TSMT Tutorial

Mikhail Mandrykin

May 6, 2020

Contents

1 About TSMT

TSMT is a set of Isabelle proof methods and commands that facilitates the use of relatively complete decision procedures, such as SMT solvers (potentially including Argo) and paramodulation-based provers (including Metis), with deterministic quantifier instantiation based on *bounded E-matching*. In particular, with some reasonable additional overhead, it enables extraction and inspection of counterexamples and quantifier instantiations, automatic proof optimization based on unsatisfiability cores, trigger management and some other small, but often useful features.

We proceed this tutorial with introduction to the problem of quantifier instantiation, traditional E-matching, bounded E-matching and the solutions implemented within TSMT tooling.

2 Quantifier instantiation and E-matching

For the vast majority of practically relevant logical fragments involving quantifiers the corresponding satisfiability problems are generally undecidable. In practice this means there are no complete or even in some sense "best" approaches to automatically proving most goals involving quantifiers. All existing approaches are in some sense heuristical. However, the approaches also tend to be adapted to the particular needs of their underlying reasoning engines. For example, quantifier instantiation for rewriting-based proof methods (e. g. *simp* and *auto*) is based on special cases of higher-order resolution (introduction and elimination rules with specially designated major premises and conclusions). Meanwhile, quantifier instantiation for methods based on Satisfiability Modulo Theories (SMT) is often based on *E-matching*. The reason for this is completeness of many SMT-based decision procedures for ground (quantifier-free) fragments of the corresponding logics. In particular, the combination of linear integer arithmetic and equality with uninterpreted functions (the so-called UFLIA logic) is decidable and, moreover, most decision procedures for this logic are very efficient in practice (although they are generally NP-complete and in theory can require exponential amount of resources). The efficient decidability for ground formulas essentially implies that given the appropriate set of relevant quantifier instances, the proof of unsatisfiability (and, dually, validity) of the original formula can usually be obtained rather quickly. So there's little to no need in any syntactic heuristics for the refutation (or inference) reasoning itself, but rather the hueristical part of the decision procedure is restricted to the choice of appropriate quantifier instantiations. Hence the particular syntactic shape of the rules (introduction, elimination, destruction, conguruence etc.) is irrelevant, what's important is what terms should be used to instantiate the variables bound by the quantifiers.

Consider the following sample ground formula (the example is from [2]):

$$(P\ (f\ 42) \longrightarrow 42 < 0) \wedge a = f\ 42 \wedge P\ a.$$

A decision procedure for the combination of linear integer arithmetic and equality with uninterpreted functions (the UFLIA logic) can easily prove this formula unsatisfiable. Consequently, in HOL it can also prove the following goal:

$$[\![P\ (f\ 42) \longrightarrow 42 < 0;\ a = f\ 42]\!] \Longrightarrow \neg\ P\ a.$$

However, in practice a more likely formulation of such a goal is a more general quantified one:

$$[\![\forall x.\ P\ (f\ x) \longrightarrow x < 0;\ a = f\ 42]\!] \Longrightarrow \neg\ P\ a.$$

To prove it with SMT, the corresponding SMT proof method first negates it:

$$(\forall x.\ P\ (f\ x) \longrightarrow x < 0) \wedge a = f\ 42 \wedge P\ a,$$

then extracts the unsatisfiability proof from a proof-producing SMT-solver (such as Z3). The obtained unsatisfiability proof is then transformed into the validity proof of the original formula. Yet to produce the

proof the SMT solver should come up with an appropriate instantiation for the quantified variable x. In general, there is an infinite set of potentially relevant instantiations for the variable x, such as 42. But once the necessary instantiation is obtained, the formula can be relatively efficiently discharged since there is an existing complete decision procedure for the ground logic.

One of the approaches to selection of the relevant terms for instantiation of bound variables is called *E-matching*. It suggests to initially designate certain special terms containing occurrences of the target bound variables and call those subterms *triggers*. Then it suggests to choose only those instantiations of the bound variables that make those triggers equal to some ground subterms of the currently considered formula.

In our example let's designate the term $P\ (f\ x)$ as a trigger. Now to extract the instantiations relevant for this trigger we have to *match* the ground subterms of the formula with the trigger to obtain the substitutions of the variable x that make the term $P\ (f\ x)$ equal to some ground subterm of the formula. However, in our example there's no such subterm! The ground part of the formula $a = f\ 42 \wedge P\ a$ does not contain any term of the form $P\ (f\ x)$. Here the "E" part of the matching approach comes to our help. The "E" means that the matching is not considered literally, but rather *modulo equality*, i. e. if there is an equality of some term t with another term u $(t = u)$ implied by the current formula and the term $f\ t$ is its ground subterm, then $f\ u$ is also considered a ground subterm of the formula, even though it is not syntactically present in it.

In our example since equality $a = f\ 42$ is implied by the formula and the term $P\ a$ occurs in it, we consider $P\ (f\ 42)$ a ground subterm of the formula. Then the bound variable x can be matched with 42 and the required instantiation $P\ (f\ 42) \longrightarrow 42 < 0$ can finally be produced.

