```
(*  Title:      isabelle/Bali/Basis.thy
    ID:         $Id: Basis.thy,v 1.15 1999/03/15 17:02:28 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Definitions extending HOL as logical basis of Bali
*)

Basis = Main +

(*#### patch for Isabelle98-1*)
syntax (xsymbols)
  "==>"        :: [prop, prop] => prop           (infixr "⟹" 1)
  "_bigimpl"   :: [asms, prop] => prop           ("((3⟦_⟧)/ ⟹ _)" [0, 1] 1)
  "op -->"     :: [bool, bool] => bool           (infixr "⟶" 25)
(**)

constdefs
          the_Inl  :: "'a + 'b ⇒ 'a"
         "the_Inl x ≡ εa. x = Inl a"
          the_Inr  :: "'a + 'b ⇒ 'b"
         "the_Inr x ≡ εb. x = Inr b"

datatype ('a, 'b, 'c, 'd) sum4 = In1 'a | In2 'b | In3 'c | In4 'd
constdefs
          the_In1  :: "('a, 'b, 'c, 'd) sum4 ⇒ 'a"
         "the_In1 x ≡ εa. x = In1 a"
          the_In2  :: "('a, 'b, 'c, 'd) sum4 ⇒ 'b"
         "the_In2 x ≡ εb. x = In2 b"
          the_In3  :: "('a, 'b, 'c, 'd) sum4 ⇒ 'c"
         "the_In3 x ≡ εc. x = In3 c"
          the_In4  :: "('a, 'b, 'c, 'd) sum4 ⇒ 'd"
         "the_In4 x ≡ εc. x = In4 c"

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 t ≡ ∀(x,y)∈set t. ∀(x',y')∈set t. x = x' ⟶ y = y'"

  list_all2 :: "('a ⇒ 'b ⇒ bool) => ('a list ⇒ 'b list ⇒ bool)"
 "list_all2 P xs ys ≡ length xs = length ys ∧ (∀(x,y)∈set (zip xs ys). P x y)"

end
```

```
(*  Title:      isabelle/Bali/Table.thy
    ID:         $Id: Table.thy,v 1.17 1999/01/28 21:01:21 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 system, 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 ⇀ 'b"
      ('a, 'b) tables   (* non-unique table with key 'a and contents 'b *)
      = "'a ⇒ '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"

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"

end
```

```
(*  Title:      isabelle/Bali/Type.thy
    ID:         $Id: Type.thy,v 1.13 1998/12/08 13:11:27 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Java types

simplifications:
* few primitive types yet
* arrays do not have special members like the length field
* the null type is regarded as reference type
*)

Type = Table +


(* for names, cf. 6.5 *)
types   tnam            (* type name, i.e. class or interface name *)
        enam    (* expression name, i.e. variable or field name *)
        mname           (* method name *)
arities tnam, enam, mname :: term

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

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

datatype ename  (* names for the This pointer and local variables *)
        = This
        | EName   enam


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 *)
```

```
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
```

```
(*  Title:      isabelle/Bali/Term.thy
    ID:         $Id: Term.thy,v 1.20 1999/02/22 09:14:37 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Java expressions and statements

design issues:
* local variables may alternatively be handled in a very uniform way, by
  representing the current invocation frame as an instance of a special pseudo
  class (one per method definition) and identifying a special location called
  StackTop (as opposed to Heap) pointing to it. We do not do so because:
  +  it is closer to the language and therefore easier to read
  +  there is no extra handling (e.g. invariants) for special constructs
     like StackTop needed
  -  there is more redundancy in the syntax and some semantic aspects
     of variable access and assignment
* for the transition semantics, the method call produces an intermediate method
  body construct that enables the restoration of the previous 'stack' contents.
* class initialization is regarded as (auxiliary) statement (required for AxSem)
simplifications:
* expression statement allowed for any expression
* no unary, binary, etc, operators
* This  is modelled as a special non-assignable local variable
* Super is modelled 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.
* no synchronized
* 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 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 = Type +

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

datatype val_ (** name not 'val' because of clash with ML token **)
        = 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_"
```

```
consts
  defpval       :: "prim_ty ⇒ val"      (* default value for primitive types *)
  default_val   :: "     ty ⇒ val"      (* default value for all types *)

primrec "defpval Void    = Unit"
        "defpval Boolean = Bool False"
        "defpval Integer = Intg #0"

primrec "default_val (PrimT pt) = defpval pt"
        "default_val (RefT  r ) = Null"

constdefs
  the_Bool   :: "val ⇒ bool"
 "the_Bool v ≡  εb. v = Bool b"
  the_Intg   :: "val ⇒ int"
 "the_Intg v ≡  εi. v = Intg i"
  the_Addr   :: "val ⇒ loc"
 "the_Addr v ≡  εa. v = Addr a"

types   locals                     (* local variables in state, see State.thy *)
        = "(ename, val) table"

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

datatype var
        = LVar                  ename(* 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 ty inv_mode expr mname (* method call *)
              (ty list) (expr list)   ("{_,_}_.._'({_}_')"[10,10,85,99,10,10]85)
(*      | Body stmt expr locals    (* 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 *)
end
```

```
(*  Title:      isabelle/Bali/Decl.thy
    ID:         $Id: Decl.thy,v 1.14 1998/12/08 13:11:24 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
* local variables are initialzed with default values

* 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 (except Object) has an explicit superclass
* the (standard) methods of Object and of system exceptions are not specified

* no packages
* no main method
*)

Decl = Term +

types   modi = bool     (* modifier: static *)

        field
        = "modi × ty"

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

types   sig             (* signature of a method, cf. 8.4.2  *)
        = "mname × ty list"

        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"
```

