1 Declaring Terms

There are 3 kinds of log lines which can be used to declare terms: `mk-app` for regular terms, `mk-var` for quantified variables and `mk-quant` for quantifiers. Each term is assigned an identifier to reference it later on (e.g. as an argument of another term). Identifiers consist of an optional namespace (must be a valid SMTlib identifier) a `#` character and an integer in decimal notation. Identifiers may, moreover, be reused and any reference to a term references the most recent definition up to the log-line where it was referenced. The following gives more detail on these types of log-lines:

`[mk-app]` lines start with `[mk-app]` followed by the identifier for a newly created term, the function that was applied, and 0 or more identifiers of the term's arguments. Note that the name `pattern` is reserved and should only be used to wrap trigger terms of quantifiers.

`[mk-var]` is followed by the identifier of a newly created quantified variable and the index (starting from 0) of the variable which will be used to determine which term in a list of bound terms was bound to this variable (it is possible to redefine variables each time a quantifier instantiation is triggered to update these indices).

`[mk-quant]` is followed by the identifier of a quantifier, the name of the quantifier, the number of variables the quantifier quantifies over, 1 or more triggers (these must be applications of `pattern` which is a reserved function name), and the body of the quantifier. For theory axioms namespaces can be used for the identifiers of terms, variables and quantifiers. E.g. the identifier `datatype#1` refers to the element with the identifier `1` in the `datatype` namespace. The name of the namespaces should be indicative of the theory the element belongs to.

1.1 Pretty Printing Hints

It is, moreover, possible to provide hints to the Axiom Profiler to pretty print constants and quantified variables:

`[attach-meaning]` followed by the identifier of a constant, the name of the theory to which this constant belongs, and a SMTlib compatible expression, is used to provide additional (theory specific) information about constants. The provided expression may be rewritten by the Axiom Profiler to improve readability. Currently such rewritings are only implemented for arithmetic expressions (e.g. rational numbers are rewritten with the division as infix notation). If you wish to extend the Axiom Profiler's rewriting capabilities please contact viper@inf.ethz.ch or create a pull request by modifying QuantifierModel/TheoryMeaning/TheoryMeaningInterpretation.cs. Algebraic numbers should be represented using `"root-obj"`-expressions. These are similar to those defined by

the Mathematica language: the first argument provides a polynomial (`x` is used for the variable); the real roots of that polynomial are ordered ascending and the index (*indexed starting from 1*) of the root corresponding to the constant is provided as the second argument. For example $\sqrt{2}$ would be represented as `(root-obj (+ (^ x 2) (- 2)) 2)` since the polynomial $x^2 - 2$ has two real roots: $-\sqrt{2}$ and $\sqrt{2}$, the second of which is $\sqrt{2}$. Rewritings by the Axiom Profiler are currently supported for expressions of the form `(root-obj (+ (^ x` i`) ` j`) ` h`)` for integers i, j and $h \in \{1, 2\}$, i.e. for numbers of the form $\pm\sqrt[i]{j}$.

`[attach-var-names]` is followed by the identifier of a quantifier and the names and sorts of its quantified variables (in the order given by the indices of those variables). The name and sort of a variable should be given as a pair of the form `(`*`name`*` ; `*`sort`*`)`, where *name* and *sort* are the name and sort of the quantified variable respectively. If either of these pieces of information should not be specified, they can be left out: e.g. `(;Int)` would specify that a quantified variable has sort `Int` and `(x;)` would indicate that a variable has the name "x" without specifying its sort. Any leading or trailing whitespace of variable names and sorts is ignored.

2 Quantifier Instantiations

Quantifier instantiations are logged in two stages: first we log that an instantiation is possible (e.g. a pattern was matched) using either `inst-discovered` (in the general case) or `new-match` (for pattern matches) then we instantiate the quantifier using an `instance` / `end-of-instance` block that contains the results of the instantiation.

