ArgoExpression: SMT-LIB 2.0 compliant expression library

Milan Banković Filip Marić
{milan,filip}@matf.bg.ac.rs

Department of Computer Science
Faculty of Mathematics
University of Belgrade

4th Workshop on Formal and Automated Theorem Proving and Applications. Belgrade 2011.

Outline

Outline

Introduction to SMT-LIB 2.0

SMT-LIB is an international effort in:

- providing a standard language for rigorous descriptions of first-order theories used in SMT
- providing a standard language for SMT solvers' input and output (including the language of expressions)
- providing a large library of benchmarks for testing SMT solvers

History and Credits

- Developed since 2003.
- Version 1.1 (2005)
- Version 1.2 (2006) adopted by most SMT solvers
- Version 2.0 (2010) still in active development
- Joint work of three work groups led by Cesare Tinelli, Clark Barrett, and Aaron Stump

Motivation

One of the first issues in developing an SMT solver is...

to develop a program library for dealing with first-order expressions that follows SMT-LIB standard.

Our goal is...

- to provide an object-oriented program library that is fully compliant with SMT-LIB 2.0 standard, easy to use, efficient and robust.
- to make the library open and available for other researchers and developers (free software / open source)
- to involve other researchers in developing the library and perhaps to make the library standard implementation of SMT-LIB 2.0, adopted by other open source solvers

Outline

Description of SMT-LIB 2.0

SMT-LIB 2.0 features:

- many-sorted (many-typed) first-order logic as its underlying logic
- sorts (types) – simple sorts, sort terms, parametric sorts...
- expressions (terms) – well sortedness!!
- formulae – just regular expressions of sort Bool
- theory declarations – signature specification (sort and function symbols)
- logic declarations – combined signatures, theory combination
- scripts – text-based interface to an SMT solver

Example of SMT-LIB 2.0 theory declaration(1)

```
(theory Core

:smt-lib-version 2.0
:written_by "Cesare Tinelli"
:date "2010-04-17"
:last_modified "2010-08-15"

:sorts ((Bool 0))

:funs ((true Bool)
       (false Bool)
       (not Bool Bool)
       (=> Bool Bool Bool :right-assoc)
       (and Bool Bool Bool :left-assoc)
       (or Bool Bool Bool :left-assoc)
       (xor Bool Bool Bool :left-assoc)
       (par (A) (= A A Bool :chainable))
       (par (A) (distinct A A Bool :pairwise))
       (par (A) (ite Bool A A A))
      )

:definition
"<some free text descripton>"
...
)
```

Example of SMT-LIB 2.0 theory declaration(2)

```
(theory Ints

:smt-lib-version 2.0
:written_by "Cesare Tinelli"
:date "2010-04-17"

:sorts ((Int 0))

:funs ((NUMERAL Int)
       (- Int Int)                  ; negation
       (- Int Int Int :left-assoc) ; subtraction
       (+ Int Int Int :left-assoc)
       (* Int Int Int :left-assoc)
       (div Int Int Int :left-assoc)
       (mod Int Int Int)
       (abs Int Int)
       (<= Int Int Bool :chainable)
       (<  Int Int Bool :chainable)
       (>= Int Int Bool :chainable)
       (>  Int Int Bool :chainable)
      )

:definition
"<some free text description>"
...
)
```

Example of SMT-LIB 2.0 theory declaration(3)

```
(theory ArraysEx

 :smt-lib-version 2.0
 :written_by "Cesare Tinelli"
 :date "2010-04-28"
 :last_modified "2010-08-15"

 :sorts ((Array 2))

 :funs ((par (X Y) (select (Array X Y) X Y))
        (par (X Y) (store (Array X Y) X Y (Array X Y))) )

 :definition
 "<some free text description>"
)
```