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

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

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

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

        class           (* class *)
        = "tname option × cbody"
        (* superclass *)

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

defs

 ObjectC_def    "ObjectC  ≡ (Object  ,  (None,[],[],Object_mdecls,Skip))"
 SXcptC_def     "SXcptC xn≡ (SXcpt xn, (Some (if xn = Throwable then Object else
                            SXcpt Throwable),[],[],SXcpt_mdecls,Skip))"

types prog = "idecl list × cdecl list"

syntax
  static    :: "modi  ⇒ bool"
  mrt       :: "mhead ⇒ ty"
  iface     :: "prog  ⇒ (tname, iface) table"
  class     :: "prog  ⇒ (tname, class) table"
  is_iface  :: "prog  ⇒ tname  ⇒ bool"
  is_class  :: "prog  ⇒ tname  ⇒ bool"

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

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

  (* auxiliary relations for recursive definitions below *)
  iface_wfrel,
  class_wfrel :: "((prog × tname) × (prog × tname)) set"

recdef iface_rec "iface_wfrel" congs "[image_cong]"
 "iface_rec (G,I) = (λ  f. if (∀(si,ib)∈iface G I: (∀J  ∈  set si.
                              ((G,J), (G,I)) ∈  iface_wfrel))
             then (case iface G I of None ⇒ arbitrary | Some (si,ib) ⇒
                   f I ib ((λJ. iface_rec (G,J) f)‘‘set si))
             else arbitrary)"
recdef class_rec "class_wfrel"
 "class_rec (G,C) = (λt f. if (∀(sc,cb)∈class G C: (∀D  ∈  sc:
                              ((G,D), (G,C)) ∈  class_wfrel))
             then (case class G C of None ⇒ arbitrary | Some (sc,cb) ⇒
                   f C cb (case sc of None ⇒ t | Some D ⇒ class_rec (G,D) t f))
             else arbitrary)"

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

defs
(* methods of an interface, with overriding and inheritance, cf. 9.2 *)
  imeths_def "imethds G I ≡ iface_rec (G,I)       (λI        ms      ts.
       (Un_tables ts) ⊕⊕  (o2s ∘ table_of (map (λ(s,m). (s,I,m)) ms)))"

(* methods of a class, with inheritance, overriding and hiding, cf. 8.4.6 *)
  cmethd_def "cmethd  G C ≡ class_rec (G,C) empty (λC (si,fs,ms,ini) ts.
                           ts ⊕ table_of (map (λ(s,m). (s,C,m)) ms))"

(* list of fields implemented by a class, including inherited and hidden ones *)
   fields_def "fields G C ≡ class_rec (G,C) []    (λC (si,fs,ms,ini) ts.
                                          map (λ(n,t). ((n,C),t)) fs @ ts)"

(* fields of a class, with inheritance and hiding, cf. 8.3 *)
  cfield_def  "cfield G C ≡ table_of((map (λ((n,d),t).(n,(d,t)))) (fields G C))"

end
```

```
(*  Title:      isabelle/Bali/TypeRel.thy
    ID:         $Id: TypeRel.thy,v 1.20 1999/02/04 21:58:30 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

The relations between Java types

simplifications:
* widening relation includes identity

improvements over Java Specification 1.0:
* narrowing reference conversion from a class to an interface only if
  any method that is common to both has a suitable return type
* narrowing reference conversion from an interface to another interface only if
  the interfaces are different

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,                                 (* direct subinterface   *)
  subint   :: "prog ⇒ (tname × tname) set"(*        subinterface   *)
  subcls1,                                 (* direct subclass       *)
  subcls   :: "prog ⇒ (tname × tname) set"(*        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) ∈ subint  G"
        "G|-C <:C1 D" == "(C,D) ∈ subcls1 G"
        "G|-C <:C  D" == "(C,D) ∈ subcls  G"
        "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, cf. 9.1.3 *)
  subint1_def   "subint1 G ≡ {(I,J). ∃i∈iface G I: J∈set (fst i)}"

  (* direct subclass, cf. 8.1.3 *)
  subcls1_def   "subcls1 G ≡ {(C,D). ∃c∈class G C: fst c = Some D}"

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


inductive "subint G" intrs (* subinterface relation, cf. 9.1.3 *)

  direct         "G⊢I≺I1K          ⟹ G⊢I≺I K"
  trans         "⟦G⊢I≺I J; G⊢J≺I K⟧ ⟹ G⊢I≺I K"


inductive "subcls G" intrs (* subclass relation, cf. 8.1.3 *)

  direct         "G⊢C≺C1E          ⟹ G⊢C≺C E"
  trans         "⟦G⊢C≺C D; G⊢D≺C E⟧ ⟹ G⊢C≺C E"


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

  direct         "G⊢C↝1J           ⟹ G⊢C↝J"
  subint        "⟦G⊢C↝1I; 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.[]"

(**
rules (* all properties of narrowing and casting conversions we actually need *)
      (* these can easily be proven from the above definitions *)

  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"
**)

(** 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 "o2s ∘ cmethd G C hidings imethds G I entails
         (λ(md,(mh,b)) (md',mh').G⊢mrt mh≼mrt mh') ⟹
         ¬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 ⟹  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
```

