

```
theory PreBasis = Main:
```

```
  hide const In0 In1
```

```
end
```

1 Basis

```
(* Title:      isabelle/Bali/Basis.thy
   ID:         $Id: Basis.thy,v 1.29 2000/11/14 09:35:34 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen
```

```
Definitions extending HOL as logical basis of Bali
*)
```

```
Basis = PreBasis +
```

```
syntax
```

```
"3" :: nat    ("3")
```

```
"4" :: nat    ("4")
```

```
translations
```

```
"3" == "Suc 2"
```

```
"4" == "Suc 3"
```

```
constdefs
```

```
  the_In1  :: "'a + 'b  $\Rightarrow$  'a"
```

```
"the_In1 x  $\equiv$   $\varepsilon$ a. x = In1 a"
```

```
  the_Inr  :: "'a + 'b  $\Rightarrow$  'b"
```

```
"the_Inr x  $\equiv$   $\varepsilon$ b. x = Inr b"
```

```
datatype ('a, 'b, 'c) sum3 = In1 'a | In2 'b | In3 'c
```

```
constdefs
```

```
  the_In1  :: "('a, 'b, 'c) sum3  $\Rightarrow$  'a"
```

```
"the_In1 x  $\equiv$   $\varepsilon$ a. x = In1 a"
```

```
  the_In2  :: "('a, 'b, 'c) sum3  $\Rightarrow$  'b"
```

```
"the_In2 x  $\equiv$   $\varepsilon$ b. x = In2 b"
```

```
  the_In3  :: "('a, 'b, 'c) sum3  $\Rightarrow$  'c"
```

```
"the_In3 x  $\equiv$   $\varepsilon$ c. x = In3 c"
```

```
syntax
```

```

      Inl  :: "'al ⇒ ('al + 'ar, 'b, 'c) sum3"
      Inr  :: "'ar ⇒ ('al + 'ar, 'b, 'c) sum3"
translations
  "Inl e" == "In1 (Inl e)"
  "Inr c" == "In1 (Inr c)"

translations
  "option" <= (type) "Option.option"
  "list" <= (type) "List.list"
  "sum3" <= (type) "Basis.sum3"

syntax
  fun_sum :: "('a => 'c) => ('b => 'c) => (('a+'b) => 'c)" (infixr "'(+')"80)
translations
  "fun_sum" == "sum_case"

syntax
  "@Oall" :: [pttrn, 'a option, bool] => bool  ("(3! _:_:/ _)" [0,0,10]
10)
  "@Oex"  :: [pttrn, 'a option, bool] => bool  ("(3? _:_:/ _)" [0,0,10]
10)

syntax (symbols)
  "@Oall" :: [pttrn, 'a option, bool] => bool  ("(3∀_∈_:/ _)" [0,0,10]
10)
  "@Oex"  :: [pttrn, 'a option, bool] => bool  ("(3∃_∈_:/ _)" [0,0,10]
10)

translations
  "! x:A: P"    == "! x:o2s A. P"
  "? x:A: P"    == "? x:o2s A. P"

constdefs
  unique  :: "('a × 'b) list ⇒ bool"
  "unique ≡ nodups ◦ map fst"

consts
  lsplit      :: "[['a, 'a list] => 'b, 'a list] => 'b"
defs
  lsplit_def  "lsplit == %f l. f (hd l) (tl l)"
(* list patterns -- extends pre-defined type "pttrn" used in abstractions
*)
syntax

```

```

    "_lpttrn"    :: [pttrn,pttrn] => pttrn    ("_#/" [901,900] 900)
translations
    "%y#x#xs. b" == "lsplit (%y x#xs. b)"
    "%x#xs . b"  == "lsplit (%x xs . b)"

syntax

    "@dummy_pat"  :: pttrn    ("'_")

end

ML

fun dummy_pat_tr [] = Free ("_",dummyT)
  | dummy_pat_tr ts = raise TERM ("dummy_pat_tr", ts);

val parse_translation = ("@dummy_pat", dummy_pat_tr)::parse_translation;

```

2 Name

```

(* Title:      isabelle/Bali/Name.thy
   ID:         $Id: Name.thy,v 1.5 2000/06/29 19:35:31 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

Java names

simplifications:

no packages, thus no internal structure of names

**)*

Name = Basis +

(cf. 6.5 *)*

types tnam (ordinary type name, i.e. class or interface name *)*

ename (expression name, i.e. variable or field name *)*

mname (method name *)*

arities tnam, ename, mname :: term

types lname (names for local variables and the This pointer*

```

*)
    = "ename + unit"
syntax  EName, This :: lname
translations
    "lname" <= (type) "ename + unit"
    "EName" => "Inl"
    "This"  => "Inr ()"

datatype xname          (* names of standard exceptions *)
    = Throwable
    | NullPointerException | OutOfMemory | ClassCast
    | NegArrSize | IndOutBound | ArrStore

datatype tname          (* type names for standard classes and other type
names *)
    = Object
    | SXcpt  xname
    | TName  tnam

translations
    "mname" <= (type) "Name.mname"
    "xname" <= (type) "Name.xname"
    "tname" <= (type) "Name.tname"
    "ename" <= (type) "Name.ename"

end

```

3 Type

```

(* Title:      isabelle/Bali/Type.thy
ID:           $Id: Type.thy,v 1.19 2000/05/02 09:40:54 oheimb Exp $
Author:       David von Oheimb
Copyright    1997 Technische Universitaet Muenchen

```

Java types

simplifications:

```

* only the most important primitive types
* the null type is regarded as reference type
*)

```

Type = Name +

```

datatype prim_ty      (* primitive type, cf. 4.2 *)
  = Void             (* 'result type' of void methods *)
  | Boolean
  | Integer

datatype ref_ty      (* reference type, cf. 4.3 *)
  = NullT           (* null type, cf. 4.1 *)
  | IfaceT tname    (* interface type *)
  | ClassT tname    (* class type *)
  | ArrayT ty       (* array type *)

and ty               (* any type, cf. 4.1 *)
  = PrimT prim_ty   (* primitive type *)
  | RefT ref_ty     (* reference type *)

translations
  "prim_ty" <= (type) "Type.prim_ty"
  "ref_ty"  <= (type) "Type.ref_ty"
  "ty"      <= (type) "Type.ty"

syntax
  NT      :: "          ty"
  Iface   :: "tname ⇒ ty"
  Class   :: "tname ⇒ ty"
  Array   :: "ty      ⇒ ty"  ("_[]" [90] 90)

translations
  "NT"      == "RefT NullT"
  "Iface I" == "RefT (IfaceT I)"
  "Class C" == "RefT (ClassT C)"
  "T.[]"    == "RefT (ArrayT T)"

constdefs
  the_Class :: "ty ⇒ tname"
  "the_Class T ≡ εC. T = Class C"

end

```

4 Value

```
(* Title:      isabelle/Bali/Value.thy
   ID:         $Id: Value.thy,v 1.3 2000/07/14 14:48:59 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

Java values
*)

Value = Type +

types   loc                (* locations, i.e. abstract references on objects
*)

arities loc :: term

datatype val_ (** name not 'val' because of nasty clash with ML token
'val' **)
  = Unit          (* dummy result value of void methods *)
  | Bool bool     (* Boolean value *)
  | Intg int      (* integer value *)
  | Null          (* null reference *)
  | Addr loc      (* addresses, i.e. locations of objects *)
types     val = val_
translations "val" <= (type) "val_"
            "val" <= (type) "Term.val_"
            "loc" <= (type) "Term.loc"

constdefs
  the_Bool  :: "val  $\Rightarrow$  bool"   "the_Bool v  $\equiv$   $\varepsilon$ b. v = Bool b"
  the_Intg  :: "val  $\Rightarrow$  int"     "the_Intg v  $\equiv$   $\varepsilon$ i. v = Intg i"
  the_Addr  :: "val  $\Rightarrow$  loc"     "the_Addr v  $\equiv$   $\varepsilon$ a. v = Addr a"

types   dyn_ty            = "loc  $\Rightarrow$  ty option"
consts
  typeof      :: "dyn_ty  $\Rightarrow$  val  $\Rightarrow$  ty option"
  defpval     :: "prim_ty  $\Rightarrow$  val"      (* default value for primitive
types *)
  default_val :: "      ty  $\Rightarrow$  val"    (* default value for all types
*)

primrec "typeof dt Unit      = Some (PrimT Void)"
       "typeof dt (Bool b) = Some (PrimT Boolean)"
       "typeof dt (Intg i) = Some (PrimT Integer)"
```

```

        "typeof dt  Null    = Some NT"
        "typeof dt (Addr a) = dt a"

primrec "defpval Void    = Unit"
        "defpval Boolean = Bool False"
        "defpval Integer = Intg #0"
primrec "default_val (PrimT pt) = defpval pt"
        "default_val (RefT r ) = Null"

end

```

5 Term

```

(* Title:      isabelle/Bali/Term.thy
   ID:         $Id: Term.thy,v 1.43 2000/11/21 07:50:57 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

Java expressions and statements

design issues:

- * invocation frames for local variables could be reduced to special static objects (one per method). This would reduce redundancy, but yield a rather

- non-standard execution model more difficult to understand.

- * method bodies separated from calls to handle assumptions in axiomat. semantics

- NB: Body is intended to be in the environment of the `_called_` method.

- * class initialization is regarded as (auxiliary) statement (required for AxSem)

simplifications:

- * expression statement allowed for any expression

- * no unary, binary, etc, operators

- * This is modeled as a special non-assignable local variable

- * Super is modeled as a general expression with the same value as This

- * access to field x in current class via `This.x`

- * `NewA` creates only one-dimensional arrays;

- initialization of further subarrays may be simulated with nested `NewAs`

- * The 'Lit' constructor is allowed to contain a reference value.

- But this is assumed to be prohibited in the input language, which is enforced

- by the type-checking rules.

* a call of a static method via a type name may be simulated by a dummy variable
 * result expression instead of return statement (see Decl.thy)
 * no nested blocks with inner local variables
 * no synchronized statements
 * no secondary forms of if, while (e.g. no for) (may be easily simulated)
 * no switch, break, continue, no labels (may be simulated with while)
 * the try_catch_finally statement is divided into the try_catch statement and
 a finally statement, which may be considered as try..finally with empty catch
 * the try_catch statement has exactly one catch clause; multiple ones can be
 simulated with instanceof
 * the compiler is supposed to add the annotations {_} during type-checking. This
 transformation is left out as its result is checked by the type rules anyway
 *)
 Term = Value +

```

datatype inv_mode          (* invocation mode for method calls *)
*)
  = Static                 (* static *)
  | SuperM                 (* super *)
  | IntVir                 (* interface or virtual *)

```

```

types   sig                (* signature of a method, cf. 8.4.2 *)
        = "mname × ty list"  (* acutally belongs to Decl.thy *)

```

```

datatype var
  = LVar                   lname(* local variable (incl. parameters) *)
*)
  | FVar tname bool expr ename(*class field*)("{_,_}..."[10,10,85,99]90)
  | AVar      expr expr      (* array component *) ("_.[_]"[90,10]90)
and expr
  = NewC tname             (* class instance creation *)
  | NewA ty expr           (* array creation *) ("New _[_]"[99,10]85)
  | Cast ty expr          (* type cast *)
  | Inst expr ref_ty      (* instanceof *)      ("_ InstOf _"[85,99]85)
  | Lit val               (* literal value, references not allowed

```

```

*)
  | Super                (* special Super keyword *)
  | Acc var              (* variable access *)
  | Ass var expr         (* variable assign *) (":=" [90,85
]85)
  | Cond expr expr expr (* conditional *) (" ? _ : _" [85,85,80]80)
  | Call ref_ty ref_ty inv_mode expr mname (* method call *)
      (ty list) (expr list) ("_{_,_,_}.._'( {_}_' "[10,10,10,85,99,10,10]85)
  | Methd tname sig      (* (folded) method *)
  | Body tname stmt expr (* (unfolded) method body *)
and stmt
= Skip                  (* empty statement *)
| Expr expr             (* expression statement *)
| Comp stmt stmt        (";; _" [ 66,65]65)
| If_ expr stmt stmt   ("If'(_') _ Else _" [ 80,79,79]70)
| Loop expr stmt        ("While'(_') _" [ 80,79]70)
| Throw expr
| TryC stmt
      tname ename stmt ("Try _ Catch'(_ _') _" [79,99,80,79]70)
| Fin stmt stmt         ("_ Finally _" [ 79,79]70)
| init tname            (* class initialization *)

```

```
types term = "(expr+stmt, var, expr list) sum3"
```

```
translations
```

```

"sig"  <= (type) "mname × ty list"
"var"  <= (type) "Term.var"
"expr" <= (type) "Term.expr"
"stmt" <= (type) "Term.stmt"
"term" <= (type) "(expr+stmt, var, expr list) sum3"

```

```
syntax
```

```

this    :: expr
LAcc    :: "ename ⇒          expr" ("!!")
LAss    :: "ename ⇒ expr ⇒ stmt" (":==" [90,85] 85)
StatRef :: "ref_ty ⇒ expr"

```

```
translations
```

```

"this"    == "Acc (LVar This)"
"!!v"     == "Acc (LVar (Inl v))"
"v:==e"   == "Expr (Ass (LVar (Inl v)) e)"
"StatRef rt" == "Cast (RefT rt) (Lit Null)"

```

```

constdefs

  is_stmt :: "term  $\Rightarrow$  bool"
  "is_stmt t  $\equiv \exists c. t = \text{Inlr } c"$ 

end

```

6 Table

```

(* Title:      isabelle/Bali/Table.thy
   ID:         $Id: Table.thy,v 1.29 2000/11/27 15:20:15 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

Abstract tables and their implementation as lists

design issues:

```

* definition of table: infinite map vs. list vs. finite set
  list chosen, because:
  + a priori finite
  + lookup is more operational than for finite set
  - not very abstract, but function table converts it to abstract mapping
* coding of lookup result: Some/None vs. value/arbitrary
  Some/None chosen, because:
  ++ makes definedness check possible (applies also to finite set),
     which is important for the type standard, hiding/overriding, etc.
     (though it may perhaps be possible at least for the operational semantics
     to treat programs as infinite, i.e. where classes, fields, methods
etc.
     of any name are considered to be defined)
  - sometimes awkward case distinctions, alleviated by operator 'the'
*)

```

Table = Basis +

```

types ('a, 'b) table    (* table with key type 'a and contents type 'b
*)
  = "'a  $\rightsquigarrow$  'b"
  ('a, 'b) tables    (* non-unique table with key 'a and contents 'b
*)
  = "'a  $\Rightarrow$  'b set"

```

```

syntax
  table_of      :: "('a × 'b) list ⇒ ('a, 'b) table"    (* concrete
table *)

translations
  "table_of" == "map_of"

