Documentation of GIDL's source code

Johan Wittocx

November 21, 2011

1 Files

vocabulary.{h,cc} Sorts (types), predicate and function symbols, domain elements

structure.{h,cc} Interpretations (tables) for sorts, predicates and functions

term.{h,cc} Terms and set expressions

theory.{h,cc} Formulas, definitions, fixpoint definitions and theories

namespace.{h,cc} Namespaces

builtin.{h,cc} Built-in sorts, predicates and functions

execute.{h,cc} Inference methods

visitor.{h,cc} Visitor patterns

options.h (Command-line) options

error.{h,cc} Error and warning messages

lex.ll, parse.yy, insert.{h,cc} Parsing

main.cc Initialize data, read command-line arguments, parse input files, delete data

common.cc Some common methods (convert string to int, maximum integer, ...)

ground.{h,cc} (Naive) grounder

clconst.h Some classes for parsing command-line constants

2 vocabulary.{h,cc}

Parse locations

class ParseInfo Stores the location (line, column and file) of a parsed object. This is used to produce clear error messages.

class FormParseInfo Also stores the originally parsed formula (containing brackets, implications, extended existential quantifiers, ...).

Note: almost all objects that can occur in the input files have a pointer to a ParseInfo object. The only exception are domain elements, because storing the parse location of each domain element in a (possibly large) structure seems too much overhead. If some object is builtin, its ParseInfo pointer is set to 0.

Domain elements

union Element A domain element (either an integer, a double, a string, or a compound element). Characters are domain elements of type string.

enum ElementType The four different types of elements

Sorts (types)

Sorts are implemented in the class *Sort*. This class stores the name of a sort, its parent and children in the sort hierarchy and a pointer to the predicate associated to this sort. The base sort of the sort, as well as its depth in the sort hierarchy can in principle be derived from the parent of the sort, but to avoid recomputation, the base and depth are stored as well.

Variables

Variables are implemented in the class *Variable*. This class stores the name and the sort of a variable.

Predicate and function symbols

Predicate and function symbols are implemented by subclasses *Predicate* and *Function* of the abstract class *PFSymbol*. A PFSymbol stores the name of a symbol and its sorts. The name includes the arity of the symbol. For instance, if

```
type A
type B
P(A,B)
F(B) = A
```

is parsed, a PFSymbol with name P/2 and sorts [A,B] is created, as well as a PFSymbol with name F/1 and sorts [B,A].

A predicate has the same attributes as a PFSymbol. A function has a boolean attribute indicating whether the function is partial or not.

Vocabulary

Vocabularies are implemented in the class *Vocabulary*. A vocabulary has a name, a list of sorts, a list of predicate symbols and a list of function symbols.

3 structure.{h,cc}

Tables for sorts

Tables for sorts are implemented in the class *SortTable*. The most important methods are

```
bool finite();
ElementType type();
Element element(unsigned int n);
bool contains(Element e, ElementType t);
```

The first one returns true if the table if finite. The second one returns the type of elements in the table (integer, double or string). The third one returns the n'th element in the table. Note that if the third method is called on an infinite table, GIDL will abort. The returned element has the type of the table (i.e., the type returned by `type()`). Even if the element at position n is an integer, e.g., 5, the returned element will be a pointer to the string `"5"` if the type of the table is *string*. Etc. The fourth method returns true if the given element e of type t belongs to the table. This method can return true even if t is not equal to the type of the table.

The elements of in a SortTable are not necessarily sorted and may contain doubles. Calling the method `sortunique` makes sure the table is sorted and contains no doubles. The elements are sorted as follows:

- All numbers are before all strings that do not represent numbers.
- The numbers are sorted according to $<$ on $\mathbb{R}$.
- The strings that do not represent numbers are sorted lexicographically.

Tables for predicates

Interpretations for predicates are implemented in the class *PredInter*. The interpretations are always 4-valued (but not necessarily strictly 4-valued). An interpretation has four attributes: two pointers to tables (`_ctpf` and `_cfpt`) and two booleans (`_ct` and `_cf`). If `_ct` is true, the table `_ctpf` contains all the tuples that are certainly in the relation. If `_ct` if false, `_ctpf` contains all the tuples that are possibly not in the relation. Similarly, if `_cf` is true, the table `_cfpt` contains all the tuples that are certainly not in the relation. If `_cf` if false, `_cfpt` contains all the tuples that are possibly in the relation. The pointers `_ctpf` and `_cfpt` may point to the same table.

The tables pointed to in a predicate interpretation are implemented in the class *PredTable*. This class is very similar to SortTable, except that it may have more than one column. Each column has its own type. When `sortunique()` is called on a PredTable, its tuples are sorted "lexicographically".