```
(*  Title:      isabelle/Bali/WellType.thy
    ID:         $Id: WellType.thy,v 1.22 1999/03/15 17:02:34 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 of array type

simplifications:
* the type rules include all static checks on expressions and statements, e.g.
  definedness of names (of parameters, locals, fields, methods)

design issues:
* unified type judgment for variables, expressions, expression lists, statements
* 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
        = "(ename, ty) table"   (* local variables, including This *)

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

        dyn_ty
        = "loc ⇒ ty option"

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

translations
  "prg" => "fst"
  "lcl" => "snd"

consts
  more_spec     :: "prog ⇒ (ty × mhead) × ty list ⇒
                           (ty × mhead) × ty list ⇒ bool"
  mheads        :: "prog ⇒  ref_ty ⇒ sig ⇒  (ty × mhead)            set"
  appl_methds   :: "prog ⇒  ref_ty ⇒ sig ⇒ ((ty × mhead) × ty list) set"
  max_spec      :: "prog ⇒  ref_ty ⇒ sig ⇒ ((ty × mhead) × ty list) set"

primrec
"mheads G   NullT    = (λsig. {})"
"mheads G (IfaceT I) = (op ‘‘)          (λ(I ,h   ).(Iface I,h)) ∘ imethds G I"
"mheads G (ClassT C) = o2s ∘ option_map (λ(C,(h,b)).(Class C,h)) ∘ cmethd  G C"
"mheads G (ArrayT T) = o2s ∘ option_map (λ(C,(h,b)).(Class C,h)) ∘ cmethd  G
                                                                          Object"
```

```
(** more detailed than necessary for type-safety, see below. *)
defs
  (* applicable methods, cf. 15.11.2.1 *)
  appl_methds_def "appl_methds G T ≡ λ(mn, pTs). {(mh,pTs') |mh pTs'.
        mh ∈ mheads G T (mn, pTs') ∧ list_all2 (λT T'. G⊢T≼T') pTs pTs'}"

  (* more specific methods, cf. 15.11.2.2 *)
  more_spec_def   "more_spec G ≡ λ((d,h),pTs). λ((d',h'),pTs'). G⊢d≼d'∧
                                list_all2 (λT T'. G⊢T≼T') pTs pTs'"

  (* maximally specific methods, cf. 15.11.2.2 *)
   max_spec_def   " max_spec G T sig ≡ {m. m ∈appl_methds G T sig ∧
                                        (∀m'∈appl_methds G T 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 T (mn, pTs) = insert (mh, pTs') A ⟹
      mh∈mheads G T (mn, pTs') ∧ list_all2 (λT T'. G⊢T≼T') pTs pTs'"
*)

consts
  typeof   :: "dyn_ty ⇒ val ⇒ ty option"
  empty_dt :: "dyn_ty"

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"
defs
 empty_dt_def   "empty_dt ≡ λa. None"

types terms = "(expr, var, expr list, stmt) sum4"
      tys   = "(ty  , ty , ty   list, unit) sum4"

consts
  wt    :: "(env × dyn_ty ×  terms × tys) set"
(*wt    :: " env ⇒ dyn_ty ⇒ (terms × tys) set" not feasible because of
                                                changing env in Try stmt *)