  (type)"'a ~> 'b"      <= (type)"'a ⇒ 'b Option.option"
  (type)"('a, 'b) table" <= (type)"'a ~> 'b"

consts
  Un_tables      :: "('a, 'b) tables set ⇒ ('a, 'b) tables"
  overrides      :: "('a, 'b) tables ⇒ ('a, 'b) tables ⇒
('a, 'b) tables"      (infixl "⊕⊕" 100)
  hidings_entails:: "('a, 'b) tables ⇒ ('a, 'c) tables ⇒
('b ⇒ 'c ⇒ bool) ⇒ bool"  ("_ hidings _ entails
_" 20)
  (* variant for unique table: *)
  hiding_entails :: "('a, 'b) table ⇒ ('a, 'c) table ⇒
('b ⇒ 'c ⇒ bool) ⇒ bool"  ("_ hiding _ entails
_" 20)

defs
  Un_tables_def      "Un_tables ts      ≡ λk. ⋃ t∈ts. t k"
  overrides_def      "s ⊕⊕ t          ≡ λk. if t k = {} then s k else
t k"
  hidings_entails_def "t hidings s entails R ≡ ∀k. ∀x∈t k. ∀y∈s k.
R x y"
  hiding_entails_def "t hiding s entails R ≡ ∀k. ∀x∈t k: ∀y∈s k:
R x y"

consts
  at_least_free :: "('a ~> 'b) => nat => bool"

primrec
  "at_least_free m 0      = True"
  "at_least_free m (Suc n) = (? a. m a = None & (!b. at_least_free (m(a|->b))
n))"

end

```

7 Decl

```
(* Title:      isabelle/Bali/Decl.thy
   ID:         $Id: Decl.thy,v 1.40 2000/11/27 15:20:15 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen
```

Field, method, interface, and class declarations, whole Java programs

simplifications:

```
* the only field and method modifier is static
* no constructors, which may be simulated by new + suitable methods
* there is just one global initializer per class, which can simulate all
  others

* no throws clause
* result statement replaced by result expression (evaluated at the end
  of the
  execution of the body; transformation is always possible (with goto,
  while)
* a void method is replaced by one that returns Unit (of dummy type Void)

* no interface modifiers yet, i.e. every interface is public
* no interface fields

* no class modifiers yet, i.e. every class is public, non-final
* every class has an explicit superclass (unused for Object)
* the (standard) methods of Object and of standard exceptions are not
  specified

* no packages
* no main method
*)
```

```
Decl = Term + Table + (** order is significant, because of clash for "var"
**)

```

```
types  modi = bool      (* modifier: static *)

```

```
field
  = "modi × ty"

```

```
fdecl      (* field declaration, cf. 8.3 *)
  = "ename × field"

```

```

translations
  "field" <= (type) "bool × ty"
  "fdecl" <= (type) "ename × field"

types (*sig: see Term.thy *)

  mhead          (* method head (excluding signature) *)
  = "modi × ename list × ty"
  (* modifier, parameter names, result type *)

  mbody          (* method body *)
  = "(ename × ty) list × stmt × expr"
  (* local variables, block+result expression *)

  methd          (* method in a class *)
  = "mhead × mbody"

  mdecl          (* method declaration in a class *)
  = "sig × methd"

```

```

translations
  "mhead" <= (type) "bool × ename list × ty"
  "mbody" <= (type) "(ename × ty) list × stmt × expr"
  "methd" <= (type) "mhead × mbody"
  "mdecl" <= (type) "sig × methd"

```

syntax

```

static    :: "modi ⇒ bool"
mrt       :: "mhead ⇒ ty"

```

translations

```

"static"    => "id"
"mrt mh"    => "snd (snd mh)"

```

```

types  ibody          (* interface body *)
      = "(sig × mhead) list"
      (* methods *)

      iface          (* interface *)
      = "tname list × ibody"
      (* superinterface list *)

```

```

idecl          (* interface declaration, cf. 9.1 *)
= "tname × iface"

translations
"ibody" <= (type) "(sig × mhead) list"
"iface" <= (type) "tname list × ibody"
"idecl" <= (type) "tname × iface"

types  cbody          (* class body *)
= "fdecl list × mdecl list × stmt"
(* fields, methods, initializer *)

class          (* class *)
= "tname × tname list × cbody"
(* superclass, implemented interfaces *)

cdecl          (* class declaration, cf. 8.1 *)
= "tname × class"

translations
"cbody" <= (type) "fdecl list × mdecl list × stmt"
"class" <= (type) "tname × tname list × cbody"
"cdecl" <= (type) "tname × class"

consts

Object_mdecls  :: "mdecl list" (* methods of Object *)
SXcpt_mdecls   :: "mdecl list" (* methods of SXcpts *)
ObjectC ::      "cdecl"      (* declaration of root      class *)
SXcptC  :: "xname ⇒ cdecl"   (* declarations of throwable classes
*)
standard_classes :: cdecl list

defs

ObjectC_def "ObjectC ≡ (Object , (arbitrary , [], [], Object_mdecls, Skip))"
SXcptC_def  "SXcptC xn ≡ (SXcpt xn, (if xn = Throwable then Object else
SXcpt Throwable, [], [], SXcpt_mdecls, Skip))"
standard_classes_def "standard_classes ≡ [ObjectC, SXcptC Throwable,
SXcptC NullPointerException, SXcptC OutOfMemory, SXcptC ClassCast,
SXcptC NegArrSize , SXcptC IndOutBound, SXcptC ArrStore]"

```

```

(* programs *)
types prog =      "idecl list × cdecl list"
translations
  "prog"<= (type) "idecl list × cdecl list"

syntax
  iface      :: "prog ⇒ (tname, iface) table"
  class      :: "prog ⇒ (tname, class) table"
  is_iface   :: "prog ⇒ tname ⇒ bool"
  is_class   :: "prog ⇒ tname ⇒ bool"

translations
  "iface G I" == "table_of (fst G) I"
  "class G C" == "table_of (snd G) C"
  "is_iface G I" == "iface G I ≠ None"
  "is_class G C" == "class G C ≠ None"

consts
  is_type :: "prog ⇒ ty ⇒ bool"
  isrtype :: "prog ⇒ ref_ty ⇒ bool"

primrec "is_type G (PrimT pt) = True"
  "is_type G (RefT rt) = isrtype G rt"
  "isrtype G (NullT _) = True"
  "isrtype G (IfaceT tn) = is_iface G tn"
  "isrtype G (ClassT tn) = is_class G tn"
  "isrtype G (ArrayT T) = is_type G T"

(* subinterface and subclass relation, in anticipation of TypeRel.thy *)
consts

  subint1,
  subcls1 :: "prog ⇒ (tname × tname) set"

defs
  subint1_def  "subint1 G ≡ {(I,J). ∃ i∈iface G I: J∈set (fst i)}"
  subcls1_def  "subcls1 G ≡ {(C,D). C≠Object ∧ (∃ c∈class G C: fst c = D)}"

(* well-structured programs *)
constdefs

```

```

ws_idecl :: "prog ⇒ tname ⇒ tname list ⇒ bool"
"ws_idecl G I si ≡ ∀ J ∈ set si. is_iface G J ∧ (J,I) ∉ (subint1 G)^+"

ws_cdecl :: "prog ⇒ tname ⇒ tname ⇒ bool"
"ws_cdecl G C sc ≡ C ≠ Object → is_class G sc ∧ (sc,C) ∉ (subcls1 G)^+"

ws_prog  :: "prog ⇒ bool"
"ws_prog G ≡ (∀ (I,(si,ib)) ∈ set (fst G). ws_idecl G I si) ∧
              (∀ (C,(sc,cb)) ∈ set (snd G). ws_cdecl G C sc)"

(* auxiliary well-founded relation for the recursion operators below *)
constdefs
  ws_wfrel :: "(prog ⇒ (tname × tname) set) ⇒
              ((prog × tname) × (prog × tname)) set"
"ws_wfrel R ≡ {((G,T),(G',T')). G' = G ∧ ws_prog G ∧ (T',T) ∈ R G}"

(* general operators for recursion over the interface and class hierarchies *)
consts
  iface_rec  :: "prog × tname ⇒ (tname ⇒ ibody ⇒ 'a set ⇒
'a) ⇒ 'a"
  class_rec  :: "prog × tname ⇒ 'a ⇒ (tname ⇒ cbody ⇒ 'a ⇒
'a) ⇒ 'a"

recdef iface_rec "ws_wfrel subint1" congr image_cong
"iface_rec (G,I) = (λf. case iface G I of None ⇒ arbitrary | Some (si,ib)
⇒
    if ws_prog G then f I ib ((λJ. iface_rec (G,J) f) ` set
    si)
    else arbitrary)"

recdef class_rec "ws_wfrel subcls1"
"class_rec(G,C) = (λt f. case class G C of None ⇒ arbitrary | Some (sc,si,cb) ⇒
    if ws_prog G then f C cb (if C = Object then t else class_rec (G,sc)
    t f)
    else arbitrary)"

types
  fspec = "ename × tname"
consts
  imethds  :: "prog ⇒ tname ⇒ ( sig , tname × mhead) tables"
  cmethd   :: "prog ⇒ tname ⇒ ( sig , tname × methd) table"
  fields   :: "prog ⇒ tname ⇒ ((ename × tname) × field) list"
  cfield   :: "prog ⇒ tname ⇒ ( ename , tname × field) table"

```

```

defs
  (* methods of an interface, with overriding and inheritance, cf. 9.2 *)
  imeths_def "imethds G I  $\equiv$  iface_rec (G,I)      ( $\lambda I$       ms      ts.

              (Un_tables ts)  $\oplus\oplus$  (o2s  $\circ$  table_of (map ( $\lambda(s,m)$ . (s,I,m)) ms)))"

  (* methods of a class, with inheritance, overriding and hiding, cf.
  8.4.6 *)
  cmethd_def "cmethd G C  $\equiv$  class_rec (G,C) empty ( $\lambda C$  (fs,ms,ini) ts.
              ts ++ table_of (map ( $\lambda(s,m)$ . (s,C,m)) ms))"

  (* list of fields of a class, including inherited and hidden ones *)
  fields_def "fields G C  $\equiv$  class_rec (G,C) []      ( $\lambda C$  (fs,ms,ini) ts.
              map ( $\lambda(n,t)$ . ((n,C),t)) fs
  @ ts)"

  (* fields of a class, with inheritance and hiding, cf. 8.3 *)
  cfield_def "cfield G C  $\equiv$  table_of((map ( $\lambda((n,d),T)$ . (n, (d,T)))) (fields
  G C))"

constdefs
  is_methd :: "prog  $\Rightarrow$  tname  $\Rightarrow$  sig  $\Rightarrow$  bool"
  "is_methd G  $\equiv$   $\lambda C$  sig. is_class G C  $\wedge$  cmethd G C sig  $\neq$  None"

end

```

8 TypeRel

```

(* Title:      isabelle/Bali/TypeRel.thy
   ID:         $Id: TypeRel.thy,v 1.32 2000/11/28 23:06:20 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

The relations between Java types

simplifications:

* subinterface, subclass and widening relation includes identity

improvements over Java Specification 1.0:
 * narrowing reference conversion also in cases where the return types
 of a pair
 of methods common to both types are in widening (rather identity) relation
 * one could add similar constraints also for other cases

design issues:

* the type relations do not require `is_type` for their arguments
 * the `subint1` and `subcls1` relations imply `is_iface/is_class` for their
 first
 arguments, which is required for their finiteness

*)

TypeRel = Decl +

consts

```
(*subint1, in Decl.thy*)           (* direct subinterface *)
(*subint,  by translation*)         (*      subinterface      *)
(*subcls1, in Decl.thy*)           (* direct subclass      *)
(*subcls,  by translation*)         (*      subclass        *)
  implmt1,                          (* direct implementation *)
*)
  implmt  :: "prog => (tname × tname) set" (*      implementation *)
  widen,   (*      widening      *)
  narrow,  (*      narrowing     *)
  cast     :: "prog => (ty    × ty    ) set" (*      casting         *)
```

syntax

```
"@subint1" :: "prog => [tname, tname] => bool" ("|_|-<:I1_" [71,71,71]
70)
"@subint"  :: "prog => [tname, tname] => bool" ("|_|-<=:I_" [71,71,71]
70)
"@subcls1" :: "prog => [tname, tname] => bool" ("|_|-<:C1_" [71,71,71]
70)
"@subcls"  :: "prog => [tname, tname] => bool" ("|_|-<=:C_" [71,71,71]
70)
"@implmt1" :: "prog => [tname, tname] => bool" ("|_|-~>1_" [71,71,71]
70)
"@implmt"  :: "prog => [tname, tname] => bool" ("|_|-~>_" [71,71,71]
70)
"@widen"   :: "prog => [ty    , ty    ] => bool" ("|_|-<=:_" [71,71,71]
```

```

70)
"@narrow" :: "prog => [ty , ty ] => bool" ("_|-_:>_" [71,71,71]
70)
"@cast"   :: "prog => [ty , ty ] => bool" ("_|-_<=? _"[71,71,71]
70)

```

syntax (symbols)

```

"@subint1" :: "prog => [tname, tname] => bool" ("_|-_<I1_" [71,71,71]
70)
"@subint"  :: "prog => [tname, tname] => bool" ("_|-_<I _" [71,71,71]
70)
"@subcls1" :: "prog => [tname, tname] => bool" ("_|-_<C1_" [71,71,71]
70)
"@subcls"  :: "prog => [tname, tname] => bool" ("_|-_<C _" [71,71,71]
70)
"@implmt1" :: "prog => [tname, tname] => bool" ("_|-_~>1_" [71,71,71]
70)
"@implmt"  :: "prog => [tname, tname] => bool" ("_|-_~>_" [71,71,71]
70)
"@widen"   :: "prog => [ty , ty ] => bool" ("_|-_<_" [71,71,71]
70)
"@narrow"  :: "prog => [ty , ty ] => bool" ("_|-_>_" [71,71,71]
70)
"@cast"    :: "prog => [ty , ty ] => bool" ("_|-_<? _" [71,71,71]
70)

```

translations

```

"G⊢I <I1 J" == "(I,J) ∈ subint1 G"
"G⊢I <I J"  == "(I,J) ∈ (subint1 G)^*" (* cf. 9.1.3 *)
"G⊢C <C1 D" == "(C,D) ∈ subcls1 G"
"G⊢C <C D"  == "(C,D) ∈ (subcls1 G)^*" (* cf. 8.1.3 *)
"G⊢C ~>1 I" == "(C,I) ∈ implmt1 G"
"G⊢C ~> I"  == "(C,I) ∈ implmt G"
"G⊢S < T"   == "(S,T) ∈ widen G"
"G⊢S > T"   == "(S,T) ∈ narrow G"
"G⊢S <? T"  == "(S,T) ∈ cast G"

```

defs

```

(* direct subinterface in Decl.thy, cf. 9.1.3 *)
(* direct subclass      in Decl.thy, cf. 8.1.3 *)