Tables for functions

Interpretations for functions are implemented in the class *FuncInter*. The main methods are

```
PredInter*        predinter();
const Element&    operator[](const vector<Element>& tuple);
```

The first returns the corresponding interpretation for the graph of the function. The second method computes the value of the given tuple according to the function. In case the function is partial, this method returns a non-existing element.

Structures

Structures are implemented in the class *Structure*. A structure has a name, a vocabulary, a list of sort interpretations, a list of predicate interpretations and a list of function interpretations. The interpretation at position n in the list of sort interpretation is the interpretation of the n'th sort of the structure's vocabulary. Similarly for predicates and function symbols.

Note: the table for a sort and its associated predicate is stored at the same address in memory.

4 term.{h,cc}

Term

Terms are implemented in the class *Term*. A term has a list of its free variables as attributes. There are several subclasses of Term:

- VarTerm: a variable. E.g., x. A VarTerm has only one attribute: its variable.
- FuncTerm: a function applied to tuple of arguments. E.g., $F(x, y)$, $x + y$, $G(F(z))$, C .... A FuncTerm has two attributes: its function and its arguments (a list of terms).
- DomainTerm: a domain element. A DomainTerm has three attributes: its sort, its type (int, double or string) and its value.

- AggTerm: an aggregate term. E.g., $\#\{x \mid P(x)\}$. An AggTerm has two attributes: its type (cardinality, sum, product, minimum, maximum) and a set expression.

Set expressions

Set expressions are implemented in the class *SetExpr*. A set expression has a list of its free variables as attributes. There are two subclasses of SetExpr.

- EnumSetExpr: set expressions of the form $\{(\varphi_1, w_1), \ldots, (\varphi_n, w_n)\}$, where the φ_i are formulas and the w_i are terms. The attribute `_subf` is a list of formulas containing the φ_i. The attribute `_weights` contains the corresponding terms w_i.

- QuantSetExpr: set expressions of the form $\{\overline{x} \mid \varphi\}$. Attribute `_vars` represents $\overline{x}$, attribute `_subf` represents φ.

5 theory.{h,cc}

Formulas

Formulas are implemented in the class *Formula*. The attributes of a formula are the list of its free variables and its sign (a boolean). The formula is negated iff the sign is false.

Formula has the following subclasses:

- PredForm: atoms[1] of the form, e.g., $P(t_1, \ldots, t_n)$, $F(t_1, \ldots, t_n) = t_{n+1}$, $t_1 < t_2$, $SUCC(t_1, t_2)$, ... Attribute `_symb` stores the predicate or function symbol that is applied, attribute `_args` is a tuple of terms representing the arguments. If `_symb` is a function symbol, say F, and `_args` is the tuple $(t_1, \ldots, t_{n+1})$, this represents the formula $F(t_1, \ldots, t_n) = t_{n+1}$.

- EqChainForm: formulas of the form

  $$t_1 \sim_1 t_2 \wedge t_2 \sim_2 t_3 \wedge \ldots \wedge t_{n-1} \sim_{n-1} t_n$$

  or

  $$t_1 \sim_1 t_2 \vee t_2 \sim_2 t_3 \vee \ldots \vee t_{n-1} \sim_{n-1} t_n,$$

  where $\sim_i \in \{=, \neq, <, >, \leq, \geq\}$ for each i. If attribute `_conj` is true, the formula is of the former format, else it is of the latter format. Attribute `_terms` stores the terms $t_1, \ldots, t_n$. The comparison operators $\sim_i$ are represented by a pair of a character and a boolean:

[1] I.e., literals, if the sign is taken into account. A similar remark holds for the other subclasses of Formula.

char	bool	$\sim$
$=$	true	$=$
$=$	false	$\neq$
$<$	true	$<$
$<$	false	$\geq$
$>$	true	$>$
$>$	false	$\leq$

The characters are stored in attribute `_comps`, the corresponding booleans in attribute `_signs`.

- EquivForm: formulas of the form $\varphi \equiv \psi$. Attribute `_left` stores φ, attribute `_right` stores ψ.
- BoolForm: Formulas of the form

$$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$$

or

$$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n.$$

If attribute `_conj` is true, the formula is of the former format, else it is of the latter format. Attribute `_subf` stores the formula $\varphi_1, \ldots, \varphi_n$.

- QuantForm: formulas of the form $\forall x_1 \cdots \forall x_n\ \varphi$ or $\exists x_1 \cdots \exists x_n\ \varphi$. If attribute `_univ` is true, the formula is of the former format, else it is of the latter format. Attribute `_subf` stores φ, attribute `_vars` stores the variables $x_1, \ldots, x_n$.

Rules, definitions and fixpoint definitions

Definitional rules are implemented in the class *Rule*. A rule has as attributes its head, its body and the (universally) quantified variables that occur in the head. The head is a PredForm, the body a formula.