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

```
syntax (xsymbols)
wt      :: "env ⇒ dyn_ty ⇒ [terms,tys] ⇒ bool" ("_,_⊨_∷÷_" [51,51,51,51] 50)
ty_expr :: "env ⇒ dyn_ty ⇒ [expr ,ty ] ⇒ bool" ("_,_⊨_∷-_" [51,51,51,51] 50)
ty_var  :: "env ⇒ dyn_ty ⇒ [var  ,ty ] ⇒ bool" ("_,_⊨_∷=_" [51,51,51,51] 50)
ty_exprs:: "env ⇒ dyn_ty ⇒ [expr list,
                             ty   list] ⇒ bool" ("_,_⊨_∷≐_" [51,51,51,51] 50)
wt_stmt :: "env ⇒ dyn_ty ⇒  stmt       ⇒ bool" ("_,_⊨_∷◇"  [51,51,51   ] 50)

translations
        "E,dt⊨t∷÷T" == "(E,dt,t,T) ∈ wt"
        "E,dt⊨e∷-T" == "E,dt⊨In1 e∷÷In1 T"
        "E,dt⊨e∷=T" == "E,dt⊨In2 e∷÷In2 T"
        "E,dt⊨e∷≐T" == "E,dt⊨In3 e∷÷In3 T"
        "E,dt⊨s∷◇"  == "E,dt⊨In4 s∷÷In4()"

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

syntax (xsymbols)
  wt_      :: "env ⇒ [terms,tys] ⇒ bool" ("_⊢_∷÷_" [51,51,51] 50)
  ty_expr_ :: "env ⇒ [expr ,ty ] ⇒ bool" ("_⊢_∷-_" [51,51,51] 50)
  ty_var_  :: "env ⇒ [var  ,ty ] ⇒ bool" ("_⊢_∷=_" [51,51,51] 50)
  ty_exprs_:: "env ⇒ [expr list,
                      ty   list] ⇒ bool" ("_⊢_∷≐_" [51,51,51] 50)
  wt_stmt_::  "env ⇒ stmt        ⇒ bool" ("_⊢_∷◇"  [51,51   ] 50)

translations
        "E⊢t∷÷T" == "E,empty_dt⊨t∷÷T"
        "E⊢e∷-T" == "E,empty_dt⊨e∷-T"
        "E⊢e∷=T" == "E,empty_dt⊨e∷=T"
        "E⊢e∷≐T" == "E,empty_dt⊨e∷≐T"
        "E⊢s∷◇" == "E,empty_dt⊨s∷◇"

inductive wt intrs

(* 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 expressions *)

  (* cf. 15.8 *)
  NewC  "⟦is_type (prg E) (Class 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);
          class (prg E) C  = Some (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')};
      mode=(if static m then Static else if e=Super then SuperM else IntVir)⟧ ⟹
                                     E,dt⊨{md,mode}e..mn({pTs'}ps)::-rT"
```

```
(* 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"
(* well-typed statements *)

  Skip                                     "E,dt⊨Skip::◇"

  Expr  "⟦E,dt⊨e::-T⟧ ⟹
                                            E,dt⊨Expr e::◇"

  Comp  "⟦E,dt⊨c1::◇;
          E,dt⊨c2::◇⟧ ⟹
                                            E,dt⊨c1;; c2::◇"

  (* cf. 14.8 *)
  If    "⟦E,dt⊨e::-PrimT Boolean;
          E,dt⊨c1::◇;
          E,dt⊨c2::◇⟧ ⟹
                                            E,dt⊨If(e) c1 Else c2::◇"

  (* cf. 14.10 *)
  Loop  "⟦E,dt⊨e::-PrimT Boolean;
          E,dt⊨c::◇⟧ ⟹
                                            E,dt⊨While(e) c::◇"

  (* cf. 14.16 *)
  Throw "⟦E,dt⊨e::-Class tn;
          prg E⊢Class tn≼Class (SXcpt Throwable)⟧ ⟹
                                            E,dt⊨Throw e::◇"
  (* cf. 14.18 *)
  Try   "⟦E,dt⊨c1::◇; prg E⊢Class tn≼Class (SXcpt Throwable);
          lcl E vn = None; (prg E,lcl E(vn↦Class tn)),dt⊨c2::◇⟧ ⟹
                                            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::◇"


end
```

```
(*  Title:      isabelle/Bali/WellForm.thy
    ID:         $Id: WellForm.thy,v 1.18 1999/02/15 20:00:38 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 inherites more than one method with the same signature, the
  methods need not have identical return types

simplifications:
* for uniformity, Object is assumed to be declared like any other class
*)
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 ∧
                           This∉set pns ∧ nodups pns"

  wf_mdecl_def  "wf_mdecl G C ≡ λ((mn,pTs),(m,pns,rT),lvars,blk,res).
                (C = Object ⟶ ¬static m) ∧
                (let  lt   = table_of lvars; ltab = table_of lvars(pns[↦]pTs);
                      E    = (G,if static m then ltab else ltab(This↦Class C))
                 in wf_mhead G    (mn,pTs) (m,pns,rT) ∧ unique lvars ∧
                    lt This = None ∧ (∀pn∈set pns. lt pn = None) ∧
                    (∀(vn,T)∈set lvars. is_type G T)∧
                    E⊢blk::◇ ∧ (∃T. E⊢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,(is,ms)). ¬is_class G I ∧
                (∀J    ∈set is. is_iface G J)   ∧
                (∀J    ∈set is. ¬ G⊢J≺I I)     ∧
                (∀(s,mh)∈set ms. wf_mhead G s mh ∧ ¬static (fst mh)) ∧
                unique ms ∧
                (o2s ∘ table_of ms hidings Un_tables((λJ.(imethds G J))``set is)
                 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)) ∈  cmethd  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::◇ ∧
              (case sc of None ⇒ C = Object | Some D ⇒
                         is_class G D ∧ ¬ G⊢D≺C C ∧
                         (table_of ms hiding cmethd G D
                          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 = set (fst G); cs = set (snd G) in
                       ObjectC ∈  cs  ∧ (∀xn. SXcptC xn ∈  cs) ∧
              (∀i∈is. wf_idecl G i) ∧ unique (fst G) ∧
