

NanoJava

David von Oheimb
Tobias Nipkow

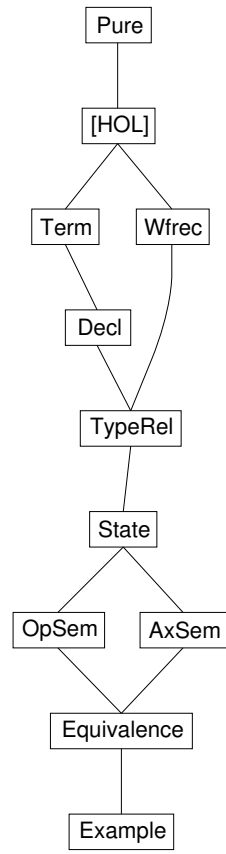
October 10, 2011

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	Well-Founded Recursion Combinator	4
3.1	Nitpick setup	5
3.2	Wellfoundedness of <i>same_fst</i>	6
4	Type relations	6
4.1	Declarations and properties not used in the meta theory	7
5	Program State	8
5.1	Properties not used in the meta theory	10
6	Operational Evaluation Semantics	11
7	Axiomatic Semantics	13
7.1	Hoare Logic Rules	13
7.2	Fully polymorphic variants, required for Example only	15
7.3	Derived Rules	15
8	Equivalence of Operational and Axiomatic Semantics	17
8.1	Validity	17
8.2	Soundness	18
8.3	(Relative) Completeness	19
9	Example	22
9.1	Program representation	23
9.2	“atleast” relation for interpretation of Nat “values”	24
9.3	Proof(s) using the Hoare logic	24



1 Statements and expression emulations

theory *Term* imports *Main* begin

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

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}"
apply (unfold is_class_def class_def)
apply (fold dom_def)
apply (rule finite_dom_map_of)
done

```

end

3 Well-Founded Recursion Combinator

```

theory Wfrec
imports Main
begin

```

inductive

```

wfrec_rel :: "('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b => bool"
for R :: "('a * 'a) set"
and F :: "('a => 'b) => 'a => 'b"

```

where

```

wfrecI: "ALL z. (z, x) : R --> wfrec_rel R F z (g z) ==>
  wfrec_rel R F x (F g x)"

```

definition

```

cut      :: "('a => 'b) => ('a * 'a)set => 'a => 'a => 'b" where
  "cut f r x == (%y. if (y,x):r then f y else undefined)"

```

definition

```
adm_wf :: "('a * 'a) set => (('a => 'b) => 'a => 'b) => bool" where
"adm_wf R F == ALL f g x.
  (ALL z. (z, x) : R --> f z = g z) --> F f x = F g x"
```

definition

```
wfrec :: "('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b" where
"wfrec R F == %x. THE y. wfrec_rel R (%f x. F (cut f R x) x) x y"
```

```
lemma cuts_eq: "(cut f r x = cut g r x) = (ALL y. (y,x):r --> f(y)=g(y))"
by (simp add: fun_eq_iff cut_def)
```

```
lemma cut_apply: "(x,a):r ==> (cut f r a)(x) = f(x)"
by (simp add: cut_def)
```

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

```
lemma wfrec_unique: "[| adm_wf R F; wf R |] ==> EX! y. wfrec_rel R F x y"
apply (simp add: adm_wf_def)
apply (erule_tac a=x in wf_induct)
apply (rule ex1I)
apply (rule_tac g = "%x. THE y. wfrec_rel R F x y" in wfrec_rel.wfrecI)
apply (fast dest!: theI')
apply (erule wfrec_rel.cases, simp)
apply (erule allE, erule allE, erule allE, erule mp)
apply (fast intro: the_equality [symmetric])
done
```

```
lemma adm_lemma: "adm_wf R (%f x. F (cut f R x) x)"
apply (simp add: adm_wf_def)
apply (intro strip)
apply (rule cuts_eq [THEN iffD2, THEN subst], assumption)
apply (rule refl)
done
```