Example of SMT-LIB 2.0 expressions

```
;Core theory expression:

( or ( and p q ) r )

;Ints theory expressions:

( + a b )
( = ( + a b ) ( + b a ) )
( => ( and ( <= a b ) ( <= b a) ) ( =  a b ) )
( forall ( X Int ) ( exists ( Y Int ) ( = ( + X 1 ) Y ) ) )

;ArraysEx theory expression:

( forall ( ( a ( Array s1 s2 ) ) ( i s1 ) ( e s2 ) )
      ( = ( select ( store a i e ) i ) e ) )
```

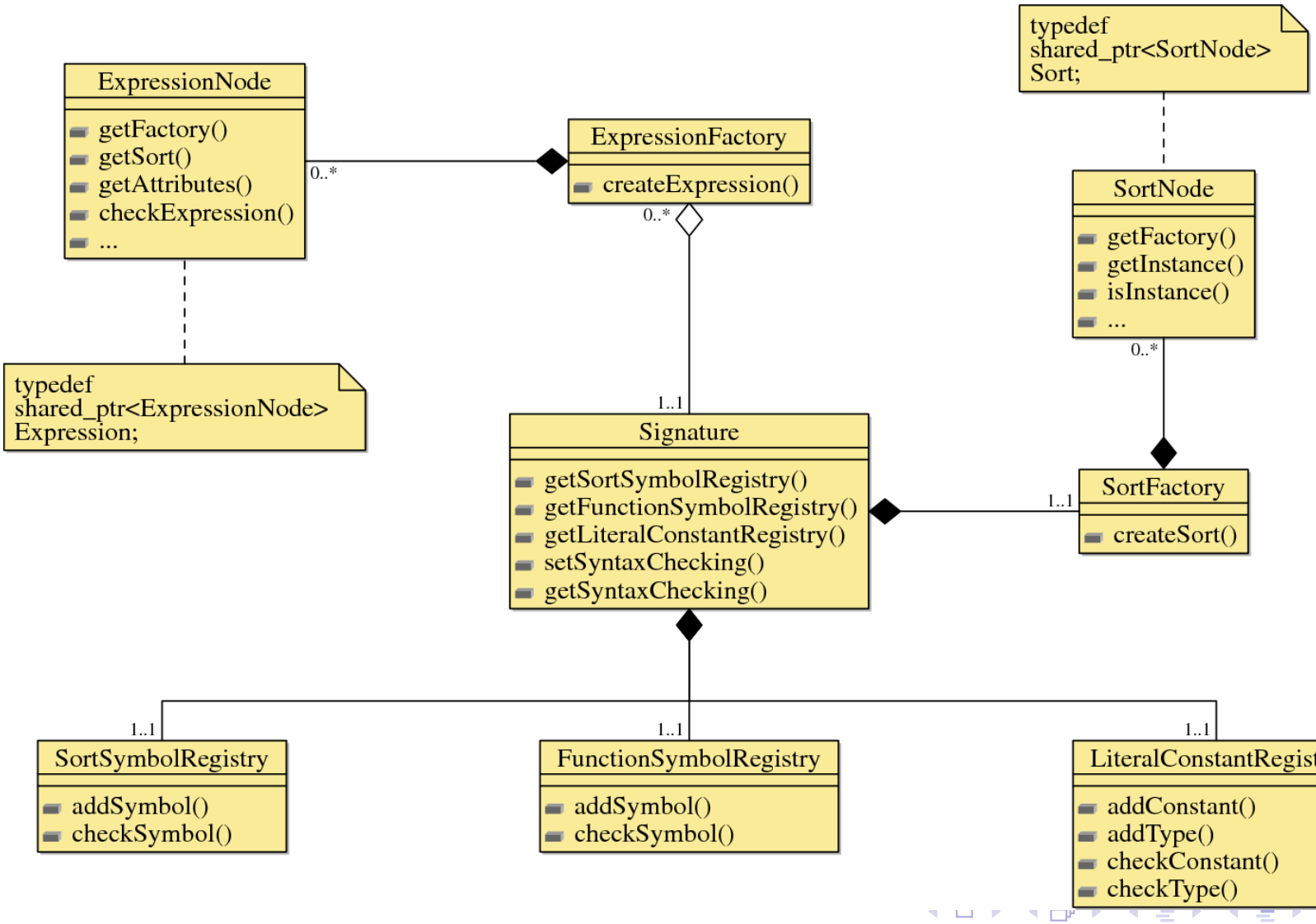
Outline

ArgoExpression description

ArgoExpression features:

- developed in standard C++ (GNU/Linux, g++)
- developed using well-known design patterns (Factory, FlyWeight, Composite, Visitor, Iterator...)
- expressions implemented using common subexpression sharing technique to minimize memory requirements (also known as hash consing). The same technique is used for sort terms.
- C++ TR1 shared pointers (shared_ptr<>) used for subexpression sharing and garbage collecting (FlyWeight pattern)
- Supports strict syntax checking and well sortedness verification during expression construction. Supports syntax checking against arbitrary signature.
- Supports combined signatures (for theory combinations).

Overall structure of ArgoExpression

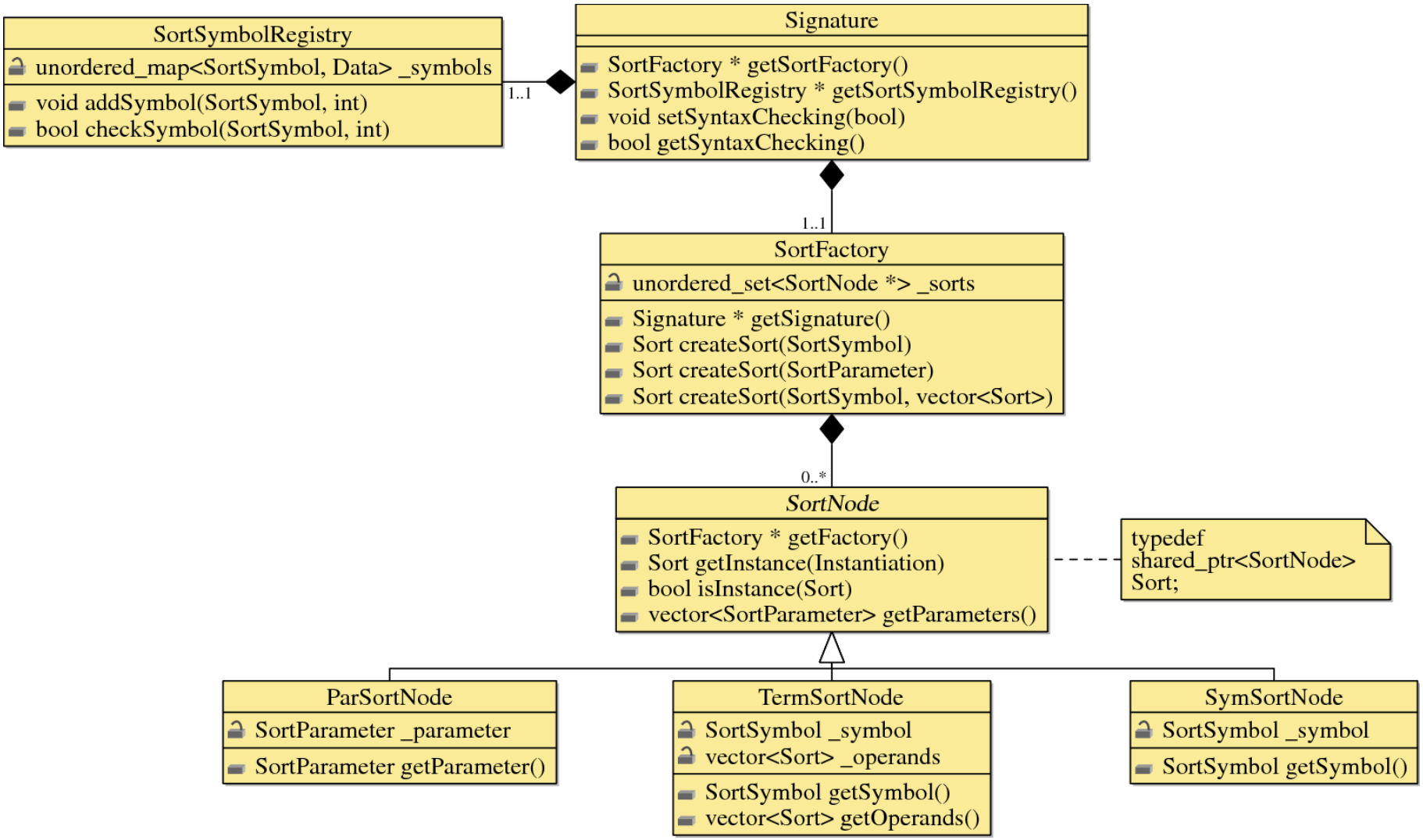


Sorts in SMT-LIB 2.0

Features:

- A sort is a term built over a finite set of sort symbols with specified arity.
- Sort symbols are part of the signature, and are declared in the corresponding SMT theory declaration (for instance (Bool 0), (Int 0), (List 1), (Array 2), ...)
- Simple sorts (sort symbols with arity 0): Bool, Int
- Sort parameter (sort variable): U, V...
- Sort terms: (List Bool), (Array Int (List Bool)), (List (Array Int Int))...
- Parametric sorts: (List U), where U is a sort parameter.
- Parameters in parametric sorts can be instantiated with an arbitrary sort.