              (∀c∈cs. wf_cdecl G c) ∧ unique (snd G)"

defs

 (* well-founded relations for termination proofs for iface_rec and class_rec *)
 iface_wfrel_def "iface_wfrel ≡ {((G,I),(G',I')). G = G' ∧
                               wf ((subint1 G)^-1) ∧ (I,I')∈(subint1 G)^-1}"
 class_wfrel_def "class_wfrel ≡ {((G,C),(G',C')). G = G' ∧
                               wf ((subcls1 G)^-1) ∧ (C,C')∈(subcls1 G)^-1}"

end
```

```
(*  Title:      isabelle/Bali/State.thy
    ID:         $Id: State.thy,v 1.28 1999/04/04 17:11:11 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 handeled 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)
* unfortunately new_Addr is not directly executable because of Hilbert operator.
* lconf allows for (arbitrary) inaccessible values

simplifications:
* the stack of invocation frames containing local variables and return addresses
  for method calls is not explicitly modeled, nor finiteness constrains on it.
* garbage collection not considered, therefore also no finalizers

*)

State = WellForm +

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

types   oref = "loc + tname"          (* generalized object reference *)
        fn   = "ename × tname"                  (* field    name      *)
        vn   = "fn + int"                       (* variable name      *)
        obj  = "obj_tag × (vn, val) table"      (* generalized object *)

constdefs

  the_obj_class :: "obj option ⇒ tname"
 "the_obj_class obj ≡ case fst (the obj) of CInst C ⇒ C | Arr T k ⇒ Object"

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

constdefs

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

  fields_table  :: "prog ⇒ tname ⇒ (modi ⇒ bool) ⇒ (fn, ty) table"
 "fields_table G C P ≡ option_map snd ∘ table_of([(fn,(m,fT))∈fields G C. P m])"

syntax

  Heap  :: "loc   ⇒ oref"
  Stat  :: "tname ⇒ oref"
```

```
translations

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

constdefs

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

  var_tys       :: "prog ⇒ obj_tag ⇒ oref ⇒ (vn, ty) table"
 "var_tys G oi r ≡ case r of Heap a ⇒ (case oi of
             CInst C ⇒ sum_case (fields_table G C (Not o static)) empty
           | Arr T k ⇒ sum_case empty (λi. if #0≤i∧i<k then Some T else None))
           | Stat C  ⇒ sum_case (fields_table G C        static ) empty"

syntax

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

translations

 "init_vars vs"    == "option_map default_val ∘ vs"

types   globs                  (* global variables: heap and static variables *)
        = "(oref , obj) table"
(*      locals
        = "(ename, val) table" (* local variables, in Term.thy *)    *)

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 ⇒ (loc  , obj) table"
 "heap s ≡ globs s ∘ Heap"

  stat   :: "st ⇒ (tname, obj) table"
 "stat s ≡ globs s ∘ Stat"


constdefs

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

  lupd      :: "ename ⇒ 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)"

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

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

consts

  the_XcptLoc :: "xcpt ⇒ loc"
  the_SysXcpt :: "xcpt ⇒ xname"

defs

  the_XcptLoc_def "the_XcptLoc xc ≡ εa. xc = XcptLoc a"
  the_SysXcpt_def "the_SysXcpt xc ≡ εx. xc = SysXcpt x"

types

        state           (* state including exception information *)
        = "xcpt option × st"

syntax

  Norm  :: "st ⇒ state"

translations

  "Norm s"  == "(None,s)"
(*###
  "st"      <= (type)"(Term.loc + Type.tname ⇀ State.obj_tag ×
                      (Type.ename × Type.tname + IntDef.int ⇀ Term.val_)) ×
                      (Type.ename ⇀ Term.val_)"
*)
  "state"   <= (type)"State.xcpt Option.option × State.st"


constdefs

  supd  :: "(st ⇒ st) ⇒ state ⇒ state"
 "supd ≡ prod_fun id"

  xupd  :: "(state ⇒ state) ⇒ state ⇒ state"
 "xupd f ≡ λ(x,s). let (x',s') = (if x = None then f else id) (x,s)
                   in  (x',if x' = None then s' else s)"

  fits          :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool" ("_,_⊢_ fits _"[61,61,61,61]60)
  "G,s⊢v fits T  ≡ (∃pt. T=PrimT pt) ∨ v=Null ∨
                  G⊢obj_ty (the (heap s (the_Addr v)))≼T"

  xcpt_if       :: "bool ⇒ xcpt option ⇒ xcpt option ⇒ xcpt option"
 "xcpt_if c x xo≡ if c ∧ (xo = None) then x else xo"
```

```
  new_Addr      :: "(loc, obj) table ⇒ loc option"
 "new_Addr h ≡ if (∀a. h a ≠ None) then None else Some (εa. h a = None)"

  enough_Addrs  :: "(loc, obj) table ⇒ loc ⇒ bool"
 "enough_Addrs h a ≡ ∃b. b ≠ a ∧ h b = None"

syntax

  raise_if :: "bool ⇒ xname ⇒ xcpt option ⇒ xcpt option"
  np       :: "val          ⇒ xcpt option ⇒ xcpt option"

translations

 "raise_if c xn" ==  "xcpt_if c (Some (SysXcpt xn))"
 "np v0"         == "raise_if (v0 = Null) NullPointer"

