

NanoJava

David von Oheimb

Tobias Nipkow

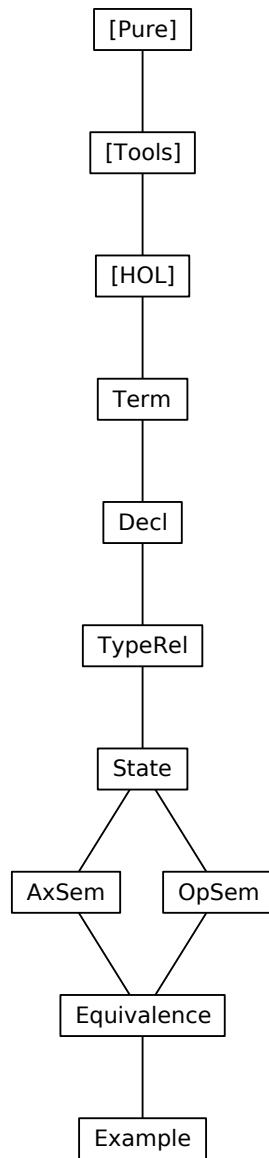
December 12, 2021

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 <https://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	4
4	Program State	6
4.1	Properties not used in the meta theory	7
5	Operational Evaluation Semantics	9
6	Axiomatic Semantics	10
6.1	Hoare Logic Rules	10
6.2	Fully polymorphic variants, required for Example only	12
6.3	Derived Rules	12
7	Equivalence of Operational and Axiomatic Semantics	13
7.1	Validity	13
7.2	Soundness	14
7.3	(Relative) Completeness	14
8	Example	15
8.1	Program representation	16
8.2	“atleast” relation for interpretation of Nat “values”	16
8.3	Proof(s) using the Hoare logic	17



1 Statements and expression emulations

theory *Term* imports *Main* begin

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

axiomatization

This — This pointer
Par — 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

type_synonym *fdecl*
 = "*fname* × *ty*"

record *methd*

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

Method declaration

type_synonym *mdecl*
 = "*mname* × *methd*"

record "*class*"

```

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

```

Class declaration

```

type_synonym cdecl
  = "cname × class"

```

```

type_synonym 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"

```

axiomatization

```

Prog    :: prog      — program as a global value
and
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}"
<proof>

```

end

3 Type relations

```

theory TypeRel
imports Decl
begin

```

Direct subclass relation

```

definition subcls1 :: "(cname × cname) set"
where
  "subcls1 ≡ {(C,D). C ≠ Object ∧ (∃ c. class C = Some c ∧ super c=D)}"

```

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*"

```

3.1 Declarations and properties not used in the meta theory

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)"
⟨proof⟩

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

lemma subcls1_def2:
  "subcls1 =
    (SIGMA C: {C. is_class C} . {D. C≠Object ∧ super (the (class C)) = D})"
⟨proof⟩

lemma finite_subcls1: "finite subcls1"
⟨proof⟩

definition ws_prog :: "bool" where
  "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+"
⟨proof⟩

lemma subcls1_irrefl_lemma1: "ws_prog ⇒ subcls1-1 ∩ subcls1+ = {}"
⟨proof⟩

lemma irrefl_trancI': "r-1 ∩ r+ = {} ⇒ ∀ x. (x, x) ∉ r+"
⟨proof⟩

lemmas subcls1_irrefl_lemma2 = subcls1_irrefl_lemma1 [THEN irrefl_trancI']

lemma subcls1_irrefl: "[[ (x, y) ∈ subcls1; ws_prog ] ⇒ x ≠ y"
⟨proof⟩

lemmas subcls1_acyclic = subcls1_irrefl_lemma2 [THEN acyclicI]

lemma wf_subcls1: "ws_prog ⇒ wf (subcls1-1)"
⟨proof⟩

definition class_rec :: "cname ⇒ (class ⇒ ('a × 'b) list) ⇒ ('a → 'b)"
where
  "class_rec ≡ wfrec (subcls1-1) (λrec C f.
    case class C of None ⇒ undefined
    | Some m ⇒ (if C = Object then Map.empty else rec (super m) f) ++ map_of (f m))"

lemma class_rec: "[[class C = Some m; ws_prog] ⇒
  class_rec C f = (if C = Object then Map.empty else class_rec (super m) f) ++
    map_of (f m)"
⟨proof⟩

definition "method" :: "cname ⇒ (mname → methd)" where
  "method C ≡ class_rec C methods"