Sorts subsystem of ArgoExpression



Sorts in ArgoExpression

```
Signature * my_sig = new Signature();
my_sig->getSortSymbolRegistry()->addSymbol(SortSymbol("Bool"), 0);
my_sig->getSortSymbolRegistry()->addSymbol(SortSymbol("Int"), 0);
my_sig->getSortSymbolRegistry()->addSymbol(SortSymbol("List"), 1);
my_sig->getSortSymbolRegistry()->addSymbol(SortSymbol("Array"), 2);

Sort boolSort = my_sig->getSortFactory()->createSort(SortSymbol("Bool"));
Sort intSort = my_sig->getSortFactory()->createSort(SortSymbol("Int"));

// (List Int)
Sort listOfInts = my_sig->getSortFactory()->createSort(SortSymbol("List"), intSort);

// (Array Int (List Int))
Sort arrayIntList = my_sig->getSortFactory()->
                            createSort(SortSymbol("Array"), intSort, listOfInts);

Sort uPar = my_sig->getSortFactory()->createSort(SortParameter("U"));
Sort vPar = my_sig->getSortFactory()->createSort(SortParameter("V"));
Sort arrayUV = my_sig->getSortFactory()->
                        createSort(SortSymbol("Array"), uPar, vPar); // (Array U V)

Instantiation ins;
ins.insert(make_pair(SortParameter("U"), intSort));
ins.insert(make_pair(SortParameter("V"), boolSort));
Sort arrayIntBool = arrayUV->getInstance(ins); // (Array Int Bool)
```

Expressions (terms) in SMT-LIB 2.0

Features:

