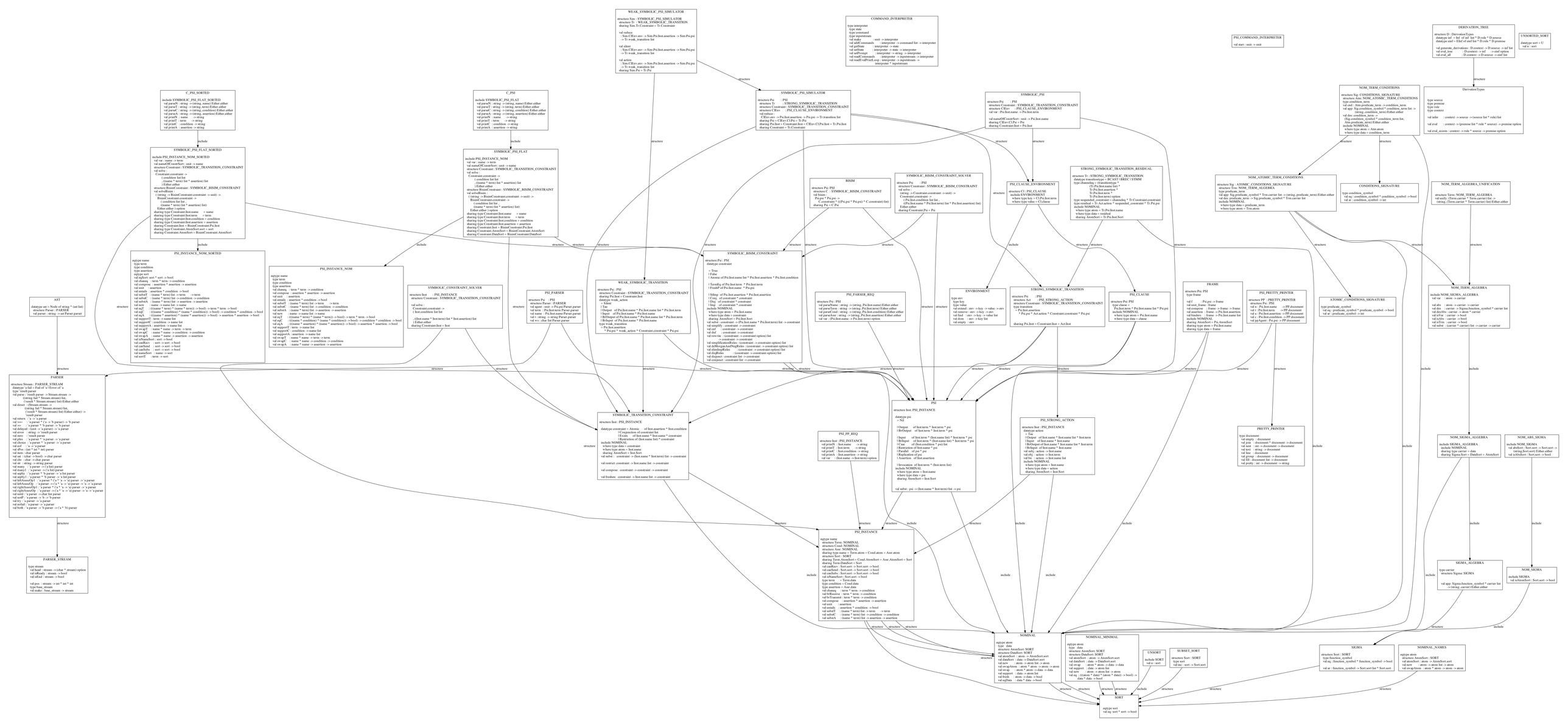
WEAK_SYMBOLIC_PSI_SIMULATOR
structure Sim : SYMBOLIC_PSI_SIMULATOR
structure Tr : WEAK_SYMBOLIC_TRANSITION
sharing Sim.Tr.Constraint = Tr.Constraint
val reduce
: Sim.CElev env -> Sim.Psi.Inst.assertion -> Sim.Psi.psi
-> Tr.weak_transition list
val silent
: Sim.CElev env -> Sim.Psi.Inst.assertion -> Sim.Psi.psi
-> Tr.weak_transition list
val tau
: Sim.CElev env -> Sim.Psi.Inst.assertion -> Sim.Psi.psi
-> Tr.weak_transition list
sharing Sim.Psi = Tr.Psi
COMMAND_INTERPRETER
type interpreter
type state
type command
type inputstream
val make : unit -> interpreter
val addCommands : interpreter -> command list -> interpreter
val getState : interpreter -> state
val setState : interpreter -> state -> interpreter
val setPrompt : interpreter -> string -> interpreter
val readCommands : interpreter -> inputstream -> interpreter
val readEvalPrintLoop : interpreter -> inputstream ->
interpreter * inputstream
PSI_COMMAND_INTERPRETER
val start : unit -> unit
DERIVATION_TREE
structure D : DerivationTypes
datatype tnf = Inf of inf Set * D.rule * D.source
datatype inf = Rel of nnf list * D.rule * D.premise
val generate_derivations : D.context -> D.source -> inf list
val eval_tree : D.context -> inf -> unit option
val eval_all : D.context -> D.source -> unit list
UNSORTED_SORT
datatype sort = U
val u : sort
structure
DerivationTypes
type source
type premise
type rule
type context
val infer : context -> source -> (source list * rule) list
val eval : context -> (premise list * rule * source) -> premise option
val eval_axiom : context -> rule * source -> premise option
C_PSI_SORTED
include SYMBOLIC_PSI_FLAT_SORTED
val parseN : string -> (string, name) Either either
val parseC : string -> (string, condition) Either either
val parseC : string -> (string, assertion) Either either
val parseA : string -> (string, assertion) Either either
val printN : name -> string
val printT : term -> string
val printC : condition -> string
val printA : assertion -> string
include
SYMBOLIC_PSI_FLAT_SORTED
include PSI_INSTANCE_NOM_SORTED
val var : name -> term
val nameOfCounterSort : unit -> name
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
val solve :
Constraint.constraint ->
(condition list list
, (((name * term) list * assertion) list
) Either.either
structure BisimConstraint : SYMBOLIC_BISIM_CONSTRAINT
val solveBisim :
((string -> BisimConstraint.constraint -> unit) ->
BisimConstraint.constraint ->
(condition list list
, ((name * term) list * assertion) list)
) Either.either option
sharing type Constraint.Inst.name = name
sharing type Constraint.Inst.term = term
sharing type Constraint.Inst.condition = condition
sharing type Constraint.Inst.assertion = assertion
sharing type Constraint.Inst.Set.set = Constraint.Inst.list
sharing type Constraint.AtomSort.sort = sort
sharing Constraint.AtomSort = BisimConstraint.AtomSort
include
PSI_INSTANCE_NOM_SORTED
eqtype name
type term
type condition
type assertion
eqtype sort
val sqSort: sort * sort -> bool
val chooseq : term * term -> condition
val assertion : assertion -> assertion
val unit : assertion
val entails : assertion * condition -> bool