```

```

lemma method_rec: "[[class C = Some m; ws_prog]] ==>
method C = (if C=Object then Map.empty else method (super m)) ++ map_of (methods m)"
<proof>
definition field :: "cname => (fname -> ty)" where
  "field C ≡ class_rec C flds"

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

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
typedec1 loc

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

type_synonym fields
  = "(fname -> val)"

type_synonym
  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 == map_option (λT. Null) o m"

private:

type_synonym heap = "loc -> obj"
type_synonym 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 := Map.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 ≡ fst (get_obj s a)"
```

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

— local function:

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

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

```
definition new_obj :: "loc => cname => state => state" where
  "new_obj a C ≡ hupd(a↦(C,init_vars (field C)))"
```

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

```
definition new_Addr      :: "state => val" where
  "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"
<proof>

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"
<proof>

lemma Impl_body_eq: "(λt. ∃ n. Z -Impl M-n → t) = (λt. ∃ n. Z -body M-n → t)"
<proof>

end

```

6 Axiomatic Semantics

```
theory AxSem imports State begin
```

```

type_synonym assn = "state => bool"
type_synonym vassn = "val => assn"
type_synonym triple = "assn × stmt × assn"
type_synonym etriple = "assn × expr × vassn"
translations
  (type) "assn" ← (type) "state => bool"
  (type) "vassn" ← (type) "val => assn"
  (type) "triple" ← (type) "assn × stmt × assn"
  (type) "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*: " $a \in A \implies A \vdash \{a\}$ "

| *ConjI*: " $\forall c \in C. A \vdash \{c\} \implies A \vdash C$ "

| *ConjE*: " $[A \vdash C; c \in C] \implies A \vdash \{c\}$ "

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

| *Conseq*: " $[\forall Z :: \text{state}. A \vdash \{P' Z\} c \{Q' Z\};$
 $\forall s t. (\forall Z. P' Z s \longrightarrow Q' Z t) \longrightarrow (P s \longrightarrow Q t)] \implies$
 $A \vdash \{P\} c \{Q\}$ "

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

| *eConseq*: " $[\forall Z :: \text{state}. A \vdash_e \{P' Z\} e \{Q' Z\};$
 $\forall s v t. (\forall Z. P' Z s \longrightarrow Q' Z v t) \longrightarrow (P s \longrightarrow Q v t)] \implies$
 $A \vdash_e \{P\} e \{Q\}$ "

6.2 Fully polymorphic variants, required for Example only

axiomatization where

Conseq: " $[\forall Z. A \vdash \{P' Z\} c \{Q' Z\};$
 $\forall s t. (\forall Z. P' Z s \longrightarrow Q' Z t) \longrightarrow (P s \longrightarrow Q t)] \implies$
 $A \vdash \{P\} c \{Q\}$ "

axiomatization where

eConseq: " $[\forall Z. A \vdash_e \{P' Z\} e \{Q' Z\};$
 $\forall s v t. (\forall Z. P' Z s \longrightarrow Q' Z v t) \longrightarrow (P s \longrightarrow Q v t)] \implies$
 $A \vdash_e \{P\} e \{Q\}$ "

axiomatization where

Impl: " $\forall Z. A \cup (\bigcup Z. (\lambda \text{Cm}. (P Z \text{Cm}, \text{Impl Cm}, Q Z \text{Cm}))' \text{Ms}) \vdash$
 $(\lambda \text{Cm}. (P Z \text{Cm}, \text{body Cm}, Q Z \text{Cm}))' \text{Ms} \implies$
 $A \vdash (\lambda \text{Cm}. (P Z \text{Cm}, \text{Impl Cm}, Q Z \text{Cm}))' \text{Ms}$ "

6.3 Derived Rules