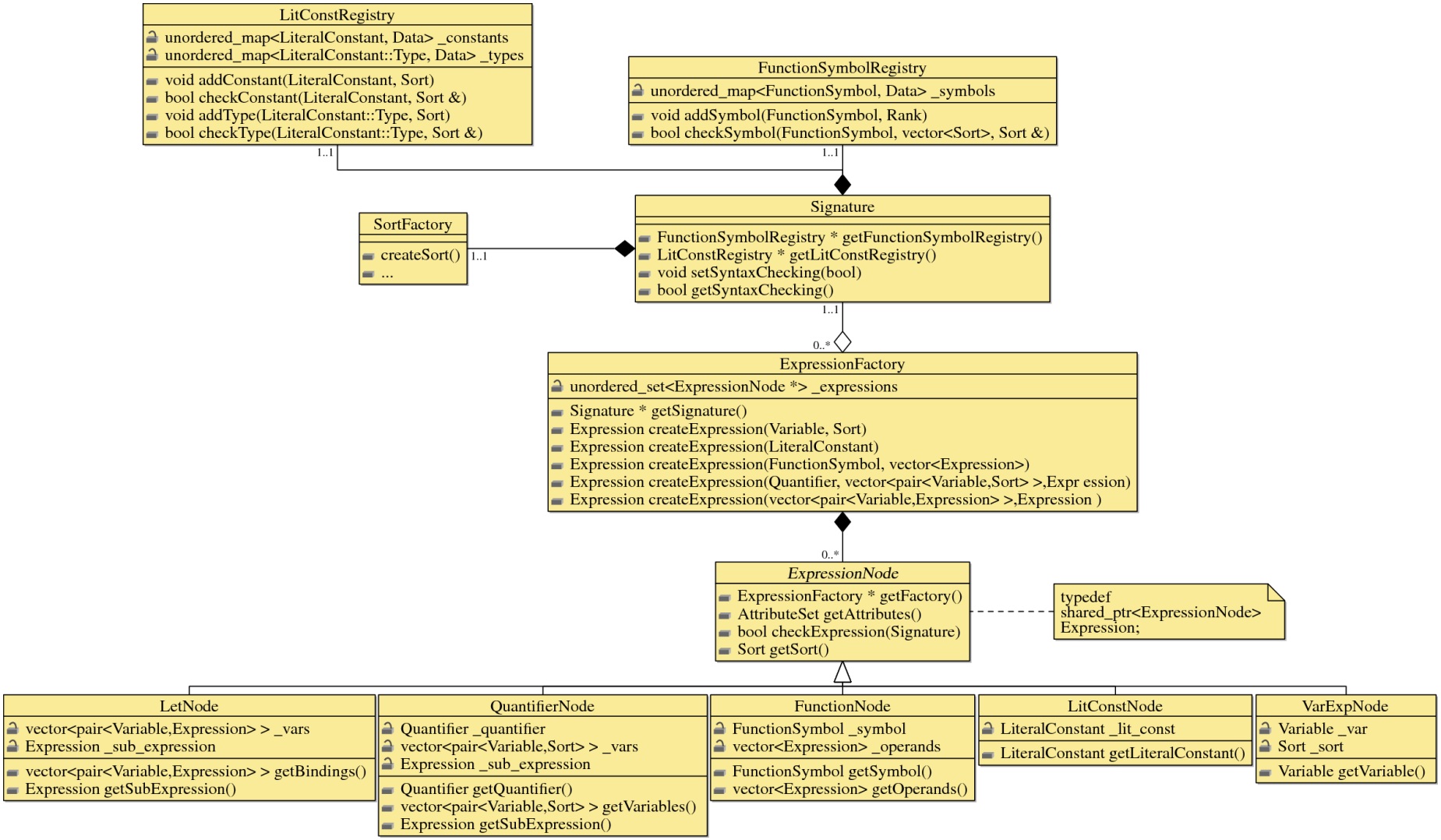
- An expression are built over variables, literal constants and a finite set of function symbols with specified ranks.
- A rank is a sequence $\sigma_1, \sigma_2, \ldots, \sigma_n, \sigma$, where σ_i is the sort of i-th argument, and σ is the return sort.
- Function symbols are part of the signature, and are declared in the corresponding SMT theory declaration: (not Bool Bool), (and Bool Bool Bool :left-assoc)...
- A function symbol a with the rank σ is called constant symbol of sort σ: (true Bool), (false Bool)
- A function symbol can have more than one ranks (overloaded function symbols): (- Int Int), (- Int Int Int :left-assoc)...
- A function symbol can be parametric:
  (par (U) (= U U Bool :chainable))

Expressions (terms) in SMT-LIB 2.0

Features:

- Literal constants of different types:
  - NUMERAL (0, 1, 2...)
  - BINARY (#b0, #b010...)
  - HEXADECIMAL (#xFF, #a2c...)
  - DECIMAL (0.25, 3.042..)
  - STRING ("string constant")
- Their sorts are specified in the signature: (#b0 Bool), (#b1 Bool), (NUMERAL Int)...
- Variables and quantifiers: (forall ((X Int) (Y Int)) (p X Y))
- Quantifiers are always applied to expressions of sort Bool (called formulae).
- Let binders: (let ((X (+ a 1)) (Y (- a 1))) (p X Y))