end
```

```
(*  Title:      isabelle/Bali/Eval.thy
    ID:         $Id: Eval.thy,v 1.40 1999/04/04 17:11:09 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 check the return type (cf. 15.11.4.4)
* throw raises a NullPointer exception if a null reference is given, and each
  throw of a system 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
* a method call in virtual mode upon a null reference initializes class Object
  (was not specified)
* fixed exact positions of class initializations (at first active use)
* during class initialization, after initializing the superclass, it is checked
  again whether the class still needs to be initialized (was not specified)

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 necessary as opposed to trans. sem.
  +  convenient rule induction for subject reduction theorem
  -  no interleaving (for parallelism) describable
  -  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 occurence.
* 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.
  ++ faciliates 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 exacly 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 system 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

simplifications:
* stack overflow and memory overflow during class initialization not modelled
* do not replace exceptions in initializations by ExceptionInInitializerError
*)
Eval = State +

types vals  = "(val, val × (val⇒state⇒state), val list, unit) sum4"
translations
      "vals"    <= (type)"(Term.val_, Term.val_ × (Term.val_⇒state⇒state),
                           Term.val_ List.list, Prod.unit) Basis.sum4"

consts
  eval  :: "prog ⇒ (state × terms   × vals × state) set"
  halloc:: "prog ⇒ (state × obj_tag × loc  × state) set"
  salloc:: "prog ⇒ (state                  × state) set"

syntax
eval ::"[prog,state,terms,vals*state]=>bool"("_|-_ -_>-> _"  [61,61,80,   61]60)
evar ::"[prog,state,var  ,val * val => state => state,
                               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)
exec ::"[prog,state,stmt      ,state]=>bool"("_|-_ -_-> _"  [61,61,65,   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)
eval ::"[prog,state,terms,vals×state]⇒bool"("_⊢_ −_≻→ _"  [61,61,80,   61]60)
evar ::"[prog,state,var  ,val × val ⇒ state ⇒ state,
                               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)
exec ::"[prog,state,stmt      ,state]⇒bool"("_⊢_ −_→ _"  [61,61,65,   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)

translations
  "G⊢s −e-≻v  →  (x,s')" <= "G⊢s −In1 e≻→ (In1 v ,_args x s' )"
  "G⊢s −e-≻v  →      s' " == "G⊢s −In1 e≻→ (In1 v ,        s' )"
  "G⊢s −e=≻vf →  (x,s')" <= "G⊢s −In2 e≻→ (In2 vf,_args 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 ,_args x s' )"
  "G⊢s −e≐≻v  →      s' " == "G⊢s −In3 e≻→ (In3 v ,        s' )"
  "G⊢s −c     →  (x,s')" <= "G⊢s −In4 c≻→ (In4 (),_args x s' )"
  "G⊢s −c     →      s' " == "G⊢s −In4 c≻→ (In4 (),        s' )"
  "G⊢s −t          ≻→ (w,s')" <= "(s,t,   (_args w s')) ∈ eval G"
  "G⊢s −t          ≻→   ws' " == "(s,t,           ws' ) ∈ eval G"
  "G⊢s −halloc oi≻a→ (x,s')" <= "(s,oi,a,(_args x s')) ∈ halloc G"
  "G⊢s −halloc oi≻a→    s' " == "(s,oi,a,          s') ∈ halloc G"
```

```
  "G⊢s      −salloc→ (x,s')" <= "(s     ,(_args x s')) ∈ salloc G"
  "G⊢s      −salloc→    s' " == "(s     ,          s') ∈ salloc G"

inductive "halloc G" intrs (* allocating objects on the heap *)

  Xcpt  "G⊢(Some x,s) −halloc oi≻arbitrary→ (Some x,s)"

  New   "⟦new_Addr (heap s) = Some a;
         (x,oi') = (if enough_Addrs (heap s) a then (None ,oi)
                    else (Some (XcptLoc a),CInst (SXcpt OutOfMemory)))⟧ ⟹
         G⊢Norm s −halloc oi≻a→ (x,new_obj G oi' (Heap a) s)"

inductive "salloc G" intrs (* allocating exception objects for
                              system exceptions (other than OutOfMemory) *)

  Norm   "G⊢ Norm              s   −salloc→  Norm             s"

  XcptL  "G⊢(Some (XcptLoc a ),s)  −salloc→ (Some (XcptLoc a),s)"

  SXcpt "⟦G⊢Norm s0 −halloc (CInst (SXcpt xn))≻a→ (x,s1)⟧ ⟹
         G⊢(Some (SysXcpt xn),s0) −salloc→ (Some (XcptLoc a),s1)"


inductive "eval G" intrs

(* propagation of exceptions *)

  (* cf. 15.5 *)
  XcptV "G⊢(Some xc,s) −v=≻arbitrary→ (Some xc,s)"
  XcptE "G⊢(Some xc,s) −e-≻arbitrary→ (Some xc,s)"
  XcptEs"G⊢(Some xc,s) −e≐≻arbitrary→ (Some xc,s)"

  (* cf. 14.1 *)
  XcptS "G⊢(Some xc,s) −c            → (Some xc,s)"


(* evaluation of variables *)

  (* cf. 15.13.1, 15.7.2 *)
  LVar  "G⊢Norm s −LVar vn=≻(the (locals s vn),
                         (λv. supd (lupd(vn↦v))))→ Norm s"

  (* cf. 15.10.1, 12.4.1 *)
  FVar  "⟦G⊢Norm s0 −init C→ s1; G⊢s1 −e-≻a'→ (x2,s2); a = the_Addr a';
        (oref,x) = (if s then (Stat C,x2) else (Heap a,np a' x2));
