Documentation for Merge of CVC4 Quantifiers2 Branch

Andrew Reynolds

1 Summary

The quantifiers2 branch of CVC4 contains support for various approaches quantifiers in SMT, including matching-based quantifier instantiation, counterexample-based quantifier instantiation, finite model finding and rewrite-based instantiation and propagation. This document contains information about the modifications made to CVC4 in the quantifiers2 branch to support these features. Here is a summary of the changes:

1. The theories `TheoryQuantifiers` and `TheoryRewriteRules` were added.
2. A `QuantifiersEngine` was added as a module within `TheoryEngine`.
3. The `Theory` class was updated to communicate with `QuantifiersEngine`, in particular through use of an `Instantiator` class contained in each theory.
4. The SAT solver interface was updated to support some specific commands required by the algorithms for handling quantifiers. Various functions were added to the `OutputChannel` and `Valutation` classes to allow Theory objects to issue these commands.
5. The `TheoryUF` was updated to contain and communicate with the finite model finding module `StrongSolverTheoryUF`.
6. A few other parts of the core were modified to handle the case of quantifiers, including parsing, printing, preregistration and ITE lifting.
7. Other generic utilities to support triggers and matching were added.

1.1 Notes

Most modifications made to existing files in trunk are contained between the comment tags //AJR-hack and //AJR-hack-end. Any modifications made between tags //AJR-hack-temp and //AJR-hack-temp-end can be ignored.

I have highlighted possible improvements in Section 8.

2 Quantifiers

2.1 TheoryQuantifiers

The directory `src/theory/quantifiers` contains definitions for the quantifiers theory.

2.2 TheoryRewriteRules

The directory `src/theory/rewriterules` contains definitions for the rewrite rules theory.

2.3 QuantifiersEngine

Included in `src/theory/quantifiers_engine.h/cpp` are definitions for the following classes:

1. `Instantiator` class, a base class for deciding instantiations. This class contains a collection of instantiation strategies `InstStrategy`. Each theory will contain an instantiator object.
2. `TermDb`, a database for storing all known ground terms that are relevant to quantifiers.
3. `QuantifiersEngine`, the central point of reference for quantifiers.

The `QuantifiersEngine` class is the central point of reference for handling quantifiers in CVC4. Included is a collection of modules it consults for managing quantifiers, utilities for generating instantiation lemmas, and ways of registering pattern terms for matching.

The `QuantifiersEngine` contains a vector of `QuantifiersModule` objects, which it consults when quantifiers are registered, check calls are issued etc. The only such module within `QuantifiersEngine` at the moment is `InstantiationEngine`, which is defined at `src/theory/quantifiers/instantiation_engine.h/cpp`. This module consults all `Instantiator` objects stored in theories for all quantifiers for deciding which instantiations to generate.

3 Modifications to Theory Interface

The following section contains changes made to support the communication between `Theory` objects and the quantifiers module `QuantifiersEngine`.

3.1 `src/theory/theory.h/cpp`

The `Theory` class contains the following new members:

1. QuantifiersEngine* d_quantEngine, a reference to the quantifiers engine.
2. Instantiator* d_inst, the instantiator object for this theory.

The arguments to the constructor for `Theory` has been updated to include a reference the quantifiers engine. I have found this to be the best way to initialize everything, since at construction time the theories will construct instantiators that will need references to the quantifiers engine.

The effort level EFFORT_LAST_CALL has been added to support quantifiers, which requires doing work *after* theory combination takes place.

3.2 src/theory/theory_engine.h/cpp

The TheoryEngine class contains the following new members:

1. theory::QuantifiersEngine* d_quantEngine, the quantifiers engine, which is created within the theory engine constructor.
2. theory::Theory* getTheory(int id), an access function for individual theories directly by id. This has been necessary in various places, for example, I may need to reference TheoryUF to access equivalence class information internal to the theory for matching modulo equality.