`[inst-discovered]` is followed the name of the method used (e.g. "MBQI", should not contain spaces) to discover a new instantiation. After that a hexadecimal fingerprint (prefixed with "0x"), the identifier of the corresponding quantifier, and the identifiers of the terms that are bound to the quantified variables (with indices corresponding to those defined in the `[mk-var]` lines. This line should not be printed for trigger based instantiations. For such instantiations a `[new-match]` line (see below) should be printed instead. For instantiations of theory axioms the method "theory-solving" should be used. Axioms that cannot be formulated as quantifiers can be logged by only indicating a namespace and omitting the bindings. Such a log line may look like: `[inst-discovered] theory-solving 0x0 datatype#`. Additionally blame terms, i.e. terms that caused this instantiation, may be explicitly assigned by adding a `";"` followed by the identifiers of these blame terms to the end of the line.

`[new-match]` followed by a fingerprint in the form of a hexadecimal number (prefixed with "0x"), the identifier of the relevant quantifier, the identifier of the trigger that was matched a list of term identifiers of terms that were bound to

the quantified variables, a ";", and a list of terms and equalities that were used for the match:

- Single term identifiers to indicate terms that were matched against the top-level terms in a (multi-)pattern.

- Pairs of identifiers to indicate an equality. E.g. `(#1 #2)` indicates that terms 1 and 2 are equal and 1 is a sub-term of a term originally matched against the trigger that is replaced with term 2 to obtain the pattern match.

This line may also be used for instantiations of theory axioms that have trigger-like instantiation behavior, i.e. axioms that are instantiated when certain term structures are encountered.

For the Axiom Profiler to be able to reconstruct equalities used for pattern matches `eq-expl` lines have to be used: `[eq-expl]` is followed by the identifier of a term and a description of a step such that one can follow these steps to the root (of the union-find data structure representing) the equivalence-class of that term. The log has to guarantee that all paths from terms used in an equality or bound to a quantified variable for a trigger match are complete and up-to-date by the time the corresponding `[new-match]` line is printed. Possible step explanations are:

- `root` indicating that the term is already the root of its equivalence-class

- `lit` followed by by the identifier of an equality term, a ";", and a term the original term is equal to as indicated by the equality term.

- `cg` followed by a list of pairs giving equalities between the arguments of the starting and ending terms of the step, a ";", and the term we reach with this step. This step indicates that two terms are equal because of the congruence closure property ($x = y \implies f(x) = f(y)$). The first element of the pairs should always be an argument of the term the step starts from. Commutativity is supported. An example of such a log line can be found in Fig. 1.

- `th` followed by the name of a theory solver, a ";", and the identifier of a term the original term is equal to according to an equality added by that theory solver.

Finally `[instance]` is followed by the fingerprint of the match and the identifier of the term produced by this instantiation. The following lines until `[end-of-instance]` contain updates to the e-graph that occur as a result of that instantiation. These updates are represented using `[attach-enode]` followed by the identifier of a term that was attached to the e-graph, i.e. became active. The Axiom Profiler uses

```
[mk-app] #5 f #1 #2
[mk-app] #6 f #3 #4
[eq-expl] #5 cg (#1 #3) (#2 #4) ; #6
```

Fig. 1. An example for what a congruence equality explanation step may look like.

these updates to construct causality relations between quantifier instantiations: if some term that was attached to the e-graph during instantiation A is used to trigger instantiation B then A caused the instantiation of B.

3 Miscellaneous

`[tool-version]` followed by the name of the tool that generated the log and its version as the first log line. These lines are used for compatibility checks.

`[mk-proof]` followed by an identifier (same namespace as `[mk-app]`, `[mk-quant]`, and `[mk-var]`) the name of the proof step (currently ignored by the Axiom Profiler), the identifiers of the prerequisites of the proof step and the identifier of the result of the proof step. Used to justify a proof step. Can only be declared in the main namespace. Referencing a proof step is the same as referencing its result term.

`[begin-check]` indicates that a `(check-sat)` command is being executed. This allows the user of the tool to only load data from a single check if they generated a log from a SMT file containing multiple checks.

`[assign]` followed by the identifier of a literal or `(not #<id>)` and a justification for assigning that literal. This data is mostly unused by the Axiom Profiler at the moment.

`[conflict]` followed by a set of decision literals that resulted in a conflict. This data is mostly unused by the Axiom Profiler at the moment.

`[push]` followed by the number of child-scopes that are active after the push. This data is mostly unused by the Axiom Profiler at the moment.

`[pop]` followed by the number of scopes to pop. This data is mostly unused by the Axiom Profiler at the moment.