         n = Inl (fn,C); f = (λv. supd (upd_gobj oref n v))⟧ ⟹
         G⊢Norm s0 −{C,s}e..fn=≻(the(snd (the (globs s2 oref)) n),f)→ (x,s2)"

  (* cf. 15.12.1, 15.25.1 *)
  AVar  "⟦G⊢ Norm s0 −e1-≻a'→     s1 ; a = the_Addr a';
         G⊢       s1 −e2-≻i'→ (x2,s2); i = the_Intg i'; n = Inr i;
         (T,k,cs) = the_Arr (heap s2 a); f = (λv (x,s).
              (raise_if (¬G,s⊢v fits T) ArrStore x, upd_gobj (Heap a) n v s));
         x2' = raise_if (¬(#0≤i ∧ i<k)) IndOutBound (np a' x2)⟧ ⟹
                    G⊢Norm s0 −e1.[e2]=≻(the (cs n),f)→ (x2',s2)"
```

```
(* 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  "⟦C = (if (∃C. T = Class C) then the_Class T else Object);
         G⊢Norm s0 −init C→ s1; G⊢s1 −e-≻i'→ (x2,s2);
         k = the_Intg i'; x2' = raise_if (k<#0) NegArrSize x2;
         G⊢(x2',s2) −halloc (Arr T k)≻a→ s3⟧ ⟹
                                G⊢Norm s0 −New T[e]-≻Addr a→ s3"

  (* cf. 15.15 *)
  Cast  "⟦G⊢Norm s0 −e-≻v→ (x1,s1);
         x2 = raise_if (¬G,s1⊢v fits T) ClassCast x1⟧ ⟹
                                G⊢Norm s0 −Cast T e-≻v→ (x2,s1)"

  (* 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-≻the (locals s This)→ 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→ xupd (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, 14.15, 12.4.1 *)
  Call  "⟦G⊢Norm s0 −e-≻a'→ s1; G⊢s1 −ps≐≻pvs→ (x2,s2); a = the_Addr a';
         C = (if mode=IntVir then the_obj_class (heap s2 a) else the_Class cT);
         (md,(m,pns,rT),lvars,blk,res) = the (cmethd G C (mn,pTs));
         G⊢(x2,s2) −init(if mode=IntVir ∧ a'=Null then Object else md)→(x3,s3);
         vs = init_vars (table_of lvars)(pns[↦]pvs);
         (x,l) = (if mode = Static then (x3,vs) else (np a' x3,vs(This↦a')));
         G⊢(x,st (globs s3) l) −blk→ s4; G⊢s4 −res-≻v→ (x5,s5)⟧ ⟹
              G⊢Norm s0 −{cT,mode}e..mn({pTs}ps)-≻v→ (x5,st (globs s5) (locals s3))"
```

```
(* evaluation of expression lists *)

  (* cf. 15.11.??? *)
  Nil
                                    "G⊢Norm s0 -[]≐≻[]→ Norm s0"

  (* cf. 15.6.4 *)
  Cons  "⟦G⊢Norm s0 -e -≻ v → s1;
          G⊢     s1 -es≐≻vs→ s2⟧ ⟹
                                   G⊢Norm s0 -e#es≐≻v#vs→ s2"


(* 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-≻v→ s1;
          G⊢     s1-(if the_Bool v 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-≻v→ s1;
          G⊢s1 -(if the_Bool v then          c else Skip)→ s2;
          G⊢s2 -(if the_Bool v then While(e) c else Skip)→ s3⟧ ⟹
                              G⊢Norm s0 -While(e) c→ s3"

  (* cf. 14.16 *)
  Throw "⟦G⊢Norm s0 -e-≻a'→ (x1,s1);
          x1' = xcpt_if True (Some (XcptLoc (the_Addr a'))) (np a' x1)⟧ ⟹
                                 G⊢Norm s0 -Throw e→ (x1',s1)"

  (* cf. 14.18.1 *)
  Try   "⟦G⊢Norm s0 -c1→ s1; G⊢s1 -salloc→ (x1',s1');
          a     = Addr (the_XcptLoc (the x1'));
          catch = (∃xc. x1' = Some xc ∧ G,s1'⊢a fits Class tn);
          s1''  = (if catch then Norm (lupd(vn↦a) s1') else (x1',s1'));
          G⊢s1'' -(if catch then c2 else Skip)→ s2⟧ ⟹
                  G⊢Norm s0 -Try c1 Catch(tn vn) c2→ s2"

  (* cf. 14.18.2 *)
  Fin   "⟦G⊢Norm s0 -c1→ (x1,s1);
          G⊢Norm s1 -c2→ (x2,s2)⟧ ⟹
                   G⊢Norm s0 -c1 Finally c2→ (xcpt_if (x1≠None) x1 x2,s2)"
```

```
  (* cf. 8.5, 12.4.1 , 12.4.2 *)
  Init  "⟦the (class G C) = (sco,si,fs,ms,ini);
         G⊢Norm s0 −(case sco of None ⇒ Skip | Some sc ⇒
                         (if stat s0 C = None then init sc else Skip))→ (x1,s1);
         still = (stat s1 C = None ∧ x1 = None);
         s1' = new_obj G arbitrary (Stat C) s1;
         G⊢(x1,(if still then st (globs s1') empty else s1))
              −(if still then ini               else Skip)→ (x2,s2)⟧ ⟹
                G⊢Norm s0 −init C→ (x2,st (globs s2) (locals s1'))"

end
```

```
(*  Title:      isabelle/Bali/TypeSafe.thy
    ID:         $Id: TypeSafe.thy,v 1.9 1999/03/15 17:02:32 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

conformance notions for
the type soundness proof for evaluation semantics of Java

design issues:
"conforms" does not directly imply that the dynamic types of all objects on the
heap are indeed existing classes. This can be inferred for all referenced objs.
*)

TypeSafe = Eval +

constdefs

  gext  :: "st ⇒ st ⇒ bool"                  ("_≤|_"     [61,61]       60)
   "s≤|s' ≡ (∀r. ∀(oi, fs)∈globs s r: ∃(oi',fs')∈globs s' r: oi' = oi)"

  conf  :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool"     ("_,_⊢_::≼_"   [61,61,61,61] 60)
          "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"
                                              ("_,_⊢_[::≼]_"[61,61,61,61] 60)
          "G,s⊢vs[::≼]Ts ≡ ∀n. ∀T∈Ts n: ∃v∈vs n: G,s⊢v::≼T"