In src/theory/theory_engine.cpp line 235, additional work is done when the theory engine passes a full effort check with no lemmas and no conflicts. The check method is called in the quantifiers theory with effort level EFFORT_LAST_CALL. This level is necessary since it has been shown the quantifiers theory needs to perform work after theory combination has taken place. The quantifiers theory will either add a lemma, return with the incompleteness flag set, or simply return after which CVC4 should answer "sat". If no lemmas are added and incompleteness is set, and the "flipDecision" option is active (see Section 5), we consult the quantifiers theory for performing this operation. This may cause CVC4 to continue the search, instead of answering "unknown".

3.3 src/theory/output_channel.h

The output channel has been updated to include these new members:

1. void requirePhase(TNode n, bool phase, ...), requires that literal n be chosen as a decision only with given phase.
2. void dependentDecision(TNode depends, TNode decision, ...), requires that literal decision can only be decided once literal depends has a value.
3. bool flipDecision(), backtrack the search to the most recent eligible decision literal, flip its polarity, and re-assert it.
4. void flipDecision(Node lit), backtracks and flips the polarity of the decision literal lit.
5. void flipDecision(unsigned level), backtracks and flips the polarity of the decision literal at given level.

More information can be found in comments within src/theory/output_channel.h.

3.4 src/theory/valuation.h/cpp

The valuation class has been updated to include these new members:

1. void ensureLiteral(TNode n), ensures that a SAT literal is assigned to the formula n.
2. int getDecisionLevel(), gets the current decision level.
3. Node getDecision(unsigned level), gets the decision made at a particular level.
4. bool isDecision(Node lit), returns true if literal lit is currently asserted as a decision.

More information can be found in comments within src/theory/valuation.h.

4 Modifications to Individual Theories

4.1 TheoryArrays

Added instantiator class `src/theory/arrays/theory_arrays_instantiator.h/cpp`. This class forwards terms in the arrays theory to the quantifier engine for use in matching.

4.2 TheoryArith

Added instantiator class `src/theory/arith/theory_arith_instantiator.h/cpp`. If the counterexample-based quantifier instantiation option is active, this class contains a strategy for quantifier instantiation. This strategy produces instantiations based on information stored in the simplex tableaux for relevant variables.

4.3 TheoryDatatypes

Added instantiator class `src/theory/datatypes/theory_datatypes_instantiator.h/cpp`. If the counterexample-based quantifier instantiation option is active, this classes produces instantiations based on information stored in the datatypes theory concerning the current model.

4.4 TheoryUF

Added instantiator class `src/theory/uf/theory_uf_instantiator.h/cpp`. This class stores additional information that is needed for quantifiers, including properties of equivalence classes, information needed for efficient E-matching, among other things. The instantiator class also contains strategies for matching-based instantiation (defined in files `src/theory/uf/inst_strategy.h/cpp`), and forwards relevant terms to the quantifiers engine for use in matching.

I have also added the CARDINALITY_CONSTRAINT kind to the kinds file for UF. I do not require cardinality constraints to be handled by any parser. Instead, they are internally created and processed by the `StrongSolverTheoryUF` module, and are ignored by `TheoryUF`. In finite model finding mode, the strong solver for theory UF will introduce lemmas and make propagations containing cardinality constraints for the purposes of finding minimal models.

`src/theory/uf/equality_engine_impl.h`: I have required that the templated class `NotifyClass` gives additional notifications in four additional places:

1. NotifyClass::notifyEqClass(...), called when a new equivalence class is created.
2. NotifyClass::notifyDisequal(...), called just before a disequality between two terms is added.
3. NotifyClass::preNotifyMerge(...), called just before two equivalence classes are merged.

4. NotifyClass::postNotifyMerge(...), called just after two equivalence classes are merged.

The third feature is required by efficient E-matching, which generates new candidate terms for matching incrementally based on which merges take place. The remaining three features are needed for finite model finding, which maintains a disequality graph for solving UF+cardinality constraints. This requires that all notifications are given in terms of representatives. All representatives are notified before they are used, using notifyEqClass(...). The call postNotifyMerge(a, b) has the meaning that a is now the representative of the equivalence class of b.