val subeT : (name * term) list -> term -> term
val subeC : (name * term) list -> condition -> condition
val subeA : (name * term) list -> assertion -> assertion
val new : name -> name list -> name
val eqT : (((name * term) * (name * term)) -> bool) -> term * term -> bool
val eqC : (((name * condition) * (name * condition)) -> bool) -> condition * condition -> bool
val eqA : (((name * assertion) * (name * assertion)) -> bool) -> assertion * assertion -> bool
val supportsT : term -> name list
val supportsC : condition -> name list
val supportsA : assertion -> name list
val swapT : (name * name) -> term -> term
val swapC : (name * name) -> condition -> condition
val swapA : (name * name) -> assertion -> assertion
val isNumsSort : sort -> bool
val canRecv : sort -> sort -> bool
val canSend : sort -> sort -> bool
val canSubs : sort -> sort -> bool
val unifySort : sort -> sort
val sortT : term -> sort
C_PSI
include SYMBOLIC_PSI_FLAT
val parseN : string -> (string, name) Either either
val parseC : string -> (string, condition) Either either
val parseC : string -> (string, assertion) Either either
val parseA : string -> (string, assertion) Either either
val printN : name -> string
val printT : term -> string
val printC : condition -> string
val printA : assertion -> string
include
SYMBOLIC_PSI_FLAT_SORTED
include PSI_INSTANCE_NOM
val var : name -> term
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
val solve :
Constraint.constraint ->
(condition list list
, ((name * term) list * assertion) list
) Either.either
structure BisimConstraint : SYMBOLIC_BISIM_CONSTRAINT
val solveBisim :
((string -> BisimConstraint.constraint -> unit) ->
BisimConstraint.constraint ->
(condition list list
, ((name * term) list * assertion) list)
) Either.either option
sharing type Constraint.Inst.name = name
sharing type Constraint.Inst.term = term
sharing type Constraint.Inst.condition = condition
sharing type Constraint.Inst.assertion = assertion
sharing Constraint.Inst = BisimConstraint.Psi.Inst
sharing Constraint.AtomSort = BisimConstraint.AtomSort
sharing Constraint.DataSort = BisimConstraint.DataSort
include
PSI_INSTANCE_NOM_SORTED
eqtype name
type term
type condition
type assertion
eqtype sort
val sqSort: sort * sort -> bool
val chooseq : term * term -> condition
val assertion : assertion -> assertion
val unit : assertion
val entails : assertion * condition -> bool
val subeT : (name * term) list -> term -> term
val subeC : (name * term) list -> condition -> condition
val subeA : (name * term) list -> assertion -> assertion
val new : name -> name list -> name
val eqT : (((name * term) * (name * term)) -> bool) -> term * term -> bool
val eqC : (((name * condition) * (name * condition)) -> bool) -> condition * condition -> bool
val eqA : (((name * assertion) * (name * assertion)) -> bool) -> assertion * assertion -> bool
val supportsT : term -> name list
val supportsC : condition -> name list
val supportsA : assertion -> name list
val swapT : (name * name) -> term -> term
val swapC : (name * name) -> condition -> condition
val swapA : (name * name) -> assertion -> assertion
SYMBOLIC_PSI_SIMULATOR
structure Psi : PSI