```

```
(* direct implementation, cf. 8.1.3 *)
implmt1_def "implmt1 G≡{(C,I). C≠Object ∧ (∃c∈class G C: I∈set (fst(snd
c)))}"
```

```
inductive "implmt G" intrs (* cf. 8.1.4 *)
```

```
direct      "G⊢C↔IJ      ⇒ G⊢C↔J"
subint      "[[G⊢C↔I; G⊢I⊆I J]] ⇒ G⊢C↔J"
subcls1     "[[G⊢C⊆C1D; G⊢D↔J]] ⇒ G⊢C↔J"
```

```
inductive "widen G" intrs (*widening, viz. method invocation conversion,
cf. 5.3
```

```
                                i.e. kind of syntactic subtyping *)
refl      "G⊢      T⊆T" (*identity conversion, cf.
5.1.1 *)
subint    "G⊢I⊆I J      ⇒ G⊢ Iface I⊆ Iface J"(*wid.ref.conv.,
cf. 5.1.4 *)
int_obj   "G⊢ Iface I⊆ Class Object"
subcls    "G⊢C⊆C D      ⇒ G⊢ Class C⊆ Class D"
implmt    "G⊢C↔I      ⇒ G⊢ Class C⊆ Iface I"
null      "G⊢      NT⊆ RefT R"
arr_obj   "G⊢      T.[]⊆ Class Object"
array     "G⊢RefT S⊆RefT T ⇒ G⊢RefT S.[]⊆ RefT T.[]"
```

```
(* all properties of narrowing and casting conversions we actually need
*)
```

```
(* these can easily be proven from the definitions below *)
```

```
(*
```

```
rules
```

```
cast_RefT2 "G⊢S⊆? RefT R ⇒ ∃t. S=RefT t"
cast_PrimT2 "G⊢S⊆? PrimT pt ⇒ ∃t. S=PrimT t ∧ G⊢PrimT t⊆PrimT
pt"
*)
```

```
constdefs
```

```
widens :: "prog ⇒ [ty list, ty list] ⇒ bool" ("⊢_⊆_" [71,71,71]
70)
"G⊢Ts[⊆]Ts' ≡ list_all2 (λT T'. G⊢T⊆T') Ts Ts'"
```

```
(* more detailed than necessary for type-safety, see above rules. *)
```

inductive "narrow G" intrs (* narrowing reference conversion, cf. 5.1.5 *)

```

subcls "G ⊢ C ⊆ C D" ⇒ G ⊢ Class D ⊆ Class C"
implmt "¬G ⊢ C ⊆ I" ⇒ G ⊢ Class C ⊆ Iface I"
obj_arr "G ⊢ Class Object ⊆ T.[]"
int_cls "G ⊢ Iface I ⊆ Class C"
subint "imethds G I hidings imethds G J entails
        (λ(md, mh) (md', mh')). G ⊢ mrt mh ⊆ mrt mh' ⇒
        ¬G ⊢ I ⊆ I J ⇒ G ⊢ Iface I ⊆ Iface J"
array "G ⊢ RefT S ⊆ RefT T ⇒ G ⊢ RefT S.[] ⊆ RefT T.[]"

```

inductive "cast G" intrs (* casting conversion, cf. 5.5 *)

```

widen "G ⊢ S ⊆ T ⇒ G ⊢ S ⊆? T"
narrow "G ⊢ S ⊆ T ⇒ G ⊢ S ⊆? T"

```

end

9 WellType

```

(* Title:      isabelle/Bali/WellType.thy
   ID:         $Id: WellType.thy,v 1.43 2000/11/28 23:06:20 oheimb Exp
   $

```

```

   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

Well-typedness of Java programs

improvements over Java Specification 1.0:

* methods of *Object* can be called upon references of interface or array type

simplifications:

* the type rules include all static checks on statements and expressions, e.g.

definedness of names (of parameters, locals, fields, methods)

design issues:

* unified type judgment for statements, variables, expressions, expression lists

* statements are typed like expressions with dummy type Void
 * the typing rules take an extra argument that is capable of determining
 the dynamic type of objects. Therefore, they can be used for both
 checking static types and determining runtime types in transition semantics.
 *)

WellType = TypeRel +

```
types  lenv
      = "(lname, ty) table"  (* local variables, including This *)

      env
      = "prog × lenv"        (* program, locals *)
```

```
syntax
  prg    :: "env ⇒ prog"
  lcl    :: "env ⇒ lenv"
```

```
translations
  "lenv" <= (type) "(lname, ty) table"
  "env"  <= (type) "prog × lenv"
  "prg"  => "fst"
  "lcl"  => "snd"
```

```
types
  emhead = "ref_ty × mhead"

consts
  cmheads :: "prog ⇒ tname ⇒ sig ⇒ emhead set"
  mheads  :: "prog ⇒ ref_ty ⇒ sig ⇒ emhead set"

defs
  cmheads_def
  "cmheads G C ≡ λsig. (λ(C,(h,b)). (ClassT C,h)) ‘‘ o2s (cmethd G C sig)''
primrec
  "mheads G NullT = (λsig. {})"
  "mheads G (IfaceT I) = (λsig. (λ(I,h).(IfaceT I,h)) ‘‘ imethds G I sig
  ∪
      cmheads G Object sig)''
  "mheads G (ClassT C) = cmheads G C"
  "mheads G (ArrayT T) = cmheads G Object"
```

(* more detailed than necessary for type-safety, see below. *)

constdefs

(* applicable methods, cf. 15.11.2.1 *)

```

appl_methds  :: "prog ⇒ ref_ty ⇒ sig ⇒ (emhead × ty list)  set"
"appl_methds G rt ≡ λ(mn, pTs). {(mh,pTs') |mh pTs'.
                               mh ∈ mheads G rt (mn, pTs') ∧ G⊢pTs[⊆]pTs'}"

(* more specific methods, cf. 15.11.2.2 *)
more_spec    :: "prog ⇒ emhead × ty list ⇒ emhead × ty list ⇒
bool"
"more_spec G ≡ λ(mh,pTs). λ(mh',pTs'). G⊢pTs[⊆]pTs'"
(*more_spec G ≡ λ((d,h),pTs). λ((d',h'),pTs'). G⊢RefT d⊆RefT d'∧G⊢pTs[⊆]pTs'*)

(* maximally specific methods, cf. 15.11.2.2 *)
max_spec     :: "prog ⇒ ref_ty ⇒ sig ⇒ (emhead × ty list)  set"

" max_spec G rt sig ≡{m. m ∈appl_methds G rt sig ∧
                      (∀m'∈appl_methds G rt sig. more_spec G m' m →
m'=m)}"
(*
rules (* all properties of more_spec, appl_methods and max_spec we actually
need

      these can easily be proven from the above definitions *)

max_spec2mheads "max_spec G rt (mn, pTs) = insert (mh, pTs') A ⇒
                mh∈mheads G rt (mn, pTs') ∧ G⊢pTs[⊆]pTs'"
*)

constdefs
  empty_dt  :: "dyn_ty"
  "empty_dt ≡ λa. None"

  invmode  :: "modi ⇒ expr ⇒ inv_mode"
  "invmode m e ≡ if static m then Static else if e=Super then SuperM else
IntVir"

types tys =      "ty + ty list"
translations
  "tys"  <= (type) "ty + ty list"

consts
  wt      :: "(env × dyn_ty × term × tys) set"
(*wt     :: " env ⇒ dyn_ty ⇒ (term × tys) set" not feasible because
of

                                                changing env in Try
stmt *)

```

```

syntax
wt      :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [term,tys]  $\Rightarrow$  bool" ("_,_|=:::" [51,51,51,51]
50)
wt_stmt :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  stmt  $\Rightarrow$  bool" ("_,_|=:<>" [51,51,51
] 50)
ty_expr :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [expr ,ty ]  $\Rightarrow$  bool" ("_,_|=:-_" [51,51,51,51]
50)
ty_var  :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [var ,ty ]  $\Rightarrow$  bool" ("_,_|=:=_" [51,51,51,51]
50)
ty_exprs:: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [expr list,
ty list]  $\Rightarrow$  bool" ("_,_|=:#_" [51,51,51,51]
50)

```

```

syntax (xsymbols)
wt      :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [term,tys]  $\Rightarrow$  bool" ("_,_|=:::" [51,51,51,51]
50)
wt_stmt :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  stmt  $\Rightarrow$  bool" ("_,_|=:: $\surd$ " [51,51,51
] 50)
ty_expr :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [expr ,ty ]  $\Rightarrow$  bool" ("_,_|=:-_" [51,51,51,51]
50)
ty_var  :: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [var ,ty ]  $\Rightarrow$  bool" ("_,_|=:=_" [51,51,51,51]
50)
ty_exprs:: "env  $\Rightarrow$  dyn_ty  $\Rightarrow$  [expr list,
ty list]  $\Rightarrow$  bool" ("_,_|=:: $\doteq$ _" [51,51,51,51]
50)

```

```

translations
"E,dt|t::T" == "(E,dt,t,T)  $\in$  wt"
"E,dt|s:: $\surd$ " == "E,dt|In1r s::In1 (PrimT Void)"
"E,dt|e::-T" == "E,dt|In1l e::In1 T"
"E,dt|e::=T" == "E,dt|In2 e::In1 T"
"E,dt|e:: $\doteq$ T" == "E,dt|In3 e::Inr T"

```

```

syntax (* for purely static typing *)
wt_      :: "env  $\Rightarrow$  [term,tys]  $\Rightarrow$  bool" ("_|-:::" [51,51,51] 50)
wt_stmt_ :: "env  $\Rightarrow$  stmt  $\Rightarrow$  bool" ("_|-:<>" [51,51 ] 50)
ty_expr_ :: "env  $\Rightarrow$  [expr ,ty ]  $\Rightarrow$  bool" ("_|-:-_" [51,51,51] 50)
ty_var_  :: "env  $\Rightarrow$  [var ,ty ]  $\Rightarrow$  bool" ("_|-:=_" [51,51,51] 50)
ty_exprs_:: "env  $\Rightarrow$  [expr list,
ty list]  $\Rightarrow$  bool" ("_|-:#_" [51,51,51] 50)

```

```

syntax (xsymbols)
wt_      :: "env  $\Rightarrow$  [term,tys]  $\Rightarrow$  bool" ("_|-:::" [51,51,51] 50)
wt_stmt_ :: "env  $\Rightarrow$  stmt  $\Rightarrow$  bool" ("_|-:: $\surd$ " [51,51 ] 50)

```

```

ty_expr_ :: "env  $\Rightarrow$  [expr ,ty ]  $\Rightarrow$  bool" ("_ $\vdash$ _:: $\_$ " [51,51,51] 50)
ty_var_  :: "env  $\Rightarrow$  [var  ,ty ]  $\Rightarrow$  bool" ("_ $\vdash$ _:: $\_$ " [51,51,51] 50)
ty_exprs_:: "env  $\Rightarrow$  [expr list,
                    ty  list]  $\Rightarrow$  bool" ("_ $\vdash$ _:: $\_$ " [51,51,51] 50)

```

translations

```

"E $\vdash$ t:: T" == "E,empty_dt $\vdash$ t:: T"
"E $\vdash$ s:: $\checkmark$ " == "E,empty_dt $\vdash$ s:: $\checkmark$ "
"E $\vdash$ e:: $\neg$ T" == "E,empty_dt $\vdash$ e:: $\neg$ T"
"E $\vdash$ e::T" == "E,empty_dt $\vdash$ e::T"
"E $\vdash$ e:: $\doteq$ T" == "E,empty_dt $\vdash$ e:: $\doteq$ T"

```

inductive wt intrs

(* well-typed statements *)

```

Skip "E,dt $\vdash$ Skip:: $\checkmark$ "

```

```

Expr "[E,dt $\vdash$ e:: $\neg$ T]  $\implies$ 
      E,dt $\vdash$ Expr e:: $\checkmark$ "

```

```

Comp "[E,dt $\vdash$ c1:: $\checkmark$ ;
      E,dt $\vdash$ c2:: $\checkmark$ ]  $\implies$ 
      E,dt $\vdash$ c1;; c2:: $\checkmark$ "

```

(* cf. 14.8 *)

```

If "[E,dt $\vdash$ e:: $\neg$ PrimT Boolean;
    E,dt $\vdash$ c1:: $\checkmark$ ;
    E,dt $\vdash$ c2:: $\checkmark$ ]  $\implies$ 
    E,dt $\vdash$ If(e) c1 Else c2:: $\checkmark$ "

```

(* cf. 14.10 *)

```

Loop "[E,dt $\vdash$ e:: $\neg$ PrimT Boolean;
      E,dt $\vdash$ c:: $\checkmark$ ]  $\implies$ 
      E,dt $\vdash$ While(e) c:: $\checkmark$ "

```

(* cf. 14.16 *)

```

Throw "[E,dt $\vdash$ e:: $\neg$ Class tn;
        prg E $\vdash$ tn $\leq$ C SXcpt Throwable]  $\implies$ 
        E,dt $\vdash$ Throw e:: $\checkmark$ "

```

(* cf. 14.18 *)

```

Try "[E,dt $\vdash$ c1:: $\checkmark$ ; prg E $\vdash$ tn $\leq$ C SXcpt Throwable;
     lcl E (EName vn)=None; (prg E,lcl E(EName vn $\mapsto$ Class tn)),dt $\vdash$ c2:: $\checkmark$ ]
 $\implies$ 

```

```

E,dt|=Try c1 Catch(tn vn) c2::√"

(* cf. 14.18 *)
Fin  "[E,dt|=c1::√; E,dt|=c2::√] ==>
E,dt|=c1 Finally c2::√"

Init "[is_class (prg E) C] ==>
E,dt|=init C::√"

(* well-typed expressions *)

(* cf. 15.8 *)
NewC "[is_class (prg E) C] ==>
E,dt|=NewC C::-Class C"

(* cf. 15.9 *)
NewA "[is_type (prg E) T;
      E,dt|=i::-PrimT Integer] ==>
E,dt|=New T[i]::-T. []"

(* cf. 15.15 *)
Cast "[E,dt|=e::-T; is_type (prg E) T';
      prg E ⊢ T ≤? T'] ==>
E,dt|=Cast T' e::-T"

(* cf. 15.19.2 *)
Inst "[E,dt|=e::-RefT T;
      prg E ⊢ RefT T ≤? RefT T'] ==>
E,dt|=e InstOf T'::-PrimT Boolean"

(* cf. 15.7.1 *)
Lit  "[typeof dt x = Some T] ==>
E,dt|=Lit x::-T"

(* cf. 15.10.2, 15.11.1 *)
Super "[lcl E This = Some (Class C); C ≠ Object;
      class (prg E) C = Some (D, rest)] ==>
E,dt|=Super::-Class D"

(* cf. 15.13.1, 15.10.1, 15.12 *)
Acc  "[E,dt|=va::-T] ==>
E,dt|=Acc va::-T"

```

```

(* cf. 15.25, 15.25.1 *)
Ass  "[E,dt|=va::=T; va ≠ LVar This;
      E,dt|=v ::-T';
      prg E⊢T' ≤T] ⇒
                                           E,dt|=va:=v::-T'"

(* cf. 15.24 *)
Cond "[E,dt|=e0::-PrimT Boolean;
      E,dt|=e1::-T1; E,dt|=e2::-T2;
      prg E⊢T1 ≤T2 ∧ T = T2 ∨ prg E⊢T2 ≤T1 ∧ T = T1] ⇒
                                           E,dt|=e0 ? e1 : e2::-T"

(* cf. 15.11.1, 15.11.2, 15.11.3 *)
Call "[E,dt|=e::-RefT t;
      E,dt|=ps::-pTs;
      max_spec (prg E) t (mn, pTs) = {((md,(m,pns,rT)),pTs')}] ⇒
                                           E,dt|={t,md,invmode m e}e..mn({pTs'}ps)::-rT"

Methd "[is_class (prg E) C;
        cmethd (prg E) C sig = Some (md,mh,lvars,blk,res);
        E,dt|=Body md blk res::-T] ⇒
                                           E,dt|=Methd C sig::-T"