lemma *Conseq1*: " $\llbracket A \vdash \{P'\} c \{Q'\}; \forall s. P s \longrightarrow P' s \rrbracket \implies A \vdash \{P\} c \{Q\}$ "
 $\langle \text{proof} \rangle$

lemma *Conseq2*: " $\llbracket A \vdash \{P\} c \{Q'\}; \forall t. Q' t \longrightarrow Q t \rrbracket \implies A \vdash \{P\} c \{Q\}$ "
 $\langle \text{proof} \rangle$

lemma *eConseq1*: " $\llbracket A \vdash_e \{P'\} e \{Q'\}; \forall s. P s \longrightarrow P' s \rrbracket \implies A \vdash_e \{P\} e \{Q\}$ "
 $\langle \text{proof} \rangle$

lemma *eConseq2*: " $\llbracket A \vdash_e \{P\} e \{Q'\}; \forall v t. Q' v t \longrightarrow Q v t \rrbracket \implies A \vdash_e \{P\} e \{Q\}$ "
 $\langle \text{proof} \rangle$

lemma *Weaken*: " $\llbracket A \vdash C'; C \subseteq C' \rrbracket \implies A \vdash C$ "
 $\langle \text{proof} \rangle$

lemma *Thin_lemma*:
 $\llbracket (A' \vdash C \longrightarrow (\forall A. A' \subseteq A \longrightarrow A \vdash C)) \wedge$
 $(A' \vdash_e \{P\} e \{Q\} \longrightarrow (\forall A. A' \subseteq A \longrightarrow A \vdash_e \{P\} e \{Q\})) \rrbracket$
 $\langle \text{proof} \rangle$

lemma *cThin*: " $\llbracket A' \vdash C; A' \subseteq A \rrbracket \implies A \vdash C$ "
 $\langle \text{proof} \rangle$

lemma *eThin*: " $\llbracket A' \vdash_e \{P\} e \{Q\}; A' \subseteq A \rrbracket \implies A \vdash_e \{P\} e \{Q\}$ "

$\langle proof \rangle$

lemma *Union*: " $A \vdash (\bigcup Z. C Z) = (\forall Z. A \vdash C Z)$ "

$\langle proof \rangle$

lemma *Impl1'*:

" $\llbracket \forall Z :: state. A \cup (\bigcup Z. (\lambda Cm. (P Z Cm, Impl Cm, Q Z Cm)) 'Ms) \vdash$
 $(\lambda Cm. (P Z Cm, body Cm, Q Z Cm)) 'Ms;$
 $Cm \in Ms \rrbracket \implies$
 $A \vdash \{P Z Cm\} Impl Cm \{Q Z Cm\}$ "

$\langle proof \rangle$

lemmas *Impl1* = *AxSem*.*Impl* [*of* _ _ " $\{Cm\}$ ", *simplified*] **for** *Cm*

end

7 Equivalence of Operational and Axiomatic Semantics

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

7.1 Validity

definition *valid* :: " $[assn, stmt, assn] \Rightarrow bool$ " (" $\models \{(1_)\} / (_) / \{(1_)\}$ " [3,90,3] 60) **where**
 $\models \{P\} c \{Q\} \equiv \forall s \ t. P s \dashrightarrow (\exists n. s \dashv c \dashv n \rightarrow t) \dashrightarrow Q \ t$ "

definition *evalid* :: " $[assn, expr, vassn] \Rightarrow bool$ " (" $\models_e \{(1_)\} / (_) / \{(1_)\}$ " [3,90,3] 60) **where**
 $\models_e \{P\} e \{Q\} \equiv \forall s \ v \ t. P s \dashrightarrow (\exists n. s \dashv e \dashv v \dashv n \rightarrow t) \dashrightarrow Q \ v \ t$ "

definition *nvalid* :: " $[nat, triple] \Rightarrow bool$ " (" $\models_{-} _$ " [61,61] 60) **where**
 $\models_{-} n: t \equiv \text{let } (P, c, Q) = t \text{ in } \forall s \ t. s \dashv c \dashv n \rightarrow t \dashrightarrow P s \dashrightarrow Q \ t$ "

definition *envalid* :: " $[nat, etriple] \Rightarrow bool$ " (" $\models_{-:e} _$ " [61,61] 60) **where**
 $\models_{-:e} n: t \equiv \text{let } (P, e, Q) = t \text{ in } \forall s \ v \ t. s \dashv e \dashv v \dashv n \rightarrow t \dashrightarrow P s \dashrightarrow Q \ v \ t$ "

definition *nvalids* :: " $[nat, triple \text{ set}] \Rightarrow bool$ " (" $\models_{-} _$ " [61,61] 60) **where**
 $\models_{-} n: T \equiv \forall t \in T. \models_{-} n: t$ "

definition *cvalids* :: " $[triple \text{ set}, triple \text{ set}] \Rightarrow bool$ " (" $\models_{-} _ / _$ " [61,61] 60) **where**
 $A \models_{-} C \equiv \forall n. \models_{-} n: A \dashrightarrow \models_{-} n: C$ "

definition *cenvalid* :: " $[triple \text{ set}, etriple] \Rightarrow bool$ " (" $\models_{-} _ / _$ " [61,61] 60) **where**