structure Tr : STRONG_SYMBOLIC_TRANSITION
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
structure CElev : PSI_CLAUSE_ENVIRONMENT
val reduce :
CElev.env -> Psi.Inst.assertion -> Psi.psi -> Tr.transition list
sharing Psi = CElev.CtPsi = Tr.Psi
sharing Psi.Inst = Constraint.Inst = CElev.CtPsi.Inst = Tr.Psi.Inst
sharing Constraint = Tr.Constraint
SYMBOLIC_PSI
structure Psi : PSI
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
structure CElev : PSI_CLAUSE_ENVIRONMENT
val var : Psi.Inst.name -> Psi.Inst.term
val nameOfCounterSort : unit -> Psi.Inst.name
sharing CElev.CtPsi = Psi
sharing Constraint.Inst = Psi.Inst
NOM_TERM_CONDITIONS
structure Sig: CONDITIONS_SIGNATURE
structure Term: ATOMIC_TERM_CONDITIONS
type condition_term
val eval : Atm.predicate_term -> condition_term
val app: Sig.condition_symbol * condition_term list -> (string, condition_term) Either.either
include NOMINAL
val dec: condition_term -> (Sig.condition_symbol * condition_term list, Atm.predicate_term) Either.either
include NOMINAL
where type atom = Atm.atom
where type data = condition_term
NOM_ATOMIC_TERM_CONDITIONS
structure Sig : ATOMIC_CONDITIONS_SIGNATURE
structure Term: NOM_TERM_ALGEBRA
type condition_term
val app: Sig.predicate_symbol * Term.carrier list -> (string, predicate_term) Either.either
val dec: predicate_term -> Sig.predicate_symbol * Term.carrier list
include NOMINAL
where type data = predicate_term
where type atom = Trm.atom
CONDITIONS_SIGNATURE
type condition_symbol
val eq : condition_symbol * condition_symbol -> bool
val ar : condition_symbol -> int
NOM_TERM_ALGEBRA_UNIFICATION
structure Term: NOM_TERM_ALGEBRA
val unify: (Term.carrier * Term.carrier) list -> (string, (Term.carrier * Term.carrier) list) Either.either
STRONG_SYMBOLIC_TRANSITION_RESIDUAL
structure Tr : STRONG_SYMBOLIC_TRANSITION
datatype transformtype = BCAST | BREC | SYMM
type channeltransform = Trans of
Tr.Psi.Inst.name list *
Tr.Psi.Inst.assertion *
Tr.Psi.Inst.term *
Tr.Psi.Inst.term option
type suspended_constraint = unresolved * Tr.Constraint.constraint
type resolved = Tr.Act.action * suspended_constraint * Tr.Psi.psi
include NOMINAL
where type atom = Tr.Psi.Inst.name
where type data = resolved
sharing AtomSort = Tr.Psi.Inst.Sort
PSI_CLAUSE_ENVIRONMENT
structure Ct : PSI_CLAUSE
include ENVIRONMENT
where type key = Ct.Psi.Inst.term
where type value = Ct.Clause
BISIM
structure Psi: PSI
structure Tr : SYMBOLIC_BISIM_CONSTRAINT
val bisim :
Psi.psi * Psi.psi ->
(Constraint * ((Psi.psi * Psi.psi) * C.constraint) list)
sharing Psi = C.Psi
SYMBOLIC_BISIM_CONSTRAINT_SOLVER
structure Psi : PSI
structure Constraint : SYMBOLIC_BISIM_CONSTRAINT
val solve :
(string -> Constraint.constraint -> unit) ->
Constraint.constraint ->
(Psi.Inst.condition list list
, ((Psi.Inst.name * Psi.Inst.term) list * Psi.Inst.assertion) list)
Either.either
sharing Constraint.Psi = Psi
SYMBOLIC_BISIM_CONSTRAINT
structure Psi : PSI
datatype constraint
:: True
| False
| Atomic of Psi.Inst.name list * Psi.Inst.assertion * Psi.Inst.condition
| Transfer of Psi.Inst.term * Psi.Inst.term
| FrameP of Psi.Inst.name * Psi.psi
| Sdisjp of Psi.Inst.assertion * Psi.Inst.assertion
| Conj of constraint * constraint
