

Contents

1	Module Constraint : Constraint	1
2	Module Ast : File : ast.ml Last updated : 4.14.09	2
3	Module AstUtils	5
4	Module ConstraintUtils	6
5	Module Parser	6
6	Module Evalenv : Module : Evalenv	7
7	Module Yparser	9
8	Module Oracle	10
9	Module Evaluator : Evaluator for LogLog programs.	10

1 Module Constraint : Constraint

```
module Constraint :
  sig
    type numconstraint =
      | CEq
      | CNeq
      | CLt
      | CGt
      | CLte
      | CGte

    type varname = string
    type recordindex = string * string * int
    type constraintbinop =
      | CPlus
      | CMinus
      | CTimes
      | CDivide

    type cvar =
      | CVar of varname
      | CRecordVar of recordindex

    type cexp =
      | CVarExp of cvar
      | CInt of int
```

```

    | CArithop of constraintbinop * cexp
      * cexp
    | CBoolop of numconstraint * cexp
      * cexp
    | CGuard of cexp * cexp
      * cexp
    | CAnd of cexp list
    | COr of cexp list
    | CTrue
    | CFalse
val negate : cexp -> cexp
val mkEqs : cexp ->
  cexp -> cexp
val mkVar : string -> cexp
val mkRVar : string -> string -> int -> cexp
end

```

2 Module Ast : File : ast.ml Last updated : 4.14.09

AST declarations for the LogLog language.

```

module Ast :
  sig
    type typerep =
      | IntType
      | FloatType
      | StringType
      | BoolType
      | RelationType of string
      | Unknown

      Type representations.

    type binop =
      | Plus
      | Minus
      | Times
      | Divide

      LogLog basics.

    type boolop =
      | Equiv
      | Nequiv

```

```

| Or
| And
| Lt
| Lte
| Gt
| Gte
type primexp =
| Float of float
| Integer of int
| String of string
| Bool of bool
type exp =
| PrimExp of primexp
| RelationExp of string * relation
| VarExp of var
| FunAppExp of string list
| Binop of binop * exp * exp
| Boolop of boolop * exp * exp
| SymExp of symexp

```

Symbolic expression.

Expressions.

```

type symexp =
| SymVar of var

```

Symbolic variables—this must come from reading input.

```

| SymExpr of exp

```

Symbolic expression contains at least one reference to a symbolic variable somewhere. This exists in the form of a variable expression (Ast.VarExp).

```

| SymPred of exp * exp * exp

```

Has a symbolic conditional.

Symbolic expressions.

Constraints.

```

type constraintexp =
| ACBoolop of boolop * constraintexp * constraintexp
| ACThis

```

"this"

```

| ACVar of string
| ACPprev of string
| ACNext of string
| ACDrop of string

```

Special "drop" keyword.

Constraint expression.

```
type constraintvalue =  
  | ConstraintForall of string * string list * constraintvalue  
  | ConstraintExists of string * string list * constraintvalue  
  | CValExp of constraintexp  
type constraintbody =  
  | ConstraintAssume of constraintvalue list  
  | ConstraintAssert of constraintvalue list  
type var =  
  | Var of string  
  | RecordIndexVar of string * string * int  
  | IndexVar of string * string
```

Data gets stored with its constraints

Data fields for a record (i.e. one line of a file).

```
type datafields = exp MapUtils.StringMap.t  
type relation = datafields MapUtils.IntMap.t  
val empty_relation : relation  
type stmt =  
  | AssignStmt of var * exp  
  | Cond of exp * stmt list * stmt list  
  | ForEachLoop of string * string * stmt list  
  | Return of exp  
  | Assert of exp  
  | Assume of exp
```

Statements show up in procedure bodies.

```
type userty = {  
  dataname : string ;  
  fieldinfo : (string * typerep) list ;  
}
```

User-defined types can only be relation/record types.

```
type fundecl = {  
  funname : string ;  
  funargs : (string * typerep) list ;  
  funbody : stmt list ;  
}  
type datarhs =  
  | RhsExp of exp  
  | Concretize of exp  
  | CRange of exp
```