 $A \models_{-} e t \equiv \forall n. \models_{-} n: A \dashrightarrow \models_{-:e} n: t$ "

lemma *nvalid_def2*: " $\models_{-} n: (P, c, Q) \equiv \forall s \ t. s \dashv c \dashv n \rightarrow t \longrightarrow P s \longrightarrow Q t$ "
 $\langle proof \rangle$

lemma *valid_def2*: " $\models \{P\} c \{Q\} = (\forall n. \models_{-} n: (P, c, Q))$ "
 $\langle proof \rangle$

lemma *envalid_def2*: " $\models_{-:e} n: (P, e, Q) \equiv \forall s \ v \ t. s \dashv e \dashv v \dashv n \rightarrow t \longrightarrow P s \longrightarrow Q v t$ "
 $\langle proof \rangle$

lemma *evalid_def2*: " $\models_e \{P\} e \{Q\} = (\forall n. \models_{-:e} n: (P, e, Q))$ "
 $\langle proof \rangle$

lemma *cenvalid_def2*:
 $A \models_{-} e (P, e, Q) = (\forall n. \models_{-} n: A \longrightarrow (\forall s \ v \ t. s \dashv e \dashv v \dashv n \rightarrow t \longrightarrow P s \longrightarrow Q v t))$ "

$\langle \text{proof} \rangle$

7.2 Soundness

declare *exec_elim_cases* [*elim!*] *eval_elim_cases* [*elim!*]

lemma *Impl_nvalid_0*: " $\models_0: (P, \text{Impl } M, Q)$ "

$\langle \text{proof} \rangle$

lemma *Impl_nvalid_Suc*: " $\models_n: (P, \text{body } M, Q) \implies \models_{\text{Suc } n}: (P, \text{Impl } M, Q)$ "

$\langle \text{proof} \rangle$

lemma *nvalid_SucD*: " $\bigwedge t. \models_{\text{Suc } n} t \implies \models_n t$ "

$\langle \text{proof} \rangle$

lemma *nvalids_SucD*: " $\text{Ball } A (\text{nvalid } (\text{Suc } n)) \implies \text{Ball } A (\text{nvalid } n)$ "

$\langle \text{proof} \rangle$

lemma *Loop_sound_lemma* [*rule_format* (*no_asm*)]:

" $\forall s \ t. s \text{ -c-n} \rightarrow t \longrightarrow P \ s \wedge s\langle x \rangle \neq \text{Null} \longrightarrow P \ t \implies$
 $(s \text{ -c0-n0} \rightarrow t \longrightarrow P \ s \longrightarrow c0 = \text{While } (x) \ c \longrightarrow n0 = n \longrightarrow P \ t \wedge t\langle x \rangle = \text{Null})$ "

$\langle \text{proof} \rangle$

lemma *Impl_sound_lemma*:

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *all3_conjunct2*:

" $\forall a \ p \ l. (P' \ a \ p \ l \wedge P \ a \ p \ l) \implies \forall a \ p \ l. P \ a \ p \ l$ "

$\langle \text{proof} \rangle$

lemma *cnvalid1_eq*:

" $A \models \{(P, c, Q)\} \equiv \forall n. \models_n: A \longrightarrow (\forall s \ t. s \text{ -c-n} \rightarrow t \longrightarrow P \ s \longrightarrow Q \ t)$ "

$\langle \text{proof} \rangle$

lemma *hoare_sound_main*: " $\bigwedge t. (A \vdash C \longrightarrow A \models C) \wedge (A \vdash_e t \longrightarrow A \models_e t)$ "

$\langle \text{proof} \rangle$

theorem *hoare_sound*: " $\{\} \vdash \{P\} \ c \ \{Q\} \implies \models \{P\} \ c \ \{Q\}$ "

$\langle \text{proof} \rangle$

theorem *ehoare_sound*: " $\{\} \vdash_e \{P\} \ e \ \{Q\} \implies \models_e \{P\} \ e \ \{Q\}$ "

$\langle \text{proof} \rangle$

7.3 (Relative) Completeness

definition *MGT* :: "*stmt* => *state* => *triple*" **where**

"*MGT* *c* *Z* $\equiv (\lambda s. Z = s, \ c, \ \lambda t. \exists n. Z \text{ -c- } n \rightarrow t)$ "

definition *MGT_e* :: "*expr* => *state* => *etriples*" **where**

"*MGT_e* *e* *Z* $\equiv (\lambda s. Z = s, \ e, \ \lambda v \ t. \exists n. Z \text{ -e>v-n} \rightarrow t)$ "

lemma *MGF_implies_complete*:

" $\forall Z. \{\} \vdash \{ \text{MGT } c \ Z \} \implies \models \{P\} \ c \ \{Q\} \implies \{\} \vdash \{P\} \ c \ \{Q\}$ "

$\langle \text{proof} \rangle$