These functions have been added to all notify classes used in conjunction with the equality engine class. This includes `SharedTermDatabase`, `TheoryArrays`, `DifferenceManager`, `TheoryBV` and `TheoryUF`. In all cases apart from UF, these functions have no effect.

Additionally on line 899, I have made the modification to areDisequal(...), which had been causing assertion failures when checking disequality between boolean terms.

`src/theory/uf/theory_uf.h/cpp`: The `TheoryUF` class contains the following new members:

1. StrongSolverTheoryUf* d_thss, a strong solver for UF+cardinality constraints. This class also handles some of the control decisions of the finite model finding algorithm. This class is activated only when the finite model finding option is turned on.
2. notifyEqClass(...), notifyDisequal(...), preNotifyMerge(...), postNotifyMerge(...), these functions forward the notifications from the equality engine to the instantiator and strong solver objects.
3. EqualityEngine < NotifyClass>* getEqualityEngine(), access function to equality engine.

Various parts of `TheoryUF` have been updated to send notifications and requests to the quantifiers engine, the instantiator, and the strong solver.

`src/theory/uf/theory_uf_strong_solver.h/cpp`: These files define the strong solver for UF+cardinality constraints.

`src/theory/uf/inst_strategy_model_find.h/cpp`: These files define the instantiation strategy used for finite model finding.

5 Modifications to SAT Solver Interface

The SAT solver interface has been updated to include a variety of features regarding when/how a literal should be decided upon. The interface also has added queries for various information.

The following has been added to `DPLLSatSolverInterface` (in `src/prop/sat_solver.h`):

1. unsigned getDecisionLevel(), returns the current decision level.
2. bool isDecision(SatVariable decn), returns true if the given variable is asserted as a decision.
3. SatLiteral getDecision(unsigned level), return the literal decided at the given decision level.
4. void requirePhasedDecision(SatLiteral lit), requests that a literal only be chosen in the polarity given.
5. void dependentDecision(SatVariable dep, SatVariable dec), requests that literal with variable dep be decided before literal with variable dec.
6. bool flipDecision(), requests a backtrack to occur. In particular, we find the most recent decision literal, backtrack the search to this literal, and assert it with the opposite polarity.
7. void flipDecision(SatVariable decn), requests the literal with variable decn to be "flipped".

For more information about these features, see the comments above their declaration in `src/theory/output_channel.h`.

5.1 MiniSat

The minisat interface in `src/prop/minisat/minisat.h/cpp` implements these functions by calling the appropriate functions in `Minisat::SimpSolver`. The base class of minisat `Solver` in `src/prop/minisat/core/Solver.h/cpp` has been updated in a few places to support these features, including the following new members:

1. void freezePolarity (Var v, bool b), declare which polarity the decision heuristic must always use for a variable.
2. void dependentDecision(Var dep, Var dec), declare that deciding on "dec" depends on "dep" having an assignment.
3. void setFlipVar(Var v, bool b), declare if a variable is eligible for flipping.
4. bool flipDecision(), backtrack and flip most recent decision.
5. void flipDecision(Var v), backtrack and flip the given decision.
6. vec<char> flippable, declares if a variable is eligible for flipping with flipDecision().
7. vec<Var> depends, the variable (if any) that a decision on this variable depends on
8. vec<Var> dependsOn, variables whose decision depends on this variable.
9. vec<int> flipped, which decisions have been flipped in this context.

A few minor changes were made to the implementation to maintain the new data members. I also made the access functions int decisionLevel(), bool isDecision(Var x) and Lit getDecision(unsigned lvl) public.

6 Other Modifications

6.1 Parser

The SMT2 parser has been updated to support quantifiers (including user-defined patterns) and rewrite rules, and supports the setting of various quantified SMT logics.