| Disj of constraint * constraint
| Imp of constraint * constraint
include NOMINAL
where type atom = Psi.Inst.name
where type data = constraint
sharing AtomSort = Psi.Inst.Sort
val solve : constraint -> (Psi.Inst.name * Psi.Inst.term) list -> constraint
val simplify : constraint -> constraint
val cnf : constraint -> constraint
val dnf : constraint -> constraint
val resolveAssertions : (assertion option) list
-> constraint -> constraint
val simplifyEquationRules : (constraint -> constraint option) list
val deMorganAndNegRules : (constraint -> constraint option) list
val eliminateRules : (constraint -> constraint option) list
val allRules : (constraint -> constraint option) list
val disjunct : constraint list -> constraint
val conjunct : constraint list -> constraint
PSI_PARSER_REQ
structure Psi : PSI
val parseName : string -> (string, Psi.Inst.name) Either either
val parseTerm : string -> (string, Psi.Inst.term) Either either
val parseCond : string -> (string, Psi.Inst.condition) Either either
val parseAssr : string -> (string, Psi.Inst.assertion) Either either
val var : (Psi.Inst.name -> Psi.Inst.term) option
ENVIRONMENT
type env
type key
type value
val extend : env -> key -> value -> env
val remove : env -> key -> env
val find : env -> key -> value list
val merge : env -> env -> env list
val empty : env
STRONG_SYMBOLIC_TRANSITION
structure Psi : PSI
structure Act : PSI_STRONG_ACTION
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
type transition
val tau : assertion -> 'a
* Psi.psi * Act.action * Constraint.constraint * Psi.psi
sharing Psi.Inst = Constraint.Inst = Act.Inst
PSI_CLAUSE
structure Psi: PSI
type clause =
Psi.Inst.term * Psi.Inst.name list * Psi.psi
include NOMINAL
where type atom = Psi.Inst.name
where type data = clause
FRAME
structure Psi: PSI
type frame
val f : Psi.psi -> frame
val unit_frame : frame
val compose : frame -> frame -> frame
val assertion : frame -> Psi.Inst.assertion
val binders : frame -> Psi.Inst.name list
include NOMINAL
sharing AtomSort = Psi.AtomSort
sharing type atom = Psi.Inst.name
sharing type data = frame
PSI_PRETTY_PRINTER
structure PP : PRETTY_PRINTER
structure Psi : PSI
val e : Psi.Inst.name -> PP.document
val a : Psi.Inst.assertion -> PP.document
val c : Psi.Inst.condition -> PP.document
val ppAgent : Psi.psi -> PP.document
ATOMIC_CONDITIONS_SIGNATURE
type predicate_symbol
val eq : predicate_symbol * predicate_symbol -> bool
val ar : predicate_symbol -> int
NOM_TERM_ALGEBRA
val var : atom -> carrier
:: Zero
| Suc of carrier
| Sigma function_symbol * carrier list
val eq : carrier -> carrier -> bool
val freeAtm : carrier -> atom * carrier
val isZero : carrier -> bool
val isSuc : carrier -> bool
val isSuc : (carrier * carrier) list -> carrier -> carrier
AST
datatype ast :: Node of string * ast list
structure Parser : PARSER
val parse : string -> ast Parser.parser
PARSER
structure Stream : PARSER_STREAM
datatype 'a result = Fail of 'a | Error of 'a
type 'result parser
val parse : 'result parser -> Stream.stream ->
((string, fail) * Stream.stream) list,
('result * Stream.stream) list) Either.either
val direct_error_parser :
(string fail * Stream.stream) list,