```

lemma eMGF_implies_complete:
  "∀ Z. {} ⊢e MGTe e Z ⇒ ⊢e {P} e {Q} ⇒ {} ⊢e {P} e {Q}"
⟨proof⟩

declare exec_eval.intros[intro!]

lemma MGF_Loop: "∀ Z. A ⊢ {(=) Z} c {λt. ∃ n. Z -c-n→ t} ⇒
  A ⊢ {(=) Z} While (x) c {λt. ∃ n. Z -While (x) c-n→ t}"
⟨proof⟩

lemma MGF_lemma: "∀ M Z. A ⊢ {MGT (Impl M) Z} ⇒
  (∀ Z. A ⊢ {MGT c Z}) ∧ (∀ Z. A ⊢e MGTe e Z)"
⟨proof⟩

lemma MGF_Impl: "{} ⊢ {MGT (Impl M) Z}"
⟨proof⟩

theorem hoare_relative_complete: "⊢ {P} c {Q} ⇒ {} ⊢ {P} c {Q}"
⟨proof⟩

theorem ehoare_relative_complete: "⊢e {P} e {Q} ⇒ {} ⊢e {P} e {Q}"
⟨proof⟩

lemma cFalse: "A ⊢ {λs. False} c {Q}"
⟨proof⟩

lemma eFalse: "A ⊢e {λs. False} e {Q}"
⟨proof⟩

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);
}
}

```

axiomatization where

```

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

```

8.1 Program representation

axiomatization

```

N      :: cname ("Nat")
and pred :: fname
and suc add :: mname
and 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.

axiomatization where `Nat_no_subclasses [simp]: "D  $\preceq^C$  Nat = (D=Nat)"`

axiomatization where `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 |)"

```

axiomatization where `method_Nat_suc [simp]: "method Nat suc = Some`

```

(| par=NT, res=Class Nat, lcl=[],
 bdy= Res := new Nat;; LAcc Res..pred := LAcc This |)"

```

axiomatization where `field_Nat [simp]: "field Nat = Map.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, Map.empty(pred $\mapsto$ v))) s"

```

<proof>

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

primrec `Nat_atleast :: "state  $\Rightarrow$  val  $\Rightarrow$  nat  $\Rightarrow$  bool" ("_:_  $\geq$  _" [51, 51, 51] 50) where`

```

"s:x $\geq$ 0      = (x $\neq$ 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)"
 <proof>

lemma Nat_atleast_set_locs [rule_format, simp]:
 "∀ s v::val. set_locs l s:v ≥ n = (s:v ≥ n)"
 <proof>

lemma Nat_atleast_del_locs [rule_format, simp]:
 "∀ s v::val. del_locs s:v ≥ n = (s:v ≥ n)"
 <proof>

lemma Nat_atleast_NullD [rule_format]: "s:Null ≥ n → False"
 <proof>

lemma Nat_atleast_pred_NullD [rule_format]:
 "Null = get_field s a pred ⇒ s:Addr a ≥ n → n = 0"
 <proof>

lemma Nat_atleast_mono [rule_format]:
 "∀ a. s:get_field s a pred ≥ n → heap s a ≠ None → s:Addr a ≥ n"
 <proof>

lemma Nat_atleast_newC [rule_format]:
 "heap s aa = None ⇒ ∀ v::val. s:v ≥ n → hupd(aa↦obj) s:v ≥ n"
 <proof>

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}"
 <proof>

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. <https://isabelle.in.tum.de/Bali/papers/CPE01.html>, to appear.