Body "[is_class (prg E) D;
      E,dt|=blk::√;
      E,dt|=res::-T] ⇒
                                           E,dt|=Body D blk res::-T"

(* well-typed variables *)

(* cf. 15.13.1 *)
LVar "[lcl E vn = Some T; is_type (prg E) T] ⇒
                                           E,dt|=LVar vn::=T"

(* cf. 15.10.1 *)
FVar "[E,dt|=e::-Class C;
      cfield (prg E) C fn = Some (fd,(m,fT))] ⇒
                                           E,dt|={fd,static m}e..fn::=fT"

(* cf. 15.12 *)
AVar "[E,dt|=e::-T. [];
      E,dt|=i::-PrimT Integer] ⇒
                                           E,dt|=e.[i]::=T"

(* well-typed expression lists *)

```

```

(* cf. 15.11.??? *)
Nil                                     "E,dt|= [] ::≐ []"

(* cf. 15.11.??? *)
Cons "[E,dt|=e ::-T;
      E,dt|=es::≐Ts] ==>
                                           E,dt|=e#es::≐T#Ts"

end

```

10 WellForm

```

(* Title:      isabelle/Bali/WellForm.thy
   ID:         $Id: WellForm.thy,v 1.32 2000/11/27 15:20:16 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

Well-formedness of Java programs
for static checks on expressions and statements, see *WellType.thy*

improvements over Java Specification 1.0 (cf. 8.4.6.3, 8.4.6.4, 9.4.1):
* a method implementing or overwriting another method may have a result type
that widens to the result type of the other method (instead of identical type)
* if an interface inherits more than one method with the same signature, the
methods need not have identical return types

simplifications:
* Object and standard exceptions are assumed to be declared like normal classes
*)
WellForm = WellType +

```

consts
wf_fdecl :: "prog =>          fdecl => bool"
wf_mhead :: "prog => sig    => mhead => bool"
wf_mdecl :: "prog => tname => mdecl => bool"
wf_idecl :: "prog =>          idecl => bool"

```

```

wf_cdecl :: "prog ⇒ cdecl ⇒ bool"
wf_prog  :: "prog ⇒ bool"

defs
  (* well-formed field declaration (common part for classes and interfaces),
     cf. 8.3 and (9.3) *)
  wf_fdecl_def "wf_fdecl G ≡ λ(fn,(m,ft)). is_type G ft"

  (*well-formed method declaration,cf. 8.4, 8.4.1, 8.4.3, 8.4.5, 14.3.2,
     (9.4)*)
  (* cf. 14.15, 15.7.2, for scope issues cf. 8.4.1 and 14.3.2 *)
  wf_mhead_def "wf_mhead G ≡ λ(mn,pTs) (m,pns,rT). length pTs = length
pns ∧
  ( ∀T∈set pTs. is_type G T) ∧ is_type
G rT ∧
  nodups pns"

  wf_mdecl_def "wf_mdecl G C ≡ λ((mn,pTs),(m,pns,rT),lvars,blk,res).
  wf_mhead G (mn,pTs) (m,pns,rT) ∧ unique lvars
  ∧
  (C=Object ⟶ ¬static m) ∧ (∀(vn,T)∈set lvars. is_type
G T) ∧
  (∀pn∈set pns. table_of lvars pn = None) ∧
  (∃T. (G,table_of lvars(pns[↦]pTs) (+)
  (if static m then empty else empty((()↦Class C))))⊢
  Body C blk res::-T ∧ G⊢T⊆rT)"

  (* well-formed interface declaration, cf. 9.1, 9.1.2.1, 9.1.3, 9.4 *)
  wf_idecl_def "wf_idecl G ≡ λ(I,(si,ms)). ws_idecl G I si ∧
  ¬is_class G I ∧
  (∀(sig,mh)∈set ms. wf_mhead G sig mh ∧ ¬static (fst
mh)) ∧
  unique ms ∧
  (o2s ∘ table_of ms hidings Un_tables((λJ.(imethds G J))‘set
si)
  entails (λmh (md,mh'). G⊢mrt mh⊆mrt mh'))"

  (* well-formed class declaration, cf. 8.1, 8.1.2.1, 8.1.2.2, 8.1.3,
     8.1.4 and
     class method declaration, cf. 8.4.3.3, 8.4.6.1, 8.4.6.2, 8.4.6.3, 8.4.6.4
     *)
  wf_cdecl_def "wf_cdecl G ≡ λ(C,(sc,si,fs,ms,init)).
  ¬is_iface G C ∧
  (∀I∈set si. is_iface G I ∧

```

```

      (∀s. ∀(md', mh' ) ∈ imethds G I s.
        (∃(md ,(mh ,b)) ∈ cmethod G C s: G⊢mrt mh ≲mrt
mh' ∧
                                                                    ¬static (fst
mh)))) ∧
      (∀f∈set fs. wf_fdecl G f) ∧ unique fs ∧
      (∀m∈set ms. wf_mdecl G C m) ∧ unique ms ∧
      (G,empty)⊢init::√ ∧ ws_cdecl G C sc ∧
      (C ≠ Object → (table_of ms hiding cmethod G sc entails
        (λ(mh,b) (md',(mh',b')). G⊢mrt mh ≲mrt
mh' ∧
                                                                    static (fst mh') = static (fst
mh))))"

(* well-formed program, cf. 8.1, 9.1 *)
wf_prog_def  "wf_prog G ≡ let is = fst G; cs = snd G in
              ObjectC ∈ set cs ∧ (∀xn. SXcptC xn ∈ set cs)
∧
              (∀i∈set is. wf_idecl G i) ∧ unique is ∧
              (∀c∈set cs. wf_cdecl G c) ∧ unique cs"

end

```

11 State

```

(* Title:      isabelle/Bali/State.thy
   ID:         $Id: State.thy,v 1.63 2000/11/23 09:57:31 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

State for evaluation of Java expressions and statements

design issues:

```

* all kinds of objects (class instances, arrays, and class objects)
  are handled via a general object abstraction
* the heap and the map for class objects are combined into a single table
  (recall (loc, obj) table × (tname, obj) table  ~ = (loc + tname, obj)
table)

```

simplifications:

```

*)

```

```

State = TypeRel +

datatype obj_tag =      (* tag for generic object   *)
                    CInst tname (* class instance           *)
                    | Arr ty int (* array with component type and length *)
                    (* | CStat      the tag is irrelevant for a class object,
                               i.e. the static fields of a class *)

types vn = "fspec + int" (* variable name      *)
      obj = "obj_tag × (vn, val) table" (* generalized object
*)

constdefs

  the_Arr :: "obj option ⇒ ty × int × (vn, val) table"
  "the_Arr obj ≡ ε(T,k,t). obj = Some (Arr T k,t)"

  upd_obj      :: "vn ⇒ val ⇒ obj ⇒ obj"
  "upd_obj n v ≡ λ(oi,vs). (oi,vs(n↦v))"

  obj_ty      :: "obj ⇒ ty"
  "obj_ty obj ≡ case fst obj of CInst C ⇒ Class C | Arr T k ⇒ T.[]"

  obj_class :: "obj ⇒ tname"
  "obj_class obj ≡ case fst obj of CInst C ⇒ C | Arr T k ⇒ Object"

types oref = "loc + tname" (* generalized object reference *)
syntax
  Heap :: "loc ⇒ oref"
  Stat :: "tname ⇒ oref"

translations
  "Heap" => "Inl"
  "Stat" => "Inr"

constdefs
  fields_table :: "prog ⇒ tname ⇒ (fspec ⇒ field ⇒ bool) ⇒ (fspec,
ty) table"
  "fields_table G C P ≡ option_map snd ∘ table_of (filter (split P) (fields
G C))"

  in_bounds :: "int ⇒ int ⇒ bool" (* ("(/ in'_bounds _) [50,

```

```

51] 50)
"i in_bounds k ≡ 0 ≤ i ∧ i < k"

arr_comps :: "'a ⇒ int ⇒ int ⇒ 'a option"
"arr_comps T k ≡ λi. if i in_bounds k then Some T else None"

var_tys      :: "prog ⇒ obj_tag ⇒ oref ⇒ (vn, ty) table"
"var_tys G oi r ≡ case r of Heap a ⇒ (case oi of
      CInst C ⇒ fields_table G C (λn (m,fT). ¬static m) (+) empty
      | Arr T k ⇒ empty (+) arr_comps T k
      | Stat C ⇒ fields_table G C
      (λ(fn,fd) (m,fT). fd = C ∧ static m)
(+ ) empty"

types  globs      (* global variables: heap and static variables
*)
      = "(oref , obj) table"
      heap
      = "(loc , obj) table"
      locals
      = "(lname, val) table" (* local variables *)

datatype st = (* pure state, i.e. contents of all variables *)
      st globs locals

constdefs

globs  :: "st ⇒ globs"
"globs ≡ st_case (λg l. g)"

locals :: "st ⇒ locals"
"locals ≡ st_case (λg l. l)"

heap   :: "st ⇒ heap"
"heap s ≡ globs s ◦ Heap"

new_Addr      :: "heap ⇒ loc option"
"new_Addr h ≡ if (∃a. h a ≠ None) then None else Some (εa. h a = None)"

syntax
val_this      :: "st ⇒ val"

```

```

lookup_obj  :: "st ⇒ val ⇒ obj"

translations
"val_this s"      == "the (locals s This)"
"lookup_obj s a'" == "the (heap s (the_Addr a'))"

syntax

init_vals      :: "('a, ty) table ⇒ ('a, val) table"

translations
"init_vals vs"   == "option_map default_val ◦ vs"

constdefs
gupd           :: "oref ⇒ obj ⇒ st ⇒ st"          ("gupd'(_↦_)"[10,10]1000)
"gupd r obj ≡ st_case (λg l. st (g(r↦obj)) l)"

lupd          :: "lname ⇒ val ⇒ st ⇒ st"          ("lupd'(_↦_)"[10,10]1000)
"lupd vn v ≡ st_case (λg l. st g (l(vn↦v)))"

upd_gobj      :: "oref ⇒ vn ⇒ val ⇒ st ⇒ st"
"upd_gobj r n v ≡ st_case (λg l. st (chg_map (upd_obj n v) r g) l)"

set_locals    :: "locals ⇒ st ⇒ st"
"set_locals l ≡ st_case (λg l'. st g l)"

init_obj      :: "prog ⇒ obj_tag ⇒ oref ⇒ st ⇒ st"
"init_obj G oi r ≡ gupd(r↦(oi, init_vals (var_tys G oi r)))"

syntax
init_class_obj :: "prog ⇒ tname ⇒ st ⇒ st"

translations
"init_class_obj G C" == "init_obj G arbitrary (Inr C)"

datatype xcpt          (* exception *)
= XcptLoc loc          (* location of allocated exception object *)
| StdXcpt xname        (* intermediate standard exception, see Eval.thy *)
*)

consts

the_XcptLoc :: "xcpt ⇒ loc"

```

```

the_StdXcpt :: "xcpt ⇒ xname"

defs

the_XcptLoc_def "the_XcptLoc xc ≡ εa. xc = XcptLoc a"
the_StdXcpt_def "the_StdXcpt xc ≡ εx. xc = StdXcpt x"

types
xopt = "xcpt option"

constdefs
xcpt_if  :: "bool ⇒ xopt ⇒ xopt ⇒ xopt"
"xcpt_if c x' x ≡ if c ∧ (x = None) then x' else x"

syntax

raise_if :: "bool ⇒ xname ⇒ xopt ⇒ xopt"
np       :: "val           ⇒ xopt ⇒ xopt"
check_neg:: "val           ⇒ xopt ⇒ xopt"

translations

"raise_if c xn" == "xcpt_if c (Some (StdXcpt xn))"
"np v"          == "raise_if (v = Null)      NullPointer"
"check_neg i'" == "raise_if (the_Intg i'<0) NegArrSize"

types
state = "xopt × st"          (* state including exception information
*)

syntax

Norm    :: "st ⇒ state"

translations

"Norm s"      == "(None,s)"
"xopt"        <= (type) "State.xcpt option"
"xopt"        <= (type) "xcpt option"
"state"       <= (type) "xopt × State.st"
"state"       <= (type) "xopt × st"

constdefs

```

```

normal      :: "state  $\Rightarrow$  bool"
"normal  $\equiv \lambda s. \text{fst } s = \text{None}$ "

inited     :: "tname  $\Rightarrow$  globs  $\Rightarrow$  bool"
"inited C g  $\equiv g \text{ (Stat C)} \neq \text{None}$ "

initd      :: "tname  $\Rightarrow$  state  $\Rightarrow$  bool"
"initd C  $\equiv$  inited C  $\circ$  globs  $\circ$  snd"

heap_free  :: "nat  $\Rightarrow$  state  $\Rightarrow$  bool"
"heap_free n  $\equiv \lambda s. \text{atleast\_free (heap (snd s)) } n"$ 

xupd       :: "(xopt  $\Rightarrow$  xopt)  $\Rightarrow$  state  $\Rightarrow$  state"
"xupd f  $\equiv$  prod_fun f id"

supd       :: "(st  $\Rightarrow$  st)  $\Rightarrow$  state  $\Rightarrow$  state"
"supd  $\equiv$  prod_fun id"

syntax

set_lvars  :: "locals  $\Rightarrow$  state  $\Rightarrow$  state"
restore_lvars :: "state  $\Rightarrow$  state  $\Rightarrow$  state"

translations

"set_lvars l" == "supd (set_locals l)"
"restore_lvars s' s" == "set_lvars (locals (snd s')) s"

end

```

12 Eval

```

(* Title:      isabelle/Bali/Eval.thy
   ID:         $Id: Eval.thy,v 1.85 2000/11/27 15:20:15 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen
Operational evaluation (big-step) semantics of Java expressions and statements

```

improvements over Java Specification 1.0:

- * dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
- * throw raises a NullPointerException exception if a null reference is given, and

each

throw of a standard exception yield a fresh exception object (was not specified)

* if there is not enough memory even to allocate an `OutOfMemory` exception, evaluation/execution fails, i.e. simply stops (was not specified)

* array assignment checks lhs (and may throw exceptions) before evaluating rhs

* fixed exact positions of class initializations (immediate at first active use)

design issues:

* evaluation vs. (single-step) transition semantics

evaluation semantics chosen, because:

++ less verbose and therefore easier to read (and to handle in proofs)

+ more abstract

+ intermediate values (appearing in recursive rules) need not be stored explicitly, e.g. no call body construct or stack of invocation frames containing local variables and return addresses for method calls

needed

+ convenient rule induction for subject reduction theorem

- no interleaving (for parallelism) can be described

- stating a property of infinite executions requires the meta-level

argument

that this property holds for any finite prefixes of it (e.g. stopped

using

a counter that is decremented to zero and then throwing an exception)

* unified evaluation for variables, expressions, expression lists, statements

* the value entry in statement rules is redundant

* the value entry in rules is irrelevant in case of exceptions, but its full

inclusion helps to make the rule structure independent of exception occurrence.

* as irrelevant value entries are ignored, it does not matter if they are unique

For simplicity, (fixed) arbitrary values are preferred over "free" values.

* the rule format is such that the start state may contain an exception.

++ facilitates exception handling

+ symmetry

* the rules are defined carefully in order to be applicable even in not type-correct situations (yielding undefined values),

e.g. `the_Addr (Val (Bool b)) = arbitrary.`

++ fewer rules

- less readable because of auxiliary functions like `the_Addr`

Alternative: "defensive" evaluation throwing some `InternalError` exception

