P t ∧ t<x> = 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:
  "⋈ [∀ z n. Ball (A ∪ B) (nvalid n) → Ball (f z ' Ms) (nvalid n);
    Cm ∈ Ms; Ball A (nvalid na); Ball B (nvalid na)] ⇒ nvalid na (f z Cm)"
by blast

```

```

lemma all_conjunct2: "∀ l. P' l ∧ P l ⇒ ∀ l. P l"
by fast

```

```

lemma all3_conjunct2:

```



```

"∀ a p l. (P' a p l ∧ P a p l) ⇒ ∀ a p l. P a p l"
by fast

lemma cinvalid1_eq:
  "A ⊨ {(P,c,Q)} ≡ ∀ n. ⊨ n: A → (∀ s t. s -c-n→ t → P s → Q t)"
by (simp add: cvalids_def nvalids_def nvalid_def2)

lemma hoare_sound_main: "⋀ t. (A ⊢ C → A ⊨ C) ∧ (A ⊢e t → A ⊨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 "hoare ?x ?y") *})
apply (tactic {* ALLGOALS (REPEAT o thin_tac "ehoare ?x ?y") *})
apply (simp_all only: cinvalid1_eq cinvalid_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: cvalids_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 cvalids_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 cinvalid_def nvalids_def)

```

```

apply fast
done

```

7.3 (Relative) Completeness

```

constdefs MGT      :: "stmt => state => triple"
      "MGT c Z ≡ (λs. Z = s, c, λ t. ∃n. Z -c- n → t)"
      MGT_e      :: "expr => state => etriple"
      "MGT_e e Z ≡ (λs. Z = s, e, λv t. ∃n. Z -e>v-n → t)"
syntax (xsymbols)
      MGT_e      :: "expr => state => etriple" ("MGT_e")
syntax (HTML output)
      MGT_e      :: "expr => state => etriple" ("MGT_e")

```

```

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 MGT_e e Z ⇒ ⊢_e {P} e {Q} ⇒ {} ⊢_e {P} e {Q}"
apply (simp only: evalid_def2)
apply (unfold MGT_e_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 MGT_e e Z)"
apply (simp add: MGT_def MGT_e_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 = " $\lambda a s. \exists n. Z \text{-} \text{expr1} \succ \text{Addr } a \text{-} n \rightarrow s$ " 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 = " $\lambda a v s. \exists n1 n2 t. Z \text{-} \text{expr1} \succ a \text{-} n1 \rightarrow t \wedge t \text{-} \text{expr2} \succ v \text{-} n2 \rightarrow s$ " 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: "{ } ⊢ {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: "⊨ {P} c {Q} ⇒ { } ⊢ {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: "⊨e {P} e {Q} ⇒ { } ⊢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 ⊢ {λs. False} c {Q}"
apply (rule cThin)
apply (rule hoare_relative_complete)
apply (auto simp add: valid_def)
done

lemma eFalse: "A ⊢e {λs. 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
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 ≤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 "simp_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 = "λZ Cm s. s:s<This> ≥ fst Z ∧ s:s<Par> ≥ snd Z" in AxSem.Impl1)
apply (clarsimp simp add: body_def)
apply (rename_tac n m)
apply (rule_tac Q = "λv s. (s:s<This> ≥ n ∧ s:s<Par> ≥ m) ∧
  (∃a. s<This> = Addr a ∧ v = get_field s 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 = "λv s. (∀m. n = Suc m ⟶ s:v ≥ m) ∧ s:s<Par> ≥ m" and
  R = "λT P s. (∀m. n = Suc m ⟶ s:T ≥ m) ∧ s:P ≥ 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 = "((case n of 0 ⇒ 0 | Suc m ⇒ 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. μ 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.