```
lemma wfrec: "wf(r) ==> wfrec r H a = H (cut (wfrec r H) r a) a"
apply (simp add: wfrec_def)
apply (rule adm_lemma [THEN wfrec_unique, THEN the1_equality], assumption)
apply (rule wfrec_rel.wfrecI)
apply (intro strip)
apply (erule adm_lemma [THEN wfrec_unique, THEN theI'])
done
```

* This form avoids giant explosions in proofs. NOTE USE OF ==

```
lemma def_wfrec: "[| f==wfrec r H; wf(r) |] ==> f(a) = H (cut f r a) a"
apply auto
apply (blast intro: wfrec)
done
```

3.1 Nitpick setup

```
axiomatization wf_wfrec :: "('a × 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b"
```

```
definition wf_wfrec' :: "('a × 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b" where
[nitpick_simp]: "wf_wfrec' R F x = F (cut (wf_wfrec R F) R x) x"
```

```
definition wfrec' :: "('a × 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b" where
```

```
"wfrec' R F x ≡ if wf R then wf_wfrec' R F x
      else THE y. wfrec_rel R (%f x. F (cut f R x) x) x y"
```

```
setup {*
  Nitpick_HOL.register_ersatz_global
  [(@{const_name wf_wfrec'}, @{const_name wf_wfrec'}),
   (@{const_name wfrec'}, @{const_name wfrec'})]
*}
```

```
hide_const (open) wf_wfrec wf_wfrec' wfrec'
hide_fact (open) wf_wfrec'_def wfrec'_def
```

3.2 Wellfoundedness of *same_fst*

definition

```
same_fst :: "('a => bool) => ('a => ('b * 'b)set) => (('a*'b)*( 'a*'b))set"
```

where

```
"same_fst P R == {(x',y'),(x,y)) . x'=x & P x & (y',y) : R x}"
```

— For *rec_def* declarations where the first n parameters stay unchanged in the recursive call.

lemma same_fstI [intro!]:

```
"[| P x; (y',y) : R x |] ==> ((x,y'),(x,y)) : same_fst P R"
```

```
by (simp add: same_fst_def)
```

lemma wf_same_fst:

```
assumes prem: "(!x. P x ==> wf(R x))"
```

```
shows "wf(same_fst P R)"
```

```
apply (simp cong del: imp_cong add: wf_def same_fst_def)
```

```
apply (intro strip)
```

```
apply (rename_tac a b)
```

```
apply (case_tac "wf (R a)")
```

```
  apply (erule_tac a = b in wf_induct, blast)
```

```
apply (blast intro: prem)
```

```
done
```

```
end
```

4 Type relations

```
theory TypeRel imports Decl "~/src/HOL/Library/Wfrec" 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^*"
```

notation (xsymbols)