6.2 Printing

Various printers have been updated to include quantifiers and rewrite rules, including `src/printer/cvc/cvc_printer.cpp` and `src/printer/smt2/smt2_printer.cpp`. This is still in progress.

6.3 ITE Lifting

To ensure bound variables do not escape due to ITE lifting, the bodies of quantifiers are ignored, see `src/util/ite_removal.cpp` line 78.

6.4 Pre-Registration

To avoid propagating literals involving bound variables, the bodies of quantifiers are ignored during pre-registration, see `src/theory/term_registration_visitor.cpp`, line 35.

6.5 Options

The following options have been added to CVC4:

1. –disable-miniscope-quant: disable miniscope quantifiers
2. –disable-miniscope-quant-fv: disable miniscope quantifiers for ground subformulas
3. –disable-prenex-quant: disable prenexing of quantifiers
4. –var-elim-quant: enable variable elimination of quantifiers
5. –disable-smart-multi-triggers: disable smart multi-triggers
6. –finite-model-find: use finite model finding heuristic for quantifier instantiation
7. –use-fmf-region-sat: use region-based SAT heuristic for fmf
8. –efficient-e-matching: use efficient E-matching
9. –literal-matching=MODE: choose literal matching mode
10. –enable-cbqi: turns on counterexample-based quantifier instantiation [on by default only for arithmetic]
11. –disable-cbqi: turns off counterexample-based quantifier instantiation
12. –disable-flip-decision: turns off flip decision heuristic

For more details, see `src/util/options.h`.

7 Utility Classes for Quantifiers

The files `src/theory/inst_match.h/cpp` and `src/theory/inst_match_impl.h` contain the utilities for performing matching on terms.

The files `src/theory/trigger.h/cpp` contain utilities for defining and using triggers.

The files `src/theory/uf/theory_uf_candidate_generator.h/cpp` defines an implementation of the base class `CandidateGenerator`. This class encapsulates how ground terms are chosen for matching according to various methods.

8 Suggestions for Improvement

8.1 General Framework for Querying Equalities and Iterating on Equivalence Classes

For matching modulo equality, I have required a way of querying whether two terms are equal/disequal, getting representatives for equivalence classes, and iterating over terms in equivalence classes.

Since the code seems to be changing a lot, I am currently using my own generic interface for all queries related to equality, disequality and representatives. This is the class `EqualityQuery` in the file `src/theory/inst_match.h`. The current implementation of matching uses the `EqualityQueryInstantiatorTheoryUf` class, as defined in `src/theory/uf/theory_uf_instantiator.h`. However, this is somewhat incomplete since it may not be aware of equalities in other theories. I would like to make this more general, perhaps using some of the updated sharing code.

I have been using an iterator `EqClassIterator` defined in `src/theory/uf/theory_uf.h` for iterating over equivalence classes. Note that this iterator looks at the equality engine stored with `TheoryUF`, and thus may not perform correctly for equivalence classes of other theories, in particular Arrays and Datatypes. On top of this, I am using a candidate generator `CandidateGeneratorTheoryUf` in `src/theory/uf/theory_uf_candidate_generator.h/cpp` for producing ground term candidates for matching. This also ideally should be more general.

8.2 Instantiator Framework

I have added lines of code to each Theory to call its corresponding Instantiator with ceratin functions such as check, preregister for terms, etc. Perhaps it may be better to standardize this, in other words, make it so that the instantiators of theories are automatically notified when certain theory functions are called.

Again, the purpose of these instantiator objects is to store any theory-specific information that is helpful for quantifiers. Since this may add overhead, instantiators should not be created when quantifiers are inactive. I'm not sure if this is occurring currently.

8.3 Cardinality Constraint Nodes

Cardinality constraints currently take two arguments so you'll have constraints like (CARDINALITY_CONSTRAINT t n), meaning the cardinality *of the type of term* t is n. Ideally we'd rather just have (CARDINALITY_CONSTRAINT T n), where T is a type. I wasn't sure how make operators over `TypeNode` objects.