Definitions are implemented in the class *Definition*. A definition has as attributes the list of its rules and the list of its defined symbols.

Fixpoint definitions are implemented in the class *FixpDef*. A fixpoint definition has as attributes its type (least or greatest fixpoint), a list of its rules, a list of its defined symbols, and a list of its direct subdefinitions.

Theories

Theories are implemented in the class *Theory*. A theory has as attributes its vocabulary, a list of its sentences, a list of its definitions and a list of its fixpoint definitions. It also has a pointer to a structure. If the sentences or rules in the theory contain domain elements, they are domain elements of that structure. The pointer may be 0 if the theory contains no domain elements.

6 namespace.{h,cc}

Namespaces are implemented in the class *Namespace*. A namespace contains a list of subspaces, a list of theories, a list of vocabularies and a list of structures. It also has a pointer to its parent in the hierarchy of namespaces.

All namespaces are decendants of the global namespace, which can be accessed through `Namespace::global()`.

7 builtin.{h,cc}

The files builtin.h and builtin.cc implement the built-in vocabulary and structure. All symbols in the built-in vocabulary are implicitly part of each vocabulary. E.g., if you call `contains(p)` on a vocabulary, and `p` is a pointer to a built-in predicate, the result will be `true`.

See 14 for information about adding built-in sorts, predicates or functions.

8 visitor.{h,cc}

The classes *Visitor* and *MutatingVisitor* are two visitor patterns for theories. The first one implements a visitor that goes depth-first, left-to-right through theories and formulas, and changes nothing. The second one traverses formulas in the same manner, but changes the children of a formula it visits by the result of visiting the children. The MutatingVisitor is the recommanded one to use when implementing a method that rewrites formulas.

For good examples of the use of Visitor and MutatingVisitor, see the classes NegationPush and EquivRemover in theory.cc.

9 execute.{h,cc}

See Section 14 on adding an inference method.

10 options.h

Contains a struct *Options* for storing the options. There is one global variable options of type Options that stores the current options. This variable is declared in file namespace.cc. To be able to access these options in another file, that file should contain

```
extern Options options;
```

11 error.{h,cc}

All error and warning messages, as well as methods to implement the "verbose" option.

12 Parsing

The parser is implemented in the files lex.ll, parse.yy, insert.h and insert.cc. If you want to add something to the parser, send me (Johan) an e-mail.

13 Guarantees when an object is parsed without errors

(The lists below are not exhaustive)

Guaranteed:

- Vocabulary (TODO)
- Theory (TODO)
- Structure
  - Every predicate and function of the structure's vocabulary has an interpretation.
  - All tables are sorted and do not contain doubles.
  - TODO

Not guaranteed:

- Vocabulary (TODO)
- Theory (TODO)
- Structure
  - Not every sort of the structure's vocabulary need to have a interpretation
  - A structure may be strictly four-valued, i.e., inconsistent
  - TODO

14 What to do ...

... to add an option?

1. Add an attribute to `struct Options` (options.h)

2. Set a default value to that attribute in the constructor of `struct Options`
3. Document the option in the method `usage` (main.cc)
4. Parse the option in the method `read_options` (main.cc)

... to add an inference method?

1. Create a subclass (say A) of `class Inference` (execute.h)
2. In the constructor of A, make sure the types of the input arguments and output argument of the inference method are initialized.
3. Implement the inference method in method `execute` of A.
4. If you want to allow a user to call the method from an execute block, add the line

   ```
   _inferences["name"].push_back(new A());
   ```

   in the method `initialize` of namespace `Insert` (insert.cc). Here, `name` is the name the user uses to call the inference method.

... to add a built-in sort, predicate or function symbol?

TODO

15 Other remarks

- Many classes have a method `string to_string()`, which can be used to print the object (e.g., while debugging).
- Have a look at namespaces with a name ending on `Utils` (e.g., `namespace TableUtils`). They contain many useful methods that are not directly implemented in the main classes. For instance:
  - ElementUtil: methods to clone elements, to convert elements to another type, to create a non-existing element (return value for partial functions), check if an element is an existing element, ...
  - SortUtils: a method to find the closest common ancestor of two sorts.
  - TableUtils: methods to return the least predicate and the least function interpretation with a given arity.
  - StructUtils: method to convert a structure to a theory (of facts).
  - TermUtils: a method to evaluate a term in a given structure
  - FormulaUtils: a method to evaluate a formula in a given structure
  - TheoryUtils:

    * Push negations inside
    * Flatten conjunctions, disjunctions and quantifiers
    * Rewrite equivalences and chains of equalities
    * Apply the Tseitin transformation
    * ...
- TVUtils: compute the inverse of a truth value, the glb of two truth values, ...