```

        in case of (impossible, for correct programs) type mismatches
* there is exactly one rule per syntactic construct
  + no redundancy in case distinctions
* halloc fails iff there is no free heap address. When there is
  only one free heap address left, it returns an OutOfMemory exception.
  In this way it is guaranteed that when an OutOfMemory exception is thrown
for
  the first time, there is a free location on the heap to allocate it.
* the allocation of objects that represent standard exceptions is deferred
until
  execution of any enclosing catch clause, which is transparent to the
program.
  - requires an auxiliary execution relation
  ++ avoids copies of allocation code and awkward case distinctions (whether

        there is enough memory to allocate the exception) in evaluation rules
* unfortunately new_Addr is not directly executable because of Hilbert
operator.

simplifications:
* local variables are initialized with default values (no definite assignment)
* garbage collection not considered, therefore also no finalizers
* stack overflow and memory overflow during class initialization not modelled
* exceptions in initializations not replaced by ExceptionInInitializerError
*)
Eval = State +

types vvar =      "val × (val ⇒ state ⇒ state)"
      vals =      "(val, vvar, val list) sum3"
translations
  "vvar" <= (type) "val × (val ⇒ state ⇒ state)"
  "vals" <= (type) "(val, vvar, val list) sum3"

constdefs
  arbitrary3 :: "('al + 'ar, 'b, 'c) sum3 ⇒ vals"
  "arbitrary3 ≡ sum3_case (In1 ∘ sum_case (λx. arbitrary) (λx. Unit))
    (λx. In2 arbitrary) (λx. In3 arbitrary)"

constdefs
  throw :: "val ⇒ xopt ⇒ xopt"
  "throw a' x ≡ xcpt_if True (Some (XcptLoc (the_Addr a')))) (np a' x)"

fits    :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool" ("_,_+_ fits _"[61,61,61,61]60)
"G,s⊢a' fits T ≡ (∃rt. T=RefT rt) → a'=Null ∨ G⊢obj_ty(lookup_obj

```

```

s a')  $\preceq$ T"

catch :: "prog  $\Rightarrow$  state  $\Rightarrow$  tname  $\Rightarrow$  bool"      ("_,_ $\vdash$ catch _"[61,61,61]60)
"G,s $\vdash$ catch C  $\equiv$   $\exists$  xc. fst s=Some xc  $\wedge$  G,snd s $\vdash$ Addr (the_XcptLoc xc) fits
Class C"

new_xcpt_var :: "ename  $\Rightarrow$  state  $\Rightarrow$  state"
"new_xcpt_var vn  $\equiv$   $\lambda$ (x,s). Norm(lupd(EName vn $\mapsto$ Addr (the_XcptLoc (the
x))) s)"

constdefs

assign      :: "('a  $\Rightarrow$  state  $\Rightarrow$  state)  $\Rightarrow$  'a  $\Rightarrow$  state  $\Rightarrow$  state"
"assign f v  $\equiv$   $\lambda$ (x,s). let (x',s') = (if x = None then f v else id) (x,s)
in (x',if x' = None then s' else s)"

init_comp_ty :: "ty  $\Rightarrow$  stmt"
"init_comp_ty T  $\equiv$  if ( $\exists$ C. T = Class C) then init (the_Class T) else
Skip"

constdefs

target      :: "inv_mode  $\Rightarrow$  st  $\Rightarrow$  val  $\Rightarrow$  ref_ty  $\Rightarrow$  tname"
"target m s a' t  $\equiv$  if m = IntVir
then obj_class (lookup_obj s a') else the_Class (RefT t)"

init_lvars  :: "prog  $\Rightarrow$  tname  $\Rightarrow$  sig  $\Rightarrow$  inv_mode  $\Rightarrow$  val  $\Rightarrow$  val list
 $\Rightarrow$ 
state  $\Rightarrow$  state"
"init_lvars G C sig mode a' pvs  $\equiv$   $\lambda$ (x,s). let
(_,(_,pns,_),lvars,_) = the (cmethd G C sig);
l = init_vals(table_of lvars)(pns[ $\mapsto$ ]pvs) (+)
(if mode=Static then empty else empty(( $\mapsto$ a')))
in set_lvars l (if mode = Static then x else np a' x,s)"

body :: "prog  $\Rightarrow$  tname  $\Rightarrow$  sig  $\Rightarrow$  expr"
"body G C sig  $\equiv$  let (D, _, _, c, e) = the (cmethd G C sig) in Body D
c e"

consts

eval      :: "prog  $\Rightarrow$  (state  $\times$  term  $\times$  vals  $\times$  state) set"
halloc:: "prog  $\Rightarrow$  (state  $\times$  obj_tag  $\times$  loc  $\times$  state) set"
sxalloc:: "prog  $\Rightarrow$  (state  $\times$  state) set"

```

constdefs

```
lvar :: "lname ⇒ st ⇒ vvar"
"lvar vn s ≡ (the (locals s vn), λv. supd (lupd(vn↦v)))"

fvar :: "tname ⇒ bool ⇒ ename ⇒ val ⇒ state ⇒ vvar × state"
"fvar C stat fn a' s ≡ let (oref,xf) = if stat then (Stat C,id)
                           else (Heap (the_Addr a'),np a');
    n = Inl (fn,C); f = (λv. supd (upd_gobj oref n
v)) in
    ((the (snd (the (globs (snd s) oref)) n),f),xupd xf
s)"

avar :: "prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state"
"avar G i' a' s ≡ let oref = Heap (the_Addr a'); i = the_Intg i'; n
= Inr i;
    (T,k,cs) = the_Arr (globs (snd s) oref); f = (λv
(x,s).
    (raise_if (¬G,s⊢v fits T) ArrStore x, upd_gobj oref n v
s)) in
    ((the (cs n),f), xupd (raise_if (¬i in_bounds k) IndOutBound ◦
np a') s)"

syntax
eval :: "[prog,state,term,vals*state]⇒bool"("_|-_ ->-> _" [61,61,80,
61]60)
exec :: "[prog,state,stmt ,state]⇒bool"("_|-_ ->-> _" [61,61,65,
61]60)
evar :: "[prog,state,var ,vvar,state]⇒bool"("_|-_ -=>-> _" [61,61,90,61,61]60)
eval_ :: "[prog,state,expr ,val ,state]⇒bool"("_|-_ ->-> _" [61,61,80,61,61]60)
evals :: "[prog,state,expr list ,
    val list ,state]⇒bool"("_|-_ -_#>-> _" [61,61,61,61,61]60)
hallo :: "[prog,state,obj_tag,
    loc,state]⇒bool"("_|-_ -halloc _>-> _" [61,61,61,61,61]60)
sallo :: "[prog,state ,state]⇒bool"("_|-_ -salloc-> _" [61,61,
61]60)

syntax (xsymbols)
dummy_res :: "vals" ("•")
eval :: "[prog,state,term,vals×state]⇒bool" ("_⊢_ -_>-> _" [61,61,80,
61]60)
exec :: "[prog,state,stmt ,state]⇒bool"("_⊢_ -_>-> _" [61,61,65,
61]60)
evar :: "[prog,state,var ,vvar,state]⇒bool"("_⊢_ -_=>-> _" [61,61,90,61,61]60)
```

```

eval_::"[prog,state,expr ,val ,state]⇒bool"("_+_ -_>_ → _"[61,61,80,61,61]60)
evals::"[prog,state,expr list ,
          val list ,state]⇒bool"("_+_ -_≐>_ → _"[61,61,61,61,61]60)
hallo::"[prog,state,obj_tag,
          loc,state]⇒bool"("_+_ -halloc _>_ → _"[61,61,61,61,61]60)
sallo::"[prog,state,
          state]⇒bool"("_+_ -sxalloc → _"[61,61,
61]60)

```

translations

```

"●" == "In1 Unit"
"G⊢s -t >→ w__s' " == "(s,t,w__s') ∈ eval G"
"G⊢s -t >→ (w, s')" <= "(s,t,w, s') ∈ eval G"
"G⊢s -t >→ (w,x,s')" <= "(s,t,w,x,s') ∈ eval G"
"G⊢s -c → (x,s')" <= "G⊢s -In1r c>→ (● ,x,s')"
"G⊢s -c → s' " == "G⊢s -In1r c>→ (● , s')"
"G⊢s -e>v → (x,s')" <= "G⊢s -In1l e>→ (In1 v ,x,s')"
"G⊢s -e>v → s' " == "G⊢s -In1l e>→ (In1 v , s')"
"G⊢s -e>vf → (x,s')" <= "G⊢s -In2 e>→ (In2 vf,x,s')"
"G⊢s -e>vf → s' " == "G⊢s -In2 e>→ (In2 vf, s')"
"G⊢s -e≐>v → (x,s')" <= "G⊢s -In3 e>→ (In3 v ,x,s')"
"G⊢s -e≐>v → s' " == "G⊢s -In3 e>→ (In3 v , s')"
"G⊢s -halloc oi>a → (x,s')" <= "(s,oi,a,x,s') ∈ halloc G"
"G⊢s -halloc oi>a → s' " == "(s,oi,a, s') ∈ halloc G"
"G⊢s -sxalloc → (x,s')" <= "(s ,x,s') ∈ sxalloc G"
"G⊢s -sxalloc → s' " == "(s , s') ∈ sxalloc G"

```

inductive "halloc G" intrs (* allocating objects on the heap, cf. 12.5 *)

Xcpt "G⊢(Some x,s) -halloc oi>arbitrary → (Some x,s)"

New "[[new_Addr (heap s) = Some a;
(x,oi') = (if atleast_free (heap s) 2 then (None,oi)
else (Some (XcptLoc a),CInst (SXcpt OutOfMemory)))]]

⇒

G⊢Norm s -halloc oi>a → (x,init_obj G oi' (Heap a) s)"

inductive "sxalloc G" intrs (* allocating exception objects for
standard exceptions (other than OutOfMemory)

*)

Norm "G⊢ Norm s -sxalloc → Norm s"

XcptL "G⊢(Some (XcptLoc a),s) -sxalloc → (Some (XcptLoc a),s)"

```

SXcpt "[G⊢Norm s0 -halloc (CInst (SXcpt xn))>a→ (x,s1)] ⇒
      G⊢(Some (StdXcpt xn),s0) -sxalloc→ (Some (XcptLoc a),s1)"

inductive "eval G" intrs

(* propagation of exceptions *)

(* cf. 14.1, 15.5 *)
Xcpt "G⊢(Some xc,s) -t>→ (arbitrary3 t,(Some xc,s))"

(* execution of statements *)

(* cf. 14.5 *)
Skip "G⊢Norm s -Skip→ Norm s"

(* cf. 14.7 *)
Expr "[G⊢Norm s0 -e->v→ s1] ⇒
      G⊢Norm s0 -Expr e→ s1"

(* cf. 14.2 *)
Comp "[G⊢Norm s0 -c1 → s1;
      G⊢      s1 -c2 → s2] ⇒
      G⊢Norm s0 -c1;; c2→ s2"

(* cf. 14.8.2 *)
If "[G⊢Norm s0 -e->b→ s1;
    G⊢      s1-(if the_Bool b then c1 else c2)→ s2] ⇒
    G⊢Norm s0 -If(e) c1 Else c2 → s2"

(* cf. 14.10, 14.10.1 *)
(* G⊢Norm s0 -If(e) (c;; While(e) c) Else Skip→ s3 *)
Loop "[G⊢Norm s0 -e->b→ s1;
      if the_Bool b then (G⊢s1 -c→ s2 ∧ G⊢s2 -While(e) c→ s3)
      else s3 = s1] ⇒
      G⊢Norm s0 -While(e) c→ s3"

(* cf. 14.16 *)
Throw "[G⊢Norm s0 -e->a'→ s1] ⇒
       G⊢Norm s0 -Throw e→ xupd (throw a')
s1"

```

```

(* cf. 14.18.1 *)
Try  "[[G⊢Norm s0 -c1→ s1; G⊢s1 -sxalloc→ s2;
      if G,s2⊢catch C then G⊢new_xcpt_var vn s2 -c2→ s3 else s3
= s2]] ⇒
      G⊢Norm s0 -Try c1 Catch(C vn) c2→ s3"

(* cf. 14.18.2 *)
Fin  "[[G⊢Norm s0 -c1→ (x1,s1);
      G⊢Norm s1 -c2→ s2]] ⇒
      G⊢Norm s0 -c1 Finally c2→ xupd (xcpt_if (x1≠None)
x1) s2"

(* cf. 12.4.2, 8.5 *)
Init "[[the (class G C) = (sc,si,fs,ms,ini);
      if inited C (globs s0) then s3 = Norm s0
      else (G⊢Norm (init_class_obj G C s0)
            -(if C = Object then Skip else init sc)→ s1 ∧
            G⊢set_lvars empty s1 -ini→ s2 ∧ s3 = restore_lvars
s1 s2)]] ⇒
      G⊢Norm s0 -init C→ s3"

(* evaluation of expressions *)

(* cf. 15.8.1, 12.4.1 *)
NewC "[[G⊢Norm s0 -init C→ s1;
      G⊢      s1 -halloc (CInst C)⋃a→ s2]] ⇒
      G⊢Norm s0 -NewC C-⋃Addr a→ s2"

(* cf. 15.9.1, 12.4.1 *)
NewA "[[G⊢Norm s0 -init_comp_ty T→ s1; G⊢s1 -e-⋃i'→ s2;
      G⊢xupd (check_neg i') s2 -halloc (Arr T (the_Intg i'))⋃a→
s3]] ⇒
      G⊢Norm s0 -New T[e]-⋃Addr a→ s3"

(* cf. 15.15 *)
Cast "[[G⊢Norm s0 -e-⋃v→ s1;
      s2 = xupd (raise_if (¬G,snd s1⊢v fits T) ClassCast) s1]] ⇒
      G⊢Norm s0 -Cast T e-⋃v→ s2"

(* cf. 15.19.2 *)
Inst "[[G⊢Norm s0 -e-⋃v→ s1;
      b = (v≠Null ∧ G,snd s1⊢v fits RefT T)]] ⇒

```

```

                                G⊢Norm s0 -e InstOf T->Bool b→ s1"

(* cf. 15.7.1 *)
Lit      "G⊢Norm s -Lit v->v→ Norm s"

(* cf. 15.10.2 *)
Super    "G⊢Norm s -Super->val_this s→ Norm
s"

(* cf. 15.2 *)
Acc      "[[G⊢Norm s0 -va=>(v,f)→ s1]] ⇒
                                G⊢Norm s0 -Acc va->v→ s1"

(* cf. 15.25.1 *)
Ass      "[[G⊢Norm s0 -va=>(w,f)→ s1;
          G⊢      s1 -e->v → s2]] ⇒
                                G⊢Norm s0 -va:=e->v→ assign f v
s2"

(* cf. 15.24 *)
Cond     "[[G⊢Norm s0 -e0->b→ s1;
          G⊢      s1 -(if the_Bool b then e1 else e2)->v→ s2]] ⇒
                                G⊢Norm s0 -e0 ? e1 : e2->v→ s2"

(* cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5 *)
Call     "[[G⊢Norm s0 -e->a'→ s1; G⊢s1 -args≐>vs→ s2;
          C = target mode (snd s2) a' cT;
          G⊢init_lvars G C (mn,pTs) mode a' vs s2 -Methd C (mn,pTs)->v→
s3]] ⇒
          G⊢Norm s0 -{t,cT,mode}e..mn({pTs}args)->v→ (restore_lvars
s2 s3)"

Methd    "[[G⊢Norm s0 -body G C sig->v→ s1]] ⇒
                                G⊢Norm s0 -Methd C sig->v→ s1"

(* cf. 14.15, 12.4.1 *)
Body     "[[G⊢Norm s0 -init D→ s1; G⊢s1 -c→ s2; G⊢s2 -e->v→ s3]] ⇒
                                G⊢Norm s0 -Body D c e->v→ s3"

(* evaluation of variables *)

```

```

(* cf. 15.13.1, 15.7.2 *)
LVar  "G⊢Norm s -LVar vn=>lvar vn s→ Norm s"

(* cf. 15.10.1, 12.4.1 *)
FVar  "[[G⊢Norm s0 -init C→ s1; G⊢s1 -e-λa→ s2;
        (v,s2') = fvar C stat fn a s2]] ⇒
        G⊢Norm s0 -{C,stat}e..fn=>v→ s2'"

(* cf. 15.12.1, 15.25.1 *)
AVar  "[[G⊢ Norm s0 -e1-λa→ s1; G⊢s1 -e2-λi→ s2;
        (v,s2') = avar G i a s2]] ⇒
        G⊢Norm s0 -e1.[e2]=λv→ s2'"

(* evaluation of expression lists *)

(* cf. 15.11.4.2 *)
Nil
                                           "G⊢Norm s0 -[] ≐λ [] → Norm s0"

(* cf. 15.6.4 *)
Cons  "[[G⊢Norm s0 -e -λ v → s1;
        G⊢      s1 -es ≐λ vs → s2]] ⇒
        G⊢Norm s0 -e#es ≐λ v#vs → s2"

monos
  if_def2

end

```

13 Evaln

```

(* Title:      isabelle/Bali/Evaln.thy
   ID:         $Id: Evaln.thy,v 1.32 2000/11/19 19:09:36 oheimb Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

```

Operational evaluation (big-step) semantics of Java expressions and statements
 Variant of eval relation with counter for bounded recursive depth
 Evaln could completely replace Eval.

*)

Evaln = Eval +

consts

evaln :: "prog \Rightarrow (state \times term \times nat \times vals \times state) set"

syntax

evaln :: "[prog, state, term, nat, vals * state] \Rightarrow bool"
(" |- \rightarrow " [61,61,80, 61,61]

60)

evarn :: "[prog, state, var , vvar , nat, state] \Rightarrow bool"
(" |- \rightarrow " [61,61,90,61,61,61]

60)

eval_n:: "[prog, state, expr , val , nat, state] \Rightarrow bool"
(" |- \rightarrow " [61,61,80,61,61,61]

60)

evalsn:: "[prog, state, expr list, val list, nat, state] \Rightarrow bool"
(" |- \rightarrow " [61,61,61,61,61,61]

60)

execn :: "[prog, state, stmt , nat, state] \Rightarrow bool"
(" |- \rightarrow " [61,61,65, 61,61]

60)

syntax (xsymbols)

evaln :: "[prog, state, term, nat, vals \times state] \Rightarrow bool"
(" \vdash \rightarrow " [61,61,80, 61,61]

60)

evarn :: "[prog, state, var , vvar , nat, state] \Rightarrow bool"
(" \vdash \rightarrow " [61,61,90,61,61,61]

60)

eval_n:: "[prog, state, expr , val , nat, state] \Rightarrow bool"
(" \vdash \rightarrow " [61,61,80,61,61,61]

60)

evalsn:: "[prog, state, expr list, val list, nat, state] \Rightarrow bool"
(" \vdash \rightarrow " [61,61,61,61,61,61]

60)

execn :: "[prog, state, stmt , nat, state] \Rightarrow bool"
(" \vdash \rightarrow " [61,61,65, 61,61]

60)

translations

" $G \vdash s \rightarrow t \quad \rightarrow n \rightarrow w_s'$ " == "(s,t,n,w_s') \in evaln G"

```

"G⊢s -t    γ-n → (w, s')" <= "(s,t,n,w, s') ∈ evaln G"
"G⊢s -t    γ-n → (w,x,s')" <= "(s,t,n,w,x,s') ∈ evaln G"
"G⊢s -c    -n → (x,s')" <= "G⊢s -In1r cγ-n → (●, x,s')"
"G⊢s -c    -n → s' " == "G⊢s -In1r cγ-n → (●, s')"
"G⊢s -e-γv -n → (x,s')" <= "G⊢s -In1l eγ-n → (In1 v ,x,s')"
"G⊢s -e-γv -n → s' " == "G⊢s -In1l eγ-n → (In1 v , s')"
"G⊢s -e=γvf -n → (x,s')" <= "G⊢s -In2 eγ-n → (In2 vf,x,s')"
"G⊢s -e=γvf -n → s' " == "G⊢s -In2 eγ-n → (In2 vf, s')"
"G⊢s -e≐γv -n → (x,s')" <= "G⊢s -In3 eγ-n → (In3 v ,x,s')"
"G⊢s -e≐γv -n → s' " == "G⊢s -In3 eγ-n → (In3 v , s')"

```

inductive "evaln G" intrs

(* propagation of exceptions *)

```
Xcpt "G⊢(Some xc,s) -tγ-n → (arbitrary3 t,(Some xc,s))"
```

(* evaluation of variables *)

```
LVar "G⊢Norm s -LVar vn=>lvar vn s-n → Norm s"
```

```
FVar "[G⊢Norm s0 -init C-n → s1; G⊢s1 -e-γa'-n → s2;
(v,s2') = fvar C stat fn a' s2] ==>
G⊢Norm s0 -{C,stat}e..fn=>v-n → s2'"
```

```
AVar "[G⊢ Norm s0 -e1-γa-n → s1 ; G⊢s1 -e2-γi-n → s2;
(v,s2') = avar G i a s2] ==>
G⊢Norm s0 -e1.[e2]=>v-n → s2'"
```

(* evaluation of expressions *)

```
NewC "[G⊢Norm s0 -init C-n → s1;
G⊢ s1 -halloc (CInst C)γa → s2] ==>
G⊢Norm s0 -NewC C-γAddr a-n → s2"
```

```
NewA "[G⊢Norm s0 -init_comp_ty T-n → s1; G⊢s1 -e-γi'-n → s2;
G⊢xupd (check_neg i') s2 -halloc (Arr T (the_Intg i'))γa →
s3] ==>
G⊢Norm s0 -New T[e]-γAddr a-n → s3"
```

$$\text{Cast } \llbracket G \vdash \text{Norm } s0 \text{ -}e\text{-}\lambda v\text{-}n \rightarrow s1; \\ s2 = \text{xupd } (\text{raise_if } (\neg G, \text{snd } s1 \vdash v \text{ fits } T) \text{ ClassCast}) s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Cast } T \text{ } e\text{-}\lambda v\text{-}n \rightarrow s2$$

$$\text{Inst } \llbracket G \vdash \text{Norm } s0 \text{ -}e\text{-}\lambda v\text{-}n \rightarrow s1; \\ b = (v \neq \text{Null} \wedge G, \text{snd } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -}e \text{ InstOf } T\text{-}\lambda \text{Bool } b\text{-}n \rightarrow s1$$

$$\text{Lit } \quad \quad \quad "G \vdash \text{Norm } s \text{ -Lit } v\text{-}\lambda v\text{-}n \rightarrow \text{Norm } s"$$

$$\text{Super } \quad \quad \quad "G \vdash \text{Norm } s \text{ -Super-}\lambda \text{val_this } s\text{-}n \rightarrow \text{Norm } s"$$

$$\text{Acc } \quad \llbracket G \vdash \text{Norm } s0 \text{ -}va = \lambda (v, f)\text{-}n \rightarrow s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Acc } va\text{-}\lambda v\text{-}n \rightarrow s1$$

$$\text{Ass } \quad \llbracket G \vdash \text{Norm } s0 \text{ -}va = \lambda (w, f)\text{-}n \rightarrow s1; \\ G \vdash \quad s1 \text{ -}e\text{-}\lambda v \quad \text{-}n \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -}va := e\text{-}\lambda v\text{-}n \rightarrow \text{assign } f \\ v \text{ } s2$$

$$\text{Cond } \quad \llbracket G \vdash \text{Norm } s0 \text{ -}e0\text{-}\lambda b\text{-}n \rightarrow s1; \\ G \vdash \quad s1 \text{ -(if the_Bool } b \text{ then } e1 \text{ else } e2)\text{-}\lambda v\text{-}n \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -}e0 \text{ ? } e1 : e2\text{-}\lambda v\text{-}n \rightarrow s2$$

$$\text{Call } \quad \llbracket G \vdash \text{Norm } s0 \text{ -}e\text{-}\lambda a'\text{-}n \rightarrow s1; G \vdash s1 \text{ -args } \dot{=} \lambda vs\text{-}n \rightarrow s2; \\ C = \text{target mode (snd } s2) a' \text{ } cT; \\ G \vdash \text{init_lvars } G \text{ } C (mn, pTs) \text{ mode } a' \text{ vs } s2 \text{ -Methd } C (mn, pTs)\text{-}\lambda v\text{-}n \rightarrow \\ s3 \rrbracket \\ \implies G \vdash \text{Norm } s0 \text{ -}\{t, cT, \text{mode}\}e..mn(\{pTs\}args)\text{-}\lambda v\text{-}n \rightarrow (\text{restore_lvars} \\ s2 \text{ } s3)$$

$$\text{Methd } \llbracket G \vdash \text{Norm } s0 \text{ -body } G \text{ } C \text{ sig}\text{-}\lambda v\text{-}n \rightarrow s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Methd } C \text{ sig}\text{-}\lambda v\text{-}n \text{ -Suc } n \rightarrow s1$$

$$\text{Body } \quad \llbracket G \vdash \text{Norm } s0 \text{ -init } D\text{-}n \rightarrow s1; G \vdash s1 \text{ -}c\text{-}n \rightarrow s2; G \vdash s2 \text{ -}e\text{-}\lambda v\text{-}n \rightarrow \\ s3 \rrbracket \implies \\ G \vdash \text{Norm } s0 \text{ -Body } D \text{ } c \text{ } e\text{-}\lambda v\text{-}n \rightarrow s3$$

(* evaluation of expression lists *)

$$\text{Nil} \quad \quad \quad "G \vdash \text{Norm } s0 \text{ -}[] \dot{=} \lambda []\text{-}n \rightarrow \text{Norm } s0"$$

```

Cons  "[[G⊢Norm s0 -e -> v -n→ s1;
      G⊢    s1 -es≐>vs-n→ s2]] ⇒
      G⊢Norm s0 -e#es≐>v#vs-n→ s2"

(* execution of statements *)

Skip  "G⊢Norm s -Skip-n→ Norm s"

Expr  "[[G⊢Norm s0 -e->v-n→ s1]] ⇒
      G⊢Norm s0 -Expr e-n→ s1"

Comp  "[[G⊢Norm s0 -c1 -n→ s1;
      G⊢    s1 -c2 -n→ s2]] ⇒
      G⊢Norm s0 -c1;; c2-n→ s2"

If     "[[G⊢Norm s0 -e->b-n→ s1;
      G⊢    s1-(if the_Bool b then c1 else c2)-n→ s2]] ⇒
      G⊢Norm s0 -If(e) c1 Else c2 -n→ s2"

Loop  "[[G⊢Norm s0 -e->b-n→ s1;
      if the_Bool b then (G⊢s1 -c-n→ s2 ∧ G⊢s2 -While(e) c-n→
s3)
      else s3 = s1]] ⇒
      G⊢Norm s0 -While(e) c-n→ s3"

Throw "[[G⊢Norm s0 -e->a'-n→ s1]] ⇒
      G⊢Norm s0 -Throw e-n→ xupd (throw
a') s1"

Try   "[[G⊢Norm s0 -c1-n→ s1; G⊢s1 -sxalloc→ s2;
      if G,s2⊢catch tn then G⊢new_xcpt_var vn s2 -c2-n→ s3 else
s3 = s2]] ⇒
      G⊢Norm s0 -Try c1 Catch(tn vn) c2-n→ s3"

Fin   "[[G⊢Norm s0 -c1-n→ (x1,s1);
      G⊢Norm s1 -c2-n→ s2]] ⇒
      G⊢Norm s0 -c1 Finally c2-n→ xupd (xcpt_if (x1≠None)
x1) s2"

Init  "[[the (class G C) = (sc,si,fs,ms,ini);
      if inited C (globs s0) then s3 = Norm s0
      else (G⊢Norm (init_class_obj G C s0)
      -(if C = Object then Skip else init sc)-n→ s1 ∧

```

```

    G⊢set_lvars empty s1 -ini-n→ s2 ∧ s3 = restore_lvars
s1 s2)]]⇒
    G⊢Norm s0 -init C-n→ s3"
monos
  if_def2
end

```

14 Example

```

(* Title:      isabelle/Bali/Example.thy
   ID:         $Id: Example.thy,v 1.39 2000/11/29 22:48:45 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

The following example Bali program includes:

```

* class and interface declarations with inheritance, hiding of fields,
  overriding of methods (with refined result type), array type,
* method call (with dynamic binding), parameter access, return expressions,
* expression statements, sequential composition, literal values,
  local assignment, local access, field assignment, type cast,
* exception generation and propagation, try & catch statement, throw statement
* instance creation and (default) static initialization

```

```

interface HasFoo {
  public Base foo(Base z);
}

```

```

class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

```

```

class Ext extends Base {
  int vee;
  public Ext foo(Base z) {
    ((Ext)z).vee = 1;
    return null;
  }
}

```

```

}
}

class Example {
  public static void main(String args[]) throws Throwable {
    Base e = new Ext();
    try {e.foo(null); }
    catch(NullPointerException z) {
      while(Ext.arr[2]) ;
    }
  }
}
*)

Example = Eval + WellForm +           (** cannot simply instantiate
tnam **)

datatype tnam_ = HasFoo_ | Base_ | Ext_
datatype enam_ = arr_ | vee_ | z_ | e_

consts

  tnam_ :: "tnam_  $\Rightarrow$  tnam"
  enam_ :: "enam_  $\Rightarrow$  ename"

rules (** tnam_ and enam_ are intended to be isomorphic to tnam and ename
**)

  inj_tnam_ "(tnam_ x = tnam_ y) = (x = y)"
  inj_enam_ "(enam_ x = enam_ y) = (x = y)"

  surj_tnam_ " $\exists m. n = tnam_ m$ "
  surj_enam_ " $\exists m. n = enam_ m$ "

defs

  Object_mdecls_def "Object_mdecls  $\equiv$  []"
  SXcpt_mdecls_def "SXcpt_mdecls  $\equiv$  []"

syntax

  HasFoo, Base, Ext :: tname
  arr, vee, z, e    :: ename