```
subcls1_syntax ("_ <C1 _" [71,71] 70) and
```

```
subcls_syntax ("_ <C _" [71,71] 70)
```

4.1 Declarations and properties not used in the meta theory

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)"`

`apply (unfold subcls1_def)`

`apply auto`

`done`

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

`apply (unfold subcls1_def)`

`apply auto`

`done`

lemma subcls1_def2:

`"subcls1 =`

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

`apply (unfold subcls1_def is_class_def)`

`apply (auto split:split_if_asm)`

`done`

lemma finite_subcls1: `"finite subcls1"`

`apply(subst subcls1_def2)`

`apply(rule finite_SigmaI [OF finite_is_class])`

`apply(rule_tac B = "{super (the (class C))}" in finite_subset)`

`apply auto`

`done`

definition ws_prog :: "bool" where

`"ws_prog \equiv \forall (C,c) \in 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+"`

`apply (unfold ws_prog_def class_def)`

`apply (drule_tac map_of_SomeD)`

`apply auto`

`done`

lemma subcls1_irrefl_lemma1: `"ws_prog \implies subcls1-1 \cap subcls1+ = {}"`

`by (fast dest: subcls1D ws_progD)`

lemma irrefl_tranclI': `"r-1 Int r+ = {} \implies !x. (x, x) \sim : r+"`

`by(blast elim: tranclE dest: trancl_into_rtrancl)`

lemmas subcls1_irrefl_lemma2 = subcls1_irrefl_lemma1 [THEN irrefl_tranclI']

lemma subcls1_irrefl: `"[[(x, y) \in subcls1; ws_prog]] \implies x \neq 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  $\implies$  wf (subcls1-1)"
by (auto intro: finite_acyclic_wf_converse finite_subcls1 subcls1_acyclic)

definition class_rec :: "cname  $\Rightarrow$  (class  $\Rightarrow$  ('a  $\times$  'b) list)  $\Rightarrow$  ('a  $\rightarrow$  'b)"
where
  "class_rec  $\equiv$  wfrec (subcls1-1) ( $\lambda$ rec C f.
    case class C of None  $\Rightarrow$  undefined
    | Some m  $\Rightarrow$  (if C = Object then empty else rec (super m) f) ++ map_of (f m))"

lemma class_rec: "[[class C = Some m; ws_prog]]  $\implies$ 
  class_rec C f = (if C = Object then empty else class_rec (super m) f) ++
    map_of (f m)"
apply (drule wf_subcls1)
apply (subst def_wfrec[OF class_rec_def], auto)
apply (subst cut_apply, auto intro: subcls1I)
done

— Methods of a class, with inheritance and hiding
definition method :: "cname  $\Rightarrow$  (mname  $\rightarrow$  methd)" where
  "method C  $\equiv$  class_rec C methods"

lemma method_rec: "[[class C = Some m; ws_prog]]  $\implies$ 
  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
definition field :: "cname  $\Rightarrow$  (fname  $\rightarrow$  ty)" where
  "field C  $\equiv$  class_rec C flds"

lemma flds_rec: "[[class C = Some m; ws_prog]]  $\implies$ 
  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

```

5 Program State

```

theory State imports TypeRel begin

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

Locations, i.e. abstract references to objects
typedecl loc

```

```

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

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

type_synonym
  obj = "cname  $\times$  fields"

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

definition init_vars :: "('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  val)" where
  "init_vars m  $\equiv$  Option.map ( $\lambda T$ . Null) o m"

```

private:

```

type_synonym heap = "loc  $\rightarrow$  obj"
type_synonym locals = "vname  $\rightarrow$  val"

```

private:

```

record state
  = heap    :: heap
    locals  :: locals

```

translations

```

(type) "heap"  $\leftarrow$  (type) "loc  $\Rightarrow$  obj option"
(type) "locals"  $\leftarrow$  (type) "vname  $\Rightarrow$  val option"
(type) "state"  $\leftarrow$  (type) "(|heap :: heap, locals :: locals|)"

```

```

definition del_locs :: "state  $\Rightarrow$  state" where
  "del_locs s  $\equiv$  s (| locals := empty |)"

```

```

definition init_locs      :: "cname  $\Rightarrow$  mname  $\Rightarrow$  state  $\Rightarrow$  state" where
  "init_locs C m s  $\equiv$  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  $\Rightarrow$  state  $\Rightarrow$  state" where
  "set_locs s s'  $\equiv$  s' (| locals := locals s |)"

```

```

definition get_local      :: "state  $\Rightarrow$  vname  $\Rightarrow$  val" ("_<_>" [99,0] 99) where
  "get_local s x  $\equiv$  the (locals s x)"

```

— local function:

```

definition get_obj        :: "state  $\Rightarrow$  loc  $\Rightarrow$  obj" where
  "get_obj s a  $\equiv$  the (heap s a)"

```

```

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

```

```

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

```

— local function:

```

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

```

```

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

```

```

notation (xsymbols)
  hupd ("hupd'(_|->_')" [10,10] 1000) and
  lupd ("lupd'(_|->_')" [10,10] 1000)

```

```

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"