('result * Stream.stream) list) Either.either) ->
'result parser
val return : 'a -> 'a parser
val bind : 'a parser -> ('a -> 'b parser) -> 'b parser
val >>= : 'a parser * ('a -> 'b parser) -> 'b parser
val delayed : (unit -> 'a parser) -> 'a parser
val error : string -> 'result parser
val zero : 'result parser
val choice : 'a parser * 'a parser -> 'a parser
val choice' : 'a parser * 'a parser -> 'a parser
val anf : 'a -> 'a parser
val dFns : (int * int * int) parser
val item : char parser
val sat : (char -> bool) -> char parser
val str : string -> string parser
val str : string -> string parser
val many : 'a parser -> ('a list) parser
val many1 : 'a parser -> ('a list) parser
val sepby : 'a parser * 'b parser -> 'a list parser
val sepby1 : 'a parser * 'b parser -> 'a list parser
val rightAssocOp : 'a parser * ('a * 'a -> 'a) parser -> 'a parser
val leftAssocOp : 'a parser -> ('a * 'a -> 'a) parser -> 'a -> 'a parser
val rightAssocOp1 : 'a parser * ('a * 'a -> 'a) parser -> 'a parser
val rightAssocOp : 'a parser -> ('a * 'a -> 'a) parser -> 'a -> 'a parser
val until : 'a parser -> char list parser
val between : 'a parser -> 'b parser -> 'b parser
val try : 'a parser -> 'a parser
val nothal : 'a parser -> 'a parser
val both : 'a parser -> 'b parser -> ('a * 'b) parser
PARSER_STREAM
type stream
val hasI : stream -> (char * stream) option
val isReady : stream -> bool
val isEnd : stream -> bool
val pos : stream -> int * int * int
type base_stream
val make : base_stream -> stream
SYMBOLIC_CONSTRAINT_SOLVER
structure Inst : PSI_INSTANCE
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
val solve :
Constraint.constraint ->
(Inst.condition list list
, ((Inst.name * Inst.term) list * Inst.assertion) list
) Either.either
sharing Constraint.Inst = Inst
PSI_PARSER
structure Psi : PSI
structure Parser : PARSER
val eqsint : string -> Psi.psi Parser.parser
val eqsint' : string -> Psi.psi Parser.parser
val name : Psi.Inst.name Parser.parser
val t : string -> string Parser.parser
val ws : char list Parser.parser
type weak_transition
= Psi.psi * weak_action * Constraint.constraint * Psi.psi
WEAK_SYMBOLIC_TRANSITION
structure Psi : PSI
structure Constraint : SYMBOLIC_TRANSITION_CONSTRAINT
sharing Psi.Inst = Constraint.Inst
datatype weak_action
= Silent
| Input of Psi.Inst.name * Psi.Inst.name list * Psi.Inst.term
| Output of Psi.Inst.name * Psi.Inst.name list * Psi.Inst.term
| BrInput of Psi.Inst.name * Psi.Inst.name list * Psi.Inst.term
| BrOutput of Psi.Inst.name * Psi.Inst.name list * Psi.Inst.term
| BrInput' of Psi.Inst.name * Psi.Inst.name
type weak_transition
= Psi.psi * weak_action * Constraint.constraint * Psi.psi
SYMBOLIC_TRANSITION_CONSTRAINT
structure Inst : PSI_INSTANCE
datatype constraint = Atomic of Inst.assertion * Inst.condition
| Conjunction of constraint list
| Exists of Inst.name * Inst.name * constraint
| Conclusion of Inst.name list * constraint
include NOMINAL
where type data = constraint
where type atom = Inst.name
sharing AtomSort = Inst.Sort
val solve : constraint -> (Inst.name * Inst.term) list -> constraint
val restrict: constraint -> Inst.name list -> constraint