Things that can appear on the right-hand side of a top-level data * declaration.

```
type topdecl =  
  | TypeDecl of userty * constraintbody list  
      Type declaration contains description of record type with additional *  
      assumptions/assertions.  
  | FunDecl of fundekl  
      Procedure declaration.  
  | AssignDecl of string * typerep * datarhs  
      Top-level declarations.  
  
end
```

3 Module AstUtils

```
module AstEq :  
  sig  
    val eq_types : Ast.Ast.typerep -> Ast.Ast.typerep -> bool  
    val eq_vars : Ast.Ast.var -> Ast.Ast.var -> int  
    val eq_exps : Ast.Ast.exp -> Ast.Ast.exp -> bool  
  end  
  
module Var :  
  sig  
    type t = Ast.Ast.var  
    val compare : Ast.Ast.var -> Ast.Ast.var -> int  
    val v_str : Ast.Ast.var -> string  
    val str2var : string -> Ast.Ast.var  
  end  
  
module VarMap :  
  Map.S with type key = Var.t  
  
module SymUtils :  
  sig  
    val sym : Ast.Ast.exp -> bool  
    val get_constraint_vars :  
      string list -> Constraint.Constraint.cexp -> string list  
    val get_sym_vars : Ast.Ast.exp -> Ast.Ast.var list
```

```

end

module RelationUtils :
sig
    val set_field :
        string -> Ast.Ast.exp -> Ast.Ast.datafields -> Ast.Ast.datafields
    val get_field : Ast.Ast.datafields -> string -> Ast.Ast.exp
    val set_index_field :
        string -> int -> Ast.Ast.exp -> Ast.Ast.relation -> Ast.Ast.relation
    val get_index_field : string -> int -> Ast.Ast.relation -> Ast.Ast.exp option
    val relation_fold :
        (string -> int -> Ast.Ast.exp -> 'a -> 'a) -> 'a -> Ast.Ast.relation -> 'a
end

module AstPrint :
sig
    val pickle_var : Ast.Ast.var -> string
    val pickle_exp : Ast.Ast.exp -> string
end

```

4 Module ConstraintUtils

```

module ConstraintSimplify :
sig
    val csimplify : Constraint.Constraint.cexp -> Constraint.Constraint.cexp
end

module ConstraintConverter :
sig
    val var2constraint : Ast.Ast.var -> Constraint.Constraint.cexp
    val ast2constraint : Ast.Ast.exp -> Constraint.Constraint.cexp
    val convert_constraints : Constraint.Constraint.cexp list -> string list
end

module ConstraintPrinter :
sig
    val print_cexps : Constraint.Constraint.cexp list -> unit list
end

```

5 Module Parser

```
type token =  
  | NEWLINE  
  | COMMENT  
  | EOF  
  | LPAREN  
  | RPAREN  
  | LBRACE  
  | RBRACE  
  | LBRACKET  
  | RBRACKET  
  | SEMI  
  | PERIOD  
  | COMMA  
  | DOUBLECOLON  
  | EQUALS  
  | NEQUALS  
  | DEQUALS  
  | IMPLIES  
  | ASSERT  
  | ASSUME  
  | ASSERTF  
  | CONCRETIZE  
  | CRANGE  
  | RETURN  
  | INTTY  
  | FLOATTY  
  | STRINGTY  
  | LT  
  | LTE  
  | GT  
  | GTE  
  | AND  
  | OR  
  | PLUS  
  | MINUS  
  | MULTIPLY  
  | DIVIDE  
  | NEG  
  | CARET  
  | LET  
  | RELATION  
  | RECORD
```

```

| FOREACH
| IN
| IF
| ELSE
| DATA
| FUN
| WITH
| DROP
| FLOAT of float
| INTEGER of int
| IDENTIFIER of string
val input : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Ast.Ast.topdecl list