```

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

end

```

6 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 γv-n → t"
"s -Cast C e γv-n → t"
"s -LAcc x γv-n → t"
"s -e..f γv-n → t"
"s -{C}e1..m(e2) γv-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

```

7 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"

```

7.1 Hoare Logic Rules

```
inductive
```

```

hoare :: "[triple set, triple set] => bool" ("_ | $\vdash$ /_" [61, 61] 60)
and ehoare :: "[triple set, etriple] => 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} |] ==> A  $\vdash$  {P} c1;;c2 {R}"

| Cond: "[| A  $\vdash_e$  {P} e {Q};
   $\forall v. A \vdash$  {Q v} (if v  $\neq$  Null then c1 else c2) {R} |] ==>
  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} ==>
  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 { $\lambda v s. Q (\text{lupd}(x \mapsto v) s)$ } ==>
  A  $\vdash$  {P} x::=e {Q}"

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

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

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

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

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

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

```

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

```

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

```

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

7.2 Fully polymorphic variants, required for Example only

axioms

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

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

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

7.3 Derived Rules

lemma Conseq1: " $[A \vdash \{P'\} c \{Q\}; \forall s. P s \longrightarrow P' s] \implies A \vdash \{P\} c \{Q\}$ "
 apply (rule hoare_ehoare.Conseq)
 apply (rule allI, assumption)
 apply fast
 done

lemma Conseq2: " $[A \vdash \{P\} c \{Q'\}; \forall t. Q' t \longrightarrow Q t] \implies A \vdash \{P\} c \{Q\}$ "
 apply (rule hoare_ehoare.Conseq)
 apply (rule allI, assumption)
 apply fast
 done

lemma eConseq1: " $[A \vdash_e \{P'\} e \{Q\}; \forall s. P s \longrightarrow P' s] \implies A \vdash_e \{P\} e \{Q\}$ "
 apply (rule hoare_ehoare.eConseq)
 apply (rule allI, assumption)
 apply fast
 done

lemma eConseq2: " $[A \vdash_e \{P\} e \{Q'\}; \forall v t. Q' v t \longrightarrow Q v t] \implies A \vdash_e \{P\} e \{Q\}$ "
 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 @{context}, 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

8 Equivalence of Operational and Axiomatic Semantics

theory Equivalence imports OpSem AxSem begin

8.1 Validity

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

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

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

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

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

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

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

notation (xsymbols)

valid ("|= {(1_)} / () / {(1_)}" [3,90,3] 60) and
 evalid ("|=e {(1_)} / () / {(1_)}" [3,90,3] 60) and
 nvalid ("|=_: _" [61,61] 60) and
 envalid ("|=:_e _" [61,61] 60) and
 nvalids ("||=: _" [61,61] 60) and
 cnvalids ("_ ||=/ _" [61,61] 60) and
 cenvalid ("_ ||=/_e _" [61,61] 60)

lemma nvalid_def2: "|=: (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. |=_: (P,c,Q))"
 apply (simp add: valid_def nvalid_def2)
 apply blast
 done

lemma envalid_def2: "|=:_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. |=:_e (P,e,Q))"
 apply (simp add: evalid_def envalid_def2)
 apply blast
 done

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

```
by(simp add: cenvalid_def envalid_def2)
```

8.2 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 cnvalid1_eq:
```

```
"A ⊨ {P,c,Q} ≡ ∀n. ⊨ₙ: A → (∀s t. s -c-n→ t → P s → Q t)"
```

```
by(simp add: cnvalids_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 @{context} "hoare ?x ?y") *})
```

```
apply (tactic {* ALLGOALS (REPEAT o thin_tac @{context} "ehoare ?x ?y") *})
```

```
apply (simp_all only: cnvalid1_eq cenvalid_def2)
```

```
apply fast
```

```
apply fast
```

```
apply fast
```

```
apply (clarify, tactic "smp_tac 1 1",erule(2) Loop_sound_lemma,(rule HOL.refl)+)
```

```
apply fast
```

```
apply fast
```

```
apply fast
```

```
apply fast
```

```
apply fast
```

```

    apply fast
    apply (clarsimp del: Meth_elim_cases)
    apply (force del: Impl_elim_cases)
  defer
    prefer 4 apply blast
    prefer 4 apply blast
    apply (simp_all (no_asm_use) only: 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: HOL.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 cenvalid_def nvalids_def)
  apply fast
done