  oconf :: "prog ⇒ st ⇒ obj ⇒ oref ⇒ bool"  ("_,_⊢_::≼◇_"  [61,61,61,61] 60)
          "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"          (     "_::≼_"   [61,61]      60)
             "xs::≼E ≡ let G = prg E; L = lcl 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
```

```
(*  Title:      isabelle/Bali/Example.thy
    ID:         $Id: Example.thy,v 1.10 1999/04/04 17:11:10 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

The following example Bali program includes:
 class declarations with inheritance, hiding of fields, and overriding of
  methods (with refined result type),
 instance creation, local assignment, sequential composition,
 try & catch, method call with dynamic binding, literal values,
 expression statement, local access, type cast, field assignment (in part), skip

class Base {
  boolean vee;
  Base foo(Base x) {return x;}
}

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

class Example {
  public static void main (String args[]) {
    Base e;
    e=new Ext();
    try {e.foo(null); }
    catch (NullPointerException x) {}
  }
}
*)

Example = Eval +

datatype tnam_ = Base_ | Ext_
datatype enam_ = vee_ | x_ | e_

consts

  tnam_ :: "tnam_ ⇒ tnam"
  enam_ :: "enam_ ⇒ enam"

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

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

  surj_tnam_ "∃m. n = tnam_ m"
  surj_enam_ "∃m. n = enam_ m"

defs

  Object_mdecls_def "Object_mdecls == []"
  SXcpt_mdecls_def  "SXcpt_mdecls  == []"
```

```
syntax

  Base,  Ext :: tname
  vee, x, e  :: ename

translations

  "Base" == "TName (tnam_ Base_)"
  "Ext"  == "TName (tnam_ Ext_)"
  "vee"  == "EName (enam_ vee_)"
  "x"    == "EName (enam_ x_)"
  "e"    == "EName (enam_ e_)"


consts

  system_classes :: cdecl list
  tprg           :: prog
  BaseC, ExtC    :: cdecl
  test           :: stmt
  foo            :: mname
  a,b            :: loc

defs

  BaseC_def "BaseC ≡ (Base, (Some Object, [],
                            [(vee, (False, PrimT Boolean))],
                            [((foo,[Class Base]),(False,[x],Class Base),
                              ([],Skip,Acc(LVar x)))],
                             Skip))"
  ExtC_def  "ExtC  ≡ (Ext,  (Some Base  , [],
                            [(vee, (False,PrimT Integer))],
                            [((foo,[Class Base]),(False,[x],Class Ext),
                              ([],Expr( {Ext,False}Cast (Class Ext)
                                       (Acc (LVar x))..vee:=Lit (Intg #1)),
                                  Lit Null))],
                             Skip))"

  system_classes_def "system_classes ≡ [ObjectC, SXcptC Throwable,
                SXcptC NullPointer, SXcptC OutOfMemory, SXcptC ClassCast,
                SXcptC NegArrSize , SXcptC IndOutBound, SXcptC ArrStore]"

  tprg_def "tprg ≡ ([],system_classes@[BaseC,ExtC])"

  test_def "test ≡ Expr(LVar e:=NewC Ext);;
                  Try Expr( {Class Base,IntVir}Acc (LVar e)..
                            foo({[Class Base]}[Lit Null]))
                  Catch((SXcpt NullPointer) x) (Throw (Acc (LVar x)))"
```

```
syntax

  NP                          :: xname
  "classes"                   :: cdecl list
  obj1, obj2                  :: obj
  stats,globs1,globs2         :: (oref , obj) table (* globs *)
  s0',s0,s1,s2,s3,s4,s5,s6,s7:: state

translations

  "NP"      == "NullPointer"
  "classes" == "[ObjectC, SXcptC Throwable,
               SXcptC NP         , SXcptC OutOfMemory, SXcptC ClassCast,
               SXcptC NegArrSize , SXcptC IndOutBound, SXcptC ArrStore ,
               BaseC, ExtC]"
(*"obj1"    == "(CInst Ext,sum_case (empty((vee, Base)↦Bool False)
                                        ((vee, Ext )↦####Intg #0)) empty)"*)
  "obj1"    <= "(CInst Ext,sum_case (empty((vee, Base)↦Bool False)
                                        ((vee, Ext )↦Intg #0)) empty)"
  "obj2"    == "(CInst(SXcpt NP),empty)"
  "stats"   == "empty(Inr Object↦(arbitrary, empty))
                   (Inr Base  ↦(arbitrary, empty))
                   (Inr Ext   ↦(arbitrary, empty))"
  "globs1"   == "stats (Inl a↦obj1)"
  "globs2"   == "globs1(Inl b↦obj2)"
  "s0'"== " Norm             (st empty   empty                     )"
  "s0" == " Norm             (st stats   empty                     )"
  "s1" == " Norm             (st globs1 (empty(e↦Addr a)           ))"
  "s2" == " Norm             (st globs1 (empty(This↦Addr a)(x↦Null)))"
  "s3" == "(Some (SysXcpt NP), st globs1 (empty(This↦Addr a)(x↦Null)))"
  "s4" == "(Some (SysXcpt NP), st globs1 (empty(e↦Addr a)          ))"
  "s5" == "(Some (XcptLoc b ), st globs2 (empty(e↦Addr a)          ))"
  "s6" == " Norm             (st globs2 (empty(e↦Addr a)(x↦Addr b) ))"
  "s7" == "(Some (XcptLoc b ), st globs2 (empty(e↦Addr a)(x↦Addr b) ))"

end
```