```

translations

```
"HasFoo" == "TName (tnam_ HasFoo_)"
"Base"   == "TName (tnam_ Base_)"
"Ext"    == "TName (tnam_ Ext_)"
"arr"    ==      "enam_ arr_"
"vee"    ==      "enam_ vee_"
"z"      ==      "enam_ z_"
"e"      ==      "enam_ e_"
```

consts

```
foo    :: mname
```

constdefs

```
foo_sig  :: sig
"foo_sig ≡ (foo, [Class Base])"

foo_mhead :: mhead
"foo_mhead ≡ (False, [z], Class Base)"
```

constdefs

```
Base_foo :: mdecl
"Base_foo ≡ (foo_sig, (foo_mhead, ([], Skip, !!z)))"

Ext_foo  :: mdecl
"Ext_foo ≡ (foo_sig, ((False, [z], Class Ext),
  ([], Expr({Ext, False}Cast (Class Ext) (!!z)..vee :=
    Lit (Intg #1)), Lit Null)
  ))"

arr_viewed_from :: "tname ⇒ var"
"arr_viewed_from C ≡ {Base, True}StatRef (ClassT C)..arr"
```

constdefs

```
HasFooInt :: iface
"HasFooInt ≡ ([], [(foo_sig, foo_mhead)])"

BaseCl :: class
"BaseCl ≡ (Object, [HasFoo],
  [(arr, (True, PrimT Boolean. []))],
```

```

        (vee, (False, Iface HasFoo    ))],
    [Base_foo],
    Expr(arr_viewed_from Base := New (PrimT Boolean)[Lit (Intg
#2)]))"

    ExtCl  :: class
    "ExtCl ≡ (Base , [],
        [(vee, (False, PrimT Integer  ))],
        [Ext_foo],
        Skip)"

constdefs

    ifaces :: idecl list
    "ifaces ≡ [(HasFoo,HasFooInt)]"

    classes :: cdecl list (** name not 'classes' because of clash with thy
token**)
    "classes ≡ [(Base,BaseCl),(Ext,ExtCl)]@standard_classes"

    test    :: "(ty)list ⇒ stmt"
    "test pTs ≡ e:=NewC Ext;;
        Try Expr({ClassT Base,ClassT Base,IntVir}!!e..
            foo({pTs}[Lit Null]))
        Catch((SXcpt NullPointer) z)
        (While(Acc (Acc (arr_viewed_from Ext).[Lit (Intg #2)]))
Skip)"

consts
    a,b,c    :: loc

syntax

"classes"      :: cdecl list
tprg           :: prog

obj_a, obj_b, obj_c :: obj
arr_N, arr_a    :: (vn, val) table
globs1,globs2,
globs3,globs8  :: globs
locs3,locs4,locs8  :: locals
s0,s0',s9',
s1,s1',s2,s2',
s3,s3',s4,s4',

```

```

s6',s7',s8,s8'      :: state

translations

"classes" == "clsses"
"tprg"    == "(ifaces,classes)"

"obj_a"   <= "(Arr (PrimT Boolean) #2, empty(Inr #0→Bool False)(Inr
#1→Bool False))"
"obj_b"   <= "(CInst Ext,(empty(Inl (vee, Base)→Null    )
                (Inl (vee, Ext )→Intg #0)))"
"obj_c"   == "(CInst (SXcpt NullPointer),empty)"
"arr_N"   == "empty(Inl (arr, Base)→Null)"
"arr_a"   == "empty(Inl (arr, Base)→Addr a)"
"globals1" == "empty(Inr Ext   ↳(arbitrary, empty))
                (Inr Base   ↳(arbitrary, arr_N))
                (Inr Object↳(arbitrary, empty))"
"globals2" == "empty(Inr Ext   ↳(arbitrary, empty))
                (Inr Object↳(arbitrary, empty))
                (Inl a→obj_a)
                (Inr Base   ↳(arbitrary, arr_a))"
"globals3" == "globals2(Inl b→obj_b)"
"globals8" == "globals3(Inl c→obj_c)"
"locs3"    == "empty(Inl e→Addr b)"
"locs4"    == "empty(Inl z→Null)(Inr()↳Addr b)"
"locs8"    == "locs3(Inl z→Addr c)"
"s0"      == "      st empty empty"
"s0'"     == " Norm s0"
"s1"      == "      st globals1 empty"
"s1'"     == " Norm s1"
"s2"      == "      st globals2 empty"
"s2'"     == " Norm s2"
"s3"      == "      st globals3 locs3 "
"s3'"     == " Norm s3"
"s4"      == "      st globals3 locs4"
"s4'"     == " Norm s4"
"s6'"     == "(Some (StdXcpt NullPointer), s4)"
"s7'"     == "(Some (StdXcpt NullPointer), s3)"
"s8"      == "      st globals8 locs8"
"s8'"     == " Norm s8"
"s9'"     == "(Some (StdXcpt IndOutBound), s8)"

end

```

15 Conform

```
(* Title:      isabelle/Bali/Conform.thy
   ID:         $Id: Conform.thy,v 1.5 2000/10/19 21:21:51 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen
```

Conformance notions for the type soundness proof for Java

design issues:

```
* lconf allows for (arbitrary) inaccessible values
* "conforms" does not directly imply that the dynamic types of all objects
  on
  the heap are indeed existing classes. Yet this can be inferred for all
  referenced objs.
*)
```

Conform = State +

```
types  env_ = "prog × (lname, ty) table" (* same as env of WellType.thy
*)
```

constdefs

```
gext   :: "st ⇒ st ⇒ bool"                ("_≤|_" [71,71] 70)
"s≤|s' ≡ ∀r. ∀(oi, fs)∈globs s r: ∃(oi', fs')∈globs s' r: oi' =
oi"
```

```
conf   :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool"  ("_,_⊢_::≤_" [71,71,71,71]
70)
```

```
"G,s⊢v::≤T ≡ ∃T'∈typeof (λa. option_map obj_ty (heap s a))
v:G⊢T'≤T"
```

```
lconf  :: "prog ⇒ st ⇒ ('a, val) table ⇒ ('a, ty) table ⇒ bool"
("_,_⊢_[:≤]" [71,71,71,71]
70)
```

```
"G,s⊢vs[:≤]Ts ≡ ∀n. ∀T∈Ts n: ∃v∈vs n: G,s⊢v::≤T"
```

```
oconf  :: "prog ⇒ st ⇒ obj ⇒ oref ⇒ bool" ("_,_⊢_::≤√_" [71,71,71,71]
70)
```

```
"G,s⊢obj::≤√r ≡ G,s⊢snd obj[:≤]var_tys G (fst obj) r ∧ (case
r of
```

```
  Heap a ⇒ is_type G (obj_ty obj) | Stat
```

```

C ⇒ True)"

conforms :: "state ⇒ env_ ⇒ bool"          ( "_::≲_" [71,71]
70)
    "xs::≲E ≡ let (G, L) = E; s = snd xs; l = locals s in
(∀r. ∀obj∈globs s r:          G,s⊢obj  ::≲√r) ∧
                                G,s⊢l    [::≲]L  ∧
(∀a. fst xs=Some(XcptLoc a) → G,s⊢Addr a::≲Class (SXcpt Throwable))"

end

```

16 TypeSafe

```

(* Title:      isabelle/Bali/TypeSafe.thy
   ID:         $Id: TypeSafe.thy,v 1.22 2000/11/28 23:06:20 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen

```

The type soundness proof for Java
*)

TypeSafe = Eval + WellForm + Conform +

```

constdefs
  DynT_prop::"[prog,inv_mode,tname,ref_ty] ⇒ bool" ("_⊢_→_≲_"[71,71,71,71]70)
  "G⊢mode→D≲t ≡ mode = IntVir → is_class G D ∧
    (if (∃T. t=ArrayT T) then D=Object else G⊢Class
D≲RefT t)"

  assign_conforms :: "st ⇒ (val ⇒ state ⇒ state) ⇒ ty ⇒ env ⇒ bool"
    ("_≤|_≲_::≲_" [71,71,71,71]
70)
  "s≤|f≲T::≲E ≡
  ∀s' w. Norm s'::≲E → fst E,s'⊢w::≲T → s≤|s' → assign f w (Norm
s')::≲E"

  rconf :: "prog ⇒ lenv ⇒ st ⇒ term ⇒ vals ⇒ tys ⇒ bool"
    ("_,_,_⊢_>_::≲_" [71,71,71,71,71,71]
70)
  "G,L,s⊢t>v::≲T ≡ case T of
Inl T ⇒ if (∃vf. t=In2 vf)

```

```

        then G,s⊢fst (the_In2 v)::⊆T ∧ s≤|snd (the_In2 v) ⊆T::⊆(G,L)
        else G,s⊢the_In1 v::⊆T
    | Inr Ts ⇒ list_all2 (conf G s) (the_In3 v) Ts"

end

```

17 AxSem

```

(* Title:      isabelle/Bali/AxSem.thy
   ID:         $Id: AxSem.thy,v 1.109 2000/11/25 23:58:27 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright   1998 Technische Universitaet Muenchen

```

Axiomatic semantics of Java expressions and statements (see also Eval.thy)

design issues:

- * a strong version of validity for triples with premises, namely one that takes

- the recursive depth needed to complete execution, enables correctness proof

- * auxiliary variables are handled first-class (-> Thomas Kleymann)

- * expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class => explicit result value handling

- * intermediate values not on triple, but on assertion level (with result entry)

- * multiple results with semantical substitution mechanism not requiring a stack

- * because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements

- * result values in triples exactly as in eval relation (also for xcpt states)

- * validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- * all triples in a derivation are of the same type (due to weak polymorphism)
- *)

AxSem = Evaln + TypeSafe +

```

types res = vals (* result entry *)
syntax
  Val  :: "val      ⇒ res"
  Var  :: "var      ⇒ res"
  Vals :: "val list ⇒ res"
translations
  "Val x"    => "(In1 x)"
  "Var x"    => "(In2 x)"
  "Vals x"   => "(In3 x)"

syntax
  "Val_"    :: [pttrn] => pttrn    ("Val:_" [951] 950)
  "Var_"    :: [pttrn] => pttrn    ("Var:_" [951] 950)
  "Vals_"   :: [pttrn] => pttrn    ("Vals:_" [951] 950)
translations
  "λVal:v . b" == "(λv. b) o the_In1"
  "λVar:v . b" == "(λv. b) o the_In2"
  "λVals:v. b" == "(λv. b) o the_In3"

(* relation on result values, state and auxiliary variables *)
types 'a assn = "res ⇒ state ⇒ 'a ⇒ bool"
translations
  "res"    <= (type) "AxSem.res"
  "a assn" <= (type) "vals ⇒ state ⇒ a ⇒ bool"

constdefs
  assn_imp  :: "'a assn ⇒ 'a assn ⇒ bool"          (infixr "⇒"
25)
  "P ⇒ Q ≡ ∀Y s Z. P Y s Z → Q Y s Z"

  peek_and :: "'a assn ⇒ (state ⇒ bool) ⇒ 'a assn" (infixl "∧."
13)
  "P ∧. p ≡ λY s Z. P Y s Z ∧ p s"

  assn_supd :: "'a assn ⇒ (state ⇒ state) ⇒ 'a assn" (infixl ";."
13)
  "P ;. f ≡ λY s' Z. ∃s. P Y s Z ∧ s' = f s"

  supd_assn :: "(state ⇒ state) ⇒ 'a assn ⇒ 'a assn" (infixr ".;"
13)
  "f .; P ≡ λY s. P Y (f s)"

  subst_res :: "'a assn ⇒ res ⇒ 'a assn"          ("_←_" [60,61]

```

```

60)
  "P←w ≡ λY. P w"

  subst_Bool  :: "'a assn ⇒ bool ⇒ 'a assn"          ("_←=" [60,61]
60)
  "P←=b ≡ λY s Z. ∃v. P (Val v) s Z ∧ (normal s → the_Bool v=b)"

  peek_res    :: "(res ⇒ 'a assn) ⇒ 'a assn"
  "peek_res Pf ≡ λY. Pf Y Y"

  peek_st     :: "(st ⇒ 'a assn) ⇒ 'a assn"
  "peek_st P ≡ λY s. P (snd s) Y s"

  ign_res     :: "          'a assn ⇒ 'a assn"          ("_↓" [1000]
1000)
  "P↓        ≡ λY s Z. ∃Y. P Y s Z"

syntax
  Normal      :: "          'a assn ⇒ 'a assn"
  "@peek_res" :: "pttrn ⇒ 'a assn ⇒ 'a assn"          ("λ.:. _" [0,3]
3)
  "@peek_st"  :: "pttrn ⇒ 'a assn ⇒ 'a assn"          ("λ.. _" [0,3]
3)

translations
  "Normal P" == "P ∧. normal"
  "λw:. P"   == "peek_res (λw. P)"
  "λs.. P"   == "peek_st (λs. P)"

constdefs
  ign_res_eq :: "'a assn ⇒ res ⇒ 'a assn"          ("_↓=" [60,61]
60)
  "P↓=w      ≡ λY:. P↓ ∧. (λs. Y=w)"

  RefVar     :: "(state ⇒ vvar × state) ⇒ 'a assn ⇒ 'a assn"(infixr
"..;" 13)
  "vf ..; P ≡ λY s. let (v,s') = vf s in P (Var v) s'"

  Alloc      :: "prog ⇒ obj_tag ⇒ 'a assn ⇒ 'a assn"
  "Alloc G otag P ≡ λY s Z.
      ∀s' a. G⊢s -halloc otag>a→ s' → P (Val (Addr a))
s' Z"

  SXAlloc    :: "prog ⇒ 'a assn ⇒ 'a assn"

```

```

" SXAlloc G P ≡ λY s Z. ∀ s'. G ⊢ s -sxalloc→ s' → P Y s' Z"

type_ok :: "prog ⇒ term ⇒ state ⇒ bool"
"type_ok G t s ≡ ∃ L T. (normal s → (G,L) ⊢ t :: T) ∧ s :: ⋮(G,L)"

datatype 'a triple = triple ('a assn) term ('a assn) (** should
be
something like triple = ∀ 'a. triple ('a assn) term ('a assn) **)

types 'a triples = 'a triple set

syntax

var_triple  :: "[ 'a assn, var      , 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ => / {{(1_)}" [3,80,3]
75)
expr_triple  :: "[ 'a assn, expr    , 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ -> / {{(1_)}" [3,80,3]
75)
exprs_triple :: "[ 'a assn, expr list, 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ #> / {{(1_)}" [3,65,3]
75)
stmt_triple  :: "[ 'a assn, stmt,    'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ . / {{(1_)}" [3,65,3]
75)

syntax (xsymbols)

triple      :: "[ 'a assn, term      , 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ > / {{(1_)}" [3,65,3]
75)
var_triple  :: "[ 'a assn, var      , 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ => / {{(1_)}" [3,80,3]
75)
expr_triple  :: "[ 'a assn, expr    , 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ -> / {{(1_)}" [3,80,3]
75)
exprs_triple :: "[ 'a assn, expr list, 'a assn ] ⇒ 'a triple"
              ("{{(1_)} / _ ≐ > / {{(1_)}" [3,65,3]
75)

translations
"{P} e -> {Q}" == "{P} In11 e > {Q}"