```

8.3 (Relative) Completeness

```

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

```

```

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

```

```

notation (xsymbols)

```

```

  MGTe ("MGTe")

```

```

notation (HTML output)

```

```

  MGTe ("MGTe")

```

```

lemma MGF_implies_complete:

```

```

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

```

```

lemma eMGF_implies_complete:

```

```

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

declare exec_eval.intros[intro!]

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

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

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

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

apply (erule thin_rl)
apply (rule hoare_ehoare.Cond)
apply (erule spec)
apply (rule allI)
apply (simp)
apply (rule conjI)
apply (rule impI, erule hoare_ehoare.Conseq, clarsimp, drule (1) eval_exec_max,
  erule thin_rl, erule thin_rl, force)+

apply (erule MGF_Loop)

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

```

```

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

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

apply (simp add: split_paired_all)

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

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

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

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

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

lemma MGF_Impl: "{ } | $\vdash$  {MGT (Impl M) Z}"
apply (unfold MGT_def)
apply (rule Impl1')
apply (rule_tac [2] UNIV_I)
apply clarsimp
apply (rule hoare_ehoare.ConjI)
apply clarsimp
apply (rule ssubst [OF Impl_body_eq])
apply (fold MGT_def)
apply (rule MGF_lemma [THEN conjunct1, rule_format])
apply (rule hoare_ehoare.Asm)

```

```

apply force
done

```

```

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

```

```

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

```

```

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

```

```

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

```

```

end

```

9 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"

```

9.1 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 ≼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 = 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)

```

9.2 “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 $\geq$ 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)"
apply (induct n)
by auto
```

```
lemma Nat_atleast_set_locs [rule_format, simp]:
  " $\forall$ s v::val. set_locs l s:v  $\geq$  n = (s:v  $\geq$  n)"
apply (induct n)
by auto
```

```
lemma Nat_atleast_del_locs [rule_format, simp]:
  " $\forall$ s v::val. del_locs s:v  $\geq$  n = (s:v  $\geq$  n)"
apply (induct n)
by auto
```

```
lemma Nat_atleast_NullD [rule_format]: "s:Null  $\geq$  n  $\longrightarrow$  False"
apply (induct n)
by auto
```

```
lemma Nat_atleast_pred_NullD [rule_format]:
  "Null = get_field s a pred  $\implies$  s:Addr a  $\geq$  n  $\longrightarrow$  n = 0"
apply (induct n)
by (auto dest: Nat_atleast_NullD)
```

```
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"
apply (induct n)
by auto
```

```
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"
apply (induct n)
apply auto
apply (case_tac "aa=a")
apply auto
apply (tactic "smp_tac 1 1")
apply (case_tac "aa=a")
apply auto
done
```

9.3 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}"
apply (rule hoare_ehoare.Meth)
apply clarsimp
apply (rule_tac P'= " $\lambda$ Z s. (s:s<This>  $\geq$  fst Z  $\wedge$  s:s<Par>  $\geq$  snd Z)  $\wedge$  D=Nat" and
  Q'= " $\lambda$ Z s. s:s<Res>  $\geq$  fst Z+snd Z" in AxSem.Conseq)
prefer 2
apply (clarsimp simp add: init_locs_def init_vars_def)
apply rule
apply (case_tac "D = Nat", simp_all, rule_tac [2] cFalse)
apply (rule_tac P = " $\lambda$ Z Cm s. s:s<This>  $\geq$  fst Z  $\wedge$  s:s<Par>  $\geq$  snd Z" in AxSem.Impl1)
```

```

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

```

end

References

- [1] T. Nipkow, D. v. Oheimb, and C. Pusch. 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.