Expression subsystem of ArgoExpression



Expressions in ArgoExpression

```
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("true"),boolSort);
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("false"),boolSort);
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("not"),
                                               boolSort,boolSort);
AttributeSet attr;
attr.insert(Keyword(":left-assoc"));
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("and"),boolSort,
                                                    boolSort,boolSort,attr);
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("or"),boolSort,
                                                    boolSort,boolSort,attr);
attr.clear();
attr.insert(Keyword(":chainable"));
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("="),uPar,uPar,
                                                         boolSort,attr);
ExpressionFactory * factory = new ExpressionFactory(my_sig);
Expression topExp = factory->createExpression(FunctionSymbol("true"));
Expression botExp = factory->createExpression(FunctionSymbol("false"));
Expression varP = factory->createExpression(Variable("P"), boolSort);
Expression varQ = factory->createExpression(Variable("Q"), boolSort);
// (and P Q)
Expression andPQ = factory->createExpression(FunctionSymbol("and"), varP, varQ);
// (= (and P Q) false)
Expression andPQeqFalse = factory->createExpression(FunctionSymbol("="),
                                                    andPQ, botExp);
```

Expressions in ArgoExpression

```
attr.clear();
attr.insert(Keyword(":left-assoc"));
my_sig->getFunctionSymbolRegistry()->addSymbol(FunctionSymbol("+"),
                                    intSort,intSort,intSort,attr);
my_sig->getLiteralConstantRegistry()->addType(T_NUMERAL, intSort);
Expression one = factory->createExpression(LiteralConstant(T_NUMERAL, "1"));
Expression varX = factory->createExpression(Variable("X"), intSort);
// (+ X 1)
Expression x_plus_one = factory->createExpression(FunctionSymbol("+"),
                                                  varX, one);
Expression varY = factory->createExpression(Variable("Y"), intSort);
// (= (+ X 1) Y)
Expression x_plus_one_eq_y = factory->createExpression(FunctionSymbol("="),
                                                       x_plus_one, varY);
// (exists (Y Int) (= (+ X 1) Y))
vector<pair<Variable,Sort> > vars;
vars.push_back(make_pair(Variable("Y"), intSort));
Expression exists = factory->createExpression(Q_EXISTS, vars,
                                              x_plus_one_eq_y);
// (forall (X Int) (exists (Y Int) (= (+ X 1) Y)))
vars.clear();
vars.push_back(make_pair(Variable("X"), intSort));
Expression forall = factory->createExpression(Q_FORALL, vars, exists);
```

Combined signatures support

Features:

- Signatures are combined into one composite signature, for sake of theory combination.
- Signatures that should be combined are given in SMT logic declaration: for instance :theories (Ints ArraysEx)
- Signature combining is implemented using Composite design pattern.
- Supports checking expression well-sortedness against particular subsignature (needed for purification)
- All signature in composition share common sort factory.

Subsignatures in ArgoExpression

```
Signature * my_sig = new Signature(); // composite signature
Signature *core_sig = new Signature(my_sig); // subsignature of my_sig
Signature *ints_sig = new Signature(my_sig); // subsignature of my_sig

// Populate signatures with their sort and function symbols and
// literal constants
core_sig->getSortSymbolRegistry()->addSymbol(...);
...
ints_sig->getSortSymbolRegistry()->addSymbol(...);

// Create expression factory over my_sig
ExpressionFactory * factory = new ExpressionFactory(my_sig);

// Create expressions:
Expression exp = factory->createExpression(...);

// Check expression against core_sig:
if(exp->checkExpression(core_sig))
 cout << "Boolean expression!" << endl;
```

Outline

Future work

TODO list:

- Parser for expressions and for theories and logics declarations
- Scripts support (interface to solvers)
- Intensive testing and debugging
- Profiling and efficiency improvements

THANK YOU :)