```

```

"{P} e=> {Q}" == "{P} In2 e> {Q}"
"{P} e≐> {Q}" == "{P} In3 e> {Q}"
"{P} .c. {Q}" == "{P} In1r c> {Q}"

constdefs
  mtriples :: "('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c ⇒ 'sig ⇒ expr) ⇒
            ('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c × 'sig) set ⇒ 'a triples"
            ("{{(1_)} / _-> / {(1_)} | _}" [3,65,3,65] 75)
  "{P} tf-> {Q} | ms} ≡ (λ(C,sig). {Normal(P C sig)} tf C sig-> {Q C
sig}) 'ms"

consts

  triple_valid :: "prog ⇒ nat ⇒          'a triple ⇒ bool"
                ( "_|=_" [61,0, 58]
57)
  ax_valids :: "prog ⇒ 'b triples ⇒ 'a triples ⇒ bool"
              ( "_,_|=_" [61,58,58]
57)
  ax_derivs :: "prog ⇒ ('b triples × 'a triples) set"

syntax

  triples_valid:: "prog ⇒ nat ⇒          'a triples ⇒ bool"
                 ( "_||=_" [61,0, 58]
57)
  ax_valid :: "prog ⇒ 'b triples ⇒ 'a triple ⇒ bool"
            ( "_,_|=_" [61,58,58]
57)
  ax_Derivs:: "prog ⇒ 'b triples ⇒ 'a triples ⇒ bool"
             ( "_,_||-" [61,58,58]
57)
  ax_Deriv :: "prog ⇒ 'b triples ⇒ 'a triple ⇒ bool"
            ( "_,_|-" [61,58,58]
57)

syntax (xsymbols)

  triples_valid:: "prog ⇒ nat ⇒          'a triples ⇒ bool"
                 ( "_||=_" [61,0, 58]
57)
  ax_valid :: "prog ⇒ 'b triples ⇒ 'a triple ⇒ bool"
            ( "_,_|=_" [61,58,58]
57)

```

```

ax_Derivs:: "prog ⇒ 'b triples ⇒ 'a triples ⇒ bool"
          ("_,_|t_" [61,58,58]
57)
ax_Deriv :: "prog ⇒ 'b triples ⇒ 'a triple ⇒ bool"
          ( "_,-|t_" [61,58,58]
57)

defs triple_valid_def "G|=n:t ≡ case t of {P} t> {Q} ⇒
  ∀Y s Z. P Y s Z → type_ok G t s →
  (∀Y' s'. G|s -t>-n → (Y',s') → Q Y' s'
Z)"
translations "G|=n:ts" == "Ball ts (triple_valid G n)"
defs ax_valids_def "G,A|=ts ≡ ∀n. G|=n:A → G|=n:ts"
translations "G,A|=t" == "G,A|= {t}"
          "G,A|ts" == "(A,ts) ∈ ax_derivs G"
          "G,A|t" == "G,A| {t}"

inductive "ax_derivs G" intrs

empty " G,A|{}"
insert "[G,A|t; G,A|ts] ⇒
  G,A|insert t ts"

asm "ts ⊆ A ⇒ G,A|ts"

(* could be added for convenience and efficiency, but is not necessary
cut "[G,A'|ts; G,A|A'] ⇒
  G,A|ts"
*)

weaken "[G,A|ts'; ts ⊆ ts'] ⇒ G,A|ts"

conseq "∀Y s Z . P Y s Z → (∃P' Q'. G,A|{P'} t> {Q'} ∧ (∀Y' s' .
  (∀Y Z'. P' Y s Z' → Q' Y' s' Z') →
  Q Y' s' Z ))
⇒ G,A|{P } t> {Q }"

hazard "G,A|{P ∧. Not ∘ type_ok G t} t> {Q}"

Xcpt "G,A|{P←(arbitrary3 t) ∧. Not ∘ normal} t> {P}"

(* variables *)
LVar " G,A|{Normal (λs.. P←Var (lvar vn s))} LVar vn=> {P}"

```

```

FVar "[G,A⊢{Normal P} .init C. {Q}];
      G,A⊢{Q} e-⋄ {λVal:a:. fvar C stat fn a ..; R}] ⇒
      G,A⊢{Normal P} {C,stat}e..fn=⋄ {R}"

AVar "[G,A⊢{Normal P} e1-⋄ {Q}];
      ∀a. G,A⊢{Q←Val a} e2-⋄ {λVal:i:. avar G i a ..; R}] ⇒
      G,A⊢{Normal P} e1.[e2]=⋄ {R}"

(* expressions *)

NewC "[G,A⊢{Normal P} .init C. {Alloc G (CInst C) Q}] ⇒
      G,A⊢{Normal P} NewC C-⋄ {Q}"

NewA "[G,A⊢{Normal P} .init_comp_ty T. {Q}; G,A⊢{Q} e-⋄
      {λVal:i:. xupd (check_neg i) .; Alloc G (Arr T (the_Intg i))
R}] ⇒
      G,A⊢{Normal P} New T[e]-⋄ {R}"

Cast "[G,A⊢{Normal P} e-⋄ {λVal:v:. λs..
      xupd (raise_if (¬G,s⊢v fits T) ClassCast) .; Q←Val v}] ⇒
      G,A⊢{Normal P} Cast T e-⋄ {Q}"

Inst "[G,A⊢{Normal P} e-⋄ {λVal:v:. λs..
      Q←Val (Bool (v≠Null ∧ G,s⊢v fits RefT T))}] ⇒
      G,A⊢{Normal P} e InstOf T-⋄ {Q}"

Lit "G,A⊢{Normal (P←Val v)} Lit v-⋄ {P}"

Super "G,A⊢{Normal (λs.. P←Val (val_this s))} Super-⋄ {P}"

Acc "[G,A⊢{Normal P} va=⋄ {λVar:(v,f):. Q←Val v}] ⇒
      G,A⊢{Normal P} Acc va-⋄ {Q}"

Ass "[G,A⊢{Normal P} va=⋄ {Q};
      ∀vf. G,A⊢{Q←Var vf} e-⋄ {λVal:v:. assign (snd vf) v .; R}] ⇒
      G,A⊢{Normal P} va:=e-⋄ {R}"

Cond "[G,A ⊢{Normal P} e0-⋄ {P'};
      ∀b. G,A⊢{P'←=b} (if b then e1 else e2)-⋄ {Q}] ⇒
      G,A⊢{Normal P} e0 ? e1 : e2-⋄ {Q}"

Call "[G,A⊢{Normal P} e-⋄ {Q}; ∀a. G,A⊢{Q←Val a} args⇒⋄ {R a};
      ∀a vs D l. G,A⊢{(R a←Vals vs ∧.
      (λs. D = target mode (snd s) a cT ∧ l = locals (snd s))
; .

```

```

      init_lvars G D (mn,pTs) mode a vs) ∧.
      (λs. normal s → G⊢mode→D⊢t)}
      Methd D (mn,pTs)-> {set_lvars l .; S}]] ==>
      G,A⊢{Normal P} {t,cT,mode}e..mn({pTs}args)-> {S}"

Methd "[[G,A∪ {{P} Methd-> {Q} | ms} |⊢ {{P} body G-> {Q} | ms}]] ==>
      G,A|⊢{{P} Methd-> {Q} | ms}"

Body "[[G,A⊢{Normal P} .init D. {Q}; G,A⊢{Q} .c. {R}; G,A⊢{R} e->
{S}]] ==>
      G,A⊢{Normal P} Body D c e-> {S}"

(* expression lists *)

Nil      "G,A⊢{Normal (P←Vals [])} [] ⊢> {P}"

Cons "[[G,A⊢{Normal P} e-> {Q};
      ∀ v. G,A⊢{Q←Val v} es ⊢> {λVals:vs:. R←Vals (v#vs)}]] ==>
      G,A⊢{Normal P} e#es ⊢> {R}"

(* statements *)

Skip      "G,A⊢{Normal (P←•)} .Skip. {P}"

Expr "[[G,A⊢{Normal P} e-> {Q←•}]] ==>
      G,A⊢{Normal P} .Expr e. {Q}"

Comp "[[G,A⊢{Normal P} .c1. {Q};
      G,A⊢{Q} .c2. {R}]] ==>
      G,A⊢{Normal P} .c1;;c2. {R}"

If "[[G,A ⊢{Normal P} e-> {P'};
      ∀ b. G,A⊢{P'←=b} .(if b then c1 else c2). {Q}]] ==>
      G,A⊢{Normal P} .If(e) c1 Else c2. {Q}"

(* unfolding variant of Loop, not needed here
LoopU "[[G,A ⊢{Normal P} e-> {P'};
      ∀ b. G,A⊢{P'←=b} .(if b then c;;While(e) c else Skip).{Q}]]
==>
      G,A⊢{Normal P} .While(e) c. {Q}"

*)

Loop "[[G,A⊢{P} e-> {P'}; G,A⊢{Normal (P'←=True)} .c. {P}]] ==>
      G,A⊢{P} .While(e) c. {(P'←=False)↓=•}"

Throw "[[G,A⊢{Normal P} e-> {λVal:a:. xupd (throw a) .; Q←•}]] ==>
      G,A⊢{Normal P} .Throw e. {Q}"

```

```

Try  "[[G,A⊢{Normal P} .c1. {SXAlloc G Q};
      G,A⊢{Q ∧. (λs. G,s⊢catch C) };. new_xcpt_var vn} .c2. {R}];
      (Q ∧. (λs. ¬G,s⊢catch C)) ⇒ R]] ⇒
      G,A⊢{Normal P} .Try c1 Catch(C vn) c2.
{R}"

Fin  "[[G,A⊢{Normal P} .c1. {Q};
      ∀x. G,A⊢{Q ∧. (λs. x = fst s) };. xupd (λx. None)}
      .c2. {xupd (xcpt_if (x≠None) x) .; R}]] ⇒
      G,A⊢{Normal P} .c1 Finally c2. {R}"

Done  "G,A⊢{Normal (P←• ∧. initd C)} .init C.
{P}"

Init  "[[the (class G C) = (sc,si,fs,ms,ini);
      G,A⊢{Normal ((P ∧. Not ◦ initd C) );. supd (init_class_obj G
C))}
      .(if C = Object then Skip else init sc). {Q};
      ∀l. G,A⊢{Q ∧. (λs. l = locals (snd s)) };. set_lvars empty}
      .ini. {set_lvars l .; R}]] ⇒
      G,A⊢{Normal (P ∧. Not ◦ initd C)} .init
C. {R}"

rules (** these terms are the same as above, but with generalized typing
**)
polymorphic_conseq
  "∀Y s Z . P Y s Z → (∃P' Q'. G,A⊢{P'} t> {Q'} ∧ (∀Y' s' .
  (∀Y Z' . P' Y s Z' → Q' Y' s' Z') →
  Q Y' s' Z ))
  ⇒ G,A⊢{P } t> {Q }"

polymorphic_Loop
  "[[G,A⊢{P} e-> {P'}]; G,A⊢{Normal (P'←=True)} .c. {P}]] ⇒
  G,A⊢{P} .While(e) c. {(P'←=False)↓=•}"

end

```

18 AxExample

```

(* Title:      isabelle/Bali/AxExample.thy
   ID:         $Id: AxExample.thy,v 1.6 2000/11/23 09:57:30 oheimb Exp

```

```

$
  Author:      David von Oheimb
  Copyright   2000 Technische Universitaet Muenchen
*)

AxExample = AxSem + Example +

constdefs
  arr_inv :: "st  $\Rightarrow$  bool"
  "arr_inv  $\equiv$   $\lambda$ s.  $\exists$ obj a T el. globs s (Stat Base) = Some obj  $\wedge$ 
                                snd obj (Inl (arr, Base)) = Some (Addr a)
 $\wedge$ 
                                heap s a = Some (Arr T #2,el)"
end

```

19 AxSound

```

(* Title:      isabelle/Bali/AxSound.thy
   ID:         $Id: AxSound.thy,v 1.11 2000/11/19 19:09:35 oheimb Exp
$
  Author:      David von Oheimb
  Copyright   1999 Technische Universitaet Muenchen

Soundness proof for Axiomatic semantics of Java expressions and statements
*)

AxSound = AxSem +

consts

  triple_valid2:: "prog  $\Rightarrow$  nat  $\Rightarrow$           'a triple  $\Rightarrow$  bool"
                                ( " _|=::_" [61,0, 58]
57)
  ax_valids2:: "prog  $\Rightarrow$  'a triples  $\Rightarrow$  'a triples  $\Rightarrow$  bool"
                                (" _,_|=::_" [61,58,58]
57)

defs triple_valid2_def "G|=n::t  $\equiv$  case t of {P} t> {Q}  $\Rightarrow$ 
 $\forall$ Y s Z. P Y s Z  $\longrightarrow$  ( $\forall$ L. s:: $\preceq$ (G,L)  $\longrightarrow$  ( $\forall$ T. (normal s  $\longrightarrow$  (G,L) $\vdash$ t::T)
 $\longrightarrow$ 
( $\forall$ Y' s'. G $\vdash$ s -t>-n  $\rightarrow$  (Y',s')  $\longrightarrow$  Q Y' s' Z  $\wedge$  s':: $\preceq$ (G,L))))"
```

```

defs ax_valids2_def "G,A| $\models$ ::ts  $\equiv \forall n. (\forall t \in A. G| $\models$ n::t) \longrightarrow (\forall t \in ts. G| $\models$ n::t)"$ 
```

```
end
```

20 AxCompl

```

(* Title:      isabelle/Bali/AxCompl.thy
   ID:         $Id: AxCompl.thy,v 1.32 2000/11/19 19:09:34 oheimb Exp
   $
   Author:     David von Oheimb
   Copyright  1999 Technische Universitaet Muenchen

```

Completeness proof for Axiomatic semantics of Java expressions and statements

design issues:

```

* proof structured by Most General Formulas (-> Thomas Kleymann)
*)

```

```
AxCompl = AxSem +
```

```
constdefs
```

```

  nyinitcls :: "prog  $\Rightarrow$  state  $\Rightarrow$  tname set"
  "nyinitcls G s  $\equiv$  {C. is_class G C  $\wedge$   $\neg$  initd C s}"

```

```

  init_le :: "prog  $\Rightarrow$  nat  $\Rightarrow$  state  $\Rightarrow$  bool"          ("_ $\vdash$ init $\leq$ _ " [51,51]
50)
  "G $\vdash$ init $\leq$ n  $\equiv$   $\lambda$ s. card (nyinitcls G s)  $\leq$  n"

```

```
consts (* Most General Triples and Formulas *)
```

```

  remember_init_state :: "state assn"                    (" $\doteq$ ")
  MGF :: "[state assn, term, prog]  $\Rightarrow$  state triple" ("{ $\_$ }  $\_>$  { $\_$   $\rightarrow$ }" [3,65,3]62)
  MGFn :: "[nat          , term, prog]  $\Rightarrow$  state triple" ("{=: $\_$ }  $\_>$  { $\_$   $\rightarrow$ }" [3,65,3]62)

```

```
defs
```

```
remember_init_state_def " $\doteq$   $\equiv$   $\lambda$ Y s Z. s = Z"
```

```
MGF_def
```

"{P} t > {G →} ≡ {P} t > {λY s' s. G ⊢ s - t > → (Y, s')}"

MGFn_def

"{=:n} t > {G →} ≡ {≡ ∧. G ⊢ init ≤ n} t > {G →}"

end