For now, we have not yet elaborated what is considered "current formula" and what is considered an equality "implied" by it. This is actually the exact part where one of the key differences between the traditional E-matching and the approach implemented in TSMT is to be found.

3 Traditional E-matching, its advantages and limitations

Traditional E-matching is usually implemented within the DPLL/CDCL framework [1] of a typical SMT-solver as part of the theory of quantifiers. It interleaves its work with the main DPLL core of the solver that enumerates possible models of the propositional skeleton of the normalized formula represented as the set of currently relevant CNF clauses (disjuncts). So the subterms of the formula are the literals of this CNF representation and the implied equalities are, naturally, the equalities implied by the current model of the propositional skeleton resulting from invocation of the decision procedure for the theory of equality and uninterpreted functions (the congruence closure algorithm) and other theories currently activated in the particular solver (such as linear arithmetic). The instantiations obtained during E-matching are normalized into CNF disjuncts and added to the current CNF representation of the formula. The extended CNF is passed to the DPLL core of the solver. The process is then repeated upon the next completed propagation of the current candidate model of the propositional skeleton.

This approach, in particular, implies that compound predicates containing propositional connectives such as $\wedge$ or $\vee$ can not be part of triggers as the syntactic propositional structure of non-atomic predicates is not preserved by CNF normalization. For this reason, triggers should not contain propositional connectives. To support matching on several disjoint terms simultaneously special *conjunctive triggers* are allowed, but they are matched independently regardless of the propositional structure. For instance, to match a conjunction of two triggers e. g. $P\ (f\ x)$ and $g\ y\ z$ we should find two terms of the corresponding form anywhere in the formula, e. g. a subterm $P\ (f\ (g\ a\ b))$ occurring in some CNF literal is already sufficient to match both triggers (giving substitutions $x = g\ a\ b$, $y = a$ and $z = b$).

Also, the approach implies that terms obtained as a result of instantiation are also subsequently considered subterms of the current formula and can be matched against to produce new instantiations. More specifically, it implies that *iterated instantiation* is performed and so-called instantiation loops are entirely possible and are notoriously hard to eliminate. Consider, for instance, the following property:

$$\forall\, x.\ (\textbf{when}\ f\ x\textbf{:}\ f\ x < f\ (f\ x)).$$

Here we use our notation for triggers $\forall\, x.$ (**when** $T\ x$**:** $P\ x$) to denote that the quantified predicate $P\ x$ is instantiated upon matching the trigger $T\ x$. If a term $f\ a$ occurs in some literal of the current formula, then the property is instantiated with the substitution $x = a$ and the conjunct $f\ a < f\ (f\ a)$ is added to the formula. Then the term $f\ (f\ a)$ becomes a subterm of a literal, so on the next iteration the property can be instantiated again with the substitution $x = f\ a$ producing a new conjunct $f\ (f\ a) < f\ (f\ (f\ a))$. This process is diverging as the instantiation can be repeated indefinitely. As the matching is performed modulo equality, detecting possible instantiation loops in advance becomes especially hard and most state-of-the-art SMT solver implement only very simple heuristics for that purpose that very often fail to detect the loops. Possible instantiation loops and matching modulo current congruence relation that is implied by the latest model of the propositional skeleton as well as the non-deterministic interleaving between the quantifier instantiation and propositional model search make the behavior of quantifier instantiation in most modern SMT solvers very hard to predict or even debug using specialized tools, such as Z3 axiom profiler. Before turning to our alternative approach to E-matching, specifically crafted for use in interactive theorem provers, we should note two most important advantages of traditional E-matching:

- The first is very straightforward definition of the congruence[1] relation. The relation is fully known at any time when instantiation is performed, although it can change between the rounds of instantiation together with the current model. In particular, it means that whenever we match a term $f\ x$ against, say, a term $f\ a$ given some implied equalities, say $a = b$ and $a = c$, we do not have to attempt enumerating the entire equivalence class of the term a to produce all possible matches modulo equality i. e. $f\ b$ and $f\ c$. Since if $a = b$ is implied by he congruence relation, then $f\ a = f\ b$ is also implied, the same holds for c, and thus the only instantiation $f\ a$ is sufficient to enumerate the entire equivalence class of the substitution $x = a$. This significantly reduces the number of relevant instantiations and boosts the efficiency of E-matching. For a moment, imagine we only know that a *might* be equal to b, and don't have a concrete congruence model. Then we cannot in general rely on such optimization. This problem is somewhat relevant in TSTM.