```

6 Module Evalenv : Module : Evalenv

```

module Evalenv :
sig
  type constraintenv = Constraint.Constraint.cexp Pervasives.ref list AstUtils.VarMap.t
    For storing information about data values/user-defined types along with their
    associated constraints.

  type typeconstraint = string ->
    Ast.Ast.relation ->
    constraintenv -> constraintenv

  type evalenv = {
    typeconstraints : typeconstraint MapUtils.StringMap.t ;
    tyenv : Ast.Ast.typerep MapUtils.StringMap.t ;
    tydefenv : Ast.Ast.userty MapUtils.StringMap.t ;
    functions : Ast.Ast.fundekl MapUtils.StringMap.t ;
    globalbindings : Ast.Ast.exp AstUtils.VarMap.t ;
    localbindings : Ast.Ast.exp AstUtils.VarMap.t ;
    constraintbindings : constraintenv ;
  }

  val default_tyconstraint :
    string ->
    Ast.Ast.relation ->
    constraintenv -> constraintenv
    Default type constraint.
    Default empty environment.

  val empty_evalenv : evalenv
  Type names  $\rightarrow$  type constraints.

```



```

val addFunDecl : evalenv -> Ast.Ast.fundecl -> evalenv
    Function declarations.

val lookupFunDecl : evalenv -> string -> Ast.Ast.fundecl option
val bindTypeGlobalEnv : evalenv ->
    string -> Ast.Ast.typerep -> evalenv
    Var names  $\rightarrow$  type representations.

val lookupType : evalenv -> Ast.Ast.var -> Ast.Ast.typerep option
val sameType : evalenv -> Ast.Ast.var -> Ast.Ast.var -> bool
val addTypeDecl :
    evalenv ->
    Ast.Ast.userty -> Ast.Ast.constraintbody list -> evalenv
    Type names  $\rightarrow$  constraints.

val lookupTypeDeclByData : evalenv -> string -> Ast.Ast.userty option
val addTypeConstraints : evalenv -> string -> Ast.Ast.exp -> evalenv
    Derives type-level constraints for variable name passed as arg.

val lookupValueEnv : evalenv -> Ast.Ast.var -> Ast.Ast.exp option
    Var names  $\rightarrow$  expressions

val bindValueGlobalEnv : evalenv ->
    Ast.Ast.var -> Ast.Ast.exp -> evalenv
val bindValueLocalEnv : evalenv ->
    Ast.Ast.var -> Ast.Ast.exp -> evalenv
val lookupRelationGlobalEnv :
    evalenv -> Ast.Ast.var -> Ast.Ast.relation option
val addConstraintGlobalEnv :
    evalenv ->
    Ast.Ast.var ->
    Constraint.Constraint.cexp Pervasives.ref -> evalenv
    Var names  $\rightarrow$  constraints.

val addConstraintListGlobalEnv :
    evalenv ->
    Ast.Ast.var ->
    Constraint.Constraint.cexp Pervasives.ref list -> evalenv
val lookupConstraintsEnv :
    evalenv ->
    Ast.Ast.var -> Constraint.Constraint.cexp Pervasives.ref list
end

```

Evaluation environment.

7 Module Yparser

```
type token =
  | SAT
  | UNSAT
  | NEWLINE
  | COMMENT
  | EOF
  | LPAREN
  | RPAREN
  | EQUALS
  | NEQUALS
  | LT
  | LTE
  | GT
  | GTE
  | AND
  | OR
  | PLUS
  | MINUS
  | MULTIPLY
  | DIVIDE
  | NEG
  | CARET
  | INTEGER of int
  | IDENTIFIER of string
val input : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Yices.yexp list
```

8 Module Oracle

```
module Oracle :
  sig
    val concretize :
      Evalenv.Evalenv.evalenv ->
      Ast.Ast.exp -> Evalenv.Evalenv.evalenv * Ast.Ast.exp
  end
```

9 Module Evaluator : Evaluator for LogLog programs.

```
module Evaluator :
```

```
sig
  val evaluate : Ast.Ast.topdecl list -> Evalenv.Evalenv.evalenv
end

Evaluation module.
```