val compose: constraint -> constraint -> constraint
val freshen: constraint -> Inst.name list -> constraint
PSI
structure Inst: PSI_INSTANCE
datatype psi
= Nil
| Output of Inst.term * Inst.term * psi
| BrOutput of Inst.term * Inst.term * psi
| Input of Inst.term * (Inst.name list) * Inst.term * psi
| BrInput of Inst.term * (Inst.name list) * Inst.term * psi
| Case of (Inst.condition * psi) list
| Par of psi list * psi
| Parallel of psi list
| Replication of psi
| Assertion of Inst.assertion
| Invocation of Inst.term * (Inst.term list)
include NOMINAL
where type atom = Inst.name
where type data = psi
sharing AtomSort = Inst.Sort
val subst : psi -> (Inst.name * Inst.term) list -> psi
PSI_PP_REQ
structure Inst : PSI_INSTANCE
val printN : Inst.name -> string
val printT : Inst.term -> string
val printC : Inst.condition -> string
val printA : Inst.assertion -> string
val var : (Inst.name -> Inst.term) option
PSI_STRONG_ACTION
structure Inst : PSI_INSTANCE
datatype action
= Tau
| Output of Inst.name * Inst.name list * Inst.term
| Input of Inst.name * Inst.name
| BrOutput of Inst.name * Inst.name list * Inst.term
| BrInput of Inst.name * Inst.name
val silly : action -> Inst.term
val obj : action -> Inst.term
val bn : action -> Inst.name list
include NOMINAL
where type atom = Inst.name
where type data = action
sharing AtomSort = Inst.Sort
PRETTY_PRINTER
type document
val empty : document
val join : document * document -> document
val nest : int -> document -> document
val text : string -> document
val line : document
val group : document -> document
val fill : document list -> document
val pretty : int -> document -> string
NOM_SIGMA_ALGEBRA
include SIGMA_ALGEBRA
include NOMINAL
sharing type carrier = atom
sharing Sigma.Term = DataSort = AtomSort
NOM_ABS_SIGMA
include NOM_SIGMA
val absSort : Sort.sort -> Sort.sort ->
(string,Sort.sort) Either.either
val isAbsSort : Sort.sort -> bool
SIGMA_ALGEBRA
type carrier
structure Sigma: SIGMA
include
val app: Sigma.function_symbol * carrier list
-> (string, carrier) Either.either
NOM_SIGMA
include SIGMA
val isAtomSort : Sort.sort -> bool
NOMINAL
eqtype atom
type data
structure AtomSort: SORT
structure DataSort: SORT
val atomSort : atom -> AtomSort.sort
val dataSort : data -> DataSort.sort
val new : atom -> atom list -> atom
val sw swAtom : atom * atom -> atom -> atom
val sw sw : atom * atom -> data -> data
val sw swData : atom * atom -> data -> data
val supports : data -> atom list
val eq : (atom * atom) -> bool
val eqData : data * data -> bool
NOMINAL_MINIMAL
eqtype atom
type data
structure AtomSort: SORT
structure DataSort: SORT
val atomSort : atom -> AtomSort.sort
val dataSort : data -> DataSort.sort
val sw sp : atom * atom -> data -> data
val supports : data -> atom list
val new : atom -> atom list -> atom
val eq : (((atom * data) * (atom * data)) -> bool) ->
data * data -> bool
UNSORT
include SORT
val u : sort
SUBSET_SORT
structure Sort : SORT
type sort
val inc : sort -> Sort.sort
NOMINAL_NAMES
eqtype atom
structure AtomSort : SORT
val atomSort : atom -> AtomSort.sort
val new : atom -> atom list -> atom
val sw swAtom : atom * atom -> atom -> atom
SORT
eqtype sort
val eq: sort * sort -> bool