- The second advantage is efficiency due to very tight coupling between the quantifier instantiation and the DPLL/CDCL core of the solver. In particular, we don't have to define when to stop our attempts at quantifier instantiation, as the instantiation and satisfiability check steps are interleaved and once we have enough instances to show the unsatisfiability we immediately get the verdict (and the proof). This partially alleviates the problem with instantiation loops, but not completely as the loops often make enumeration of instances diverge indefinitely without ever obtaining the relevant instantiations. Yet in general, existing modern SMT solver implementations are quite efficient at generating relevant instances and can in practice solve many realistic problems with several dozens of quantifiers. We also note another limitation of interleaving between stasisfiability checking and matching: the poor definition of counterexample. As there is no definite criterion for stopping the instantiation, the precise formula that is satisfied by the counterexample model is unknown as it can potentially include a wide range of possible instantiations. This prevents the use of counterexamples produced by SMT solvers in many practical applications including interactive theorem proving.

4 Bounded E-matching in TSMT

TSMT implements a rather simple E-matching procedure that is tailored for predictability and flexibility that are relevant for proof methods in interactive proof assistants. We can summarize three most important characteristics of the TSMT approach as follows:

- E-matching is performed only once before the satisfiability check with SMT. This makes the counterexample model much better defined, but requires all relevant instances to be generated in advance without relying on any knowledge implied by the target formula. Also, this makes matching less

[1] *Congruence relation* is a special case of equality relation that is closed under the congruence rule $\forall f\ x\ y.\ x = y \longrightarrow f\ x = f\ y$.

efficient in cases where only a small fraction of generated instances is later used to infer the unsatisfiability.

- Since no congruence relation is available to the instantiation procedure, TSMT resorts to approximation. Namely, a non-conservative over-approximation (*may-congruence*) and an under-approximation (*must-congruence*) are maintained to still support matching modulo equality. The approximations are syntactic and are based on occurrences of the equality symbol inside the formula. All occurrences of the equality symbol are treated as implied equalities in may-congruence, but only the equality symbols occurring under no connective other than conjunction are treated as implied in must-congruence. However, both the may- and must-congruence relations are always guaranteed to be closed under congruence. Matching is performed modulo may-congruence, but the pruning of equivalent instantiations is performed modulo must-congruence. Also, for the sake of efficiency, matching of any ground term with a bound variable is not performed modulo equality. So E-matching is only performed when matching against applications. Thus in the following example

  $$(\forall\, x.\ (\textbf{when}\ P\ (f\ x)\textbf{:}\ P\ (f\ x) \longrightarrow x < 0)) \wedge (a = b \vee a = c) \wedge a = d \wedge u = f\ a \wedge f\ b < f\ d \wedge P\ u$$

  may-congruence includes equalities $a = b = c = d$ and $u = f\ a = f\ b = f\ c = f\ d$, while must-congruence includes only the equalities $a = d$ and $u = f\ a = f\ d$, and therefore the resulting instances would be $P\ (f\ a) \longrightarrow a < 0$ and $P\ (f\ b) \longrightarrow b < 0$. The instantiation $P\ (f\ d) \longrightarrow d < 0$ is not considered because $a = d$ is implied by must-congruence and therefore this instantiation is necessarily equivalent to $P\ (f\ a) \longrightarrow a < 0$ and so is redundant. The instantiation $P\ (f\ c) \longrightarrow c < 0$ is not considered because matching against bound variable x is not performed modulo equality and the formula doesn't directly contain subterm $f\ c$. However, matching against subterm $f\ x$ (an *application* of f to x) is performed modulo may-congruence and since terms $f\ a$ and $f\ b$ occur in the formula and are may-congruent to u, they are both considered relevant.

- The instantiation is in general not iterative, i. e. earlier obtained instances of some property are not considered to be ground terms of the formula when matching against triggers of that property. Instead, there is explicit ordering of properties so that instantiations of preceding properties are considered ground terms when instantiating succeeding properties, but not vice versa. However, by itself this approach may be somewhat over-restrictive. Consider the following property:

  $$\forall\, x\ y\ z.\ (\textbf{when}\ g\ x\ (g\ y\ z)\textbf{:}\ g\ x\ (g\ y\ z) = g\ (g\ x\ y)\ z).$$

  It allows us to prove the following identity:

  $$g\ a\ (g\ b\ (g\ c\ d)) = g\ (g\ (g\ a\ b)\ c)\ d$$

  by iteratively instantiating the property twice, first getting

  $$g\ \underbrace{a}_{x}\ (g\ \underbrace{b}_{y}\ \underbrace{(g\ c\ d)}_{z}) = g\ (g\ a\ b)\ (g\ c\ d),$$

  and then

  $$g\ \underbrace{(g\ a\ b)}_{x}\ (g\ \underbrace{c}_{y}\ \underbrace{d}_{z}) = g\ (g\ (g\ a\ b)\ c)\ d.$$

  Since the size of such subterms in the original formula may not be bounded in general, it's desirable to still be able to perform iterated instantiation. But if this instantiation never terminates, since we do not interleave instantiation with satisfiability checks, we have no chance of ever proving the goal. To avail some restricted form of iterated instantiation that can guarantee termination TSMT uses *bounded* version of E-matching. Basically, it means that when matching a term modulo equality, the size of the resulting matched term should in some sense not exceed the size of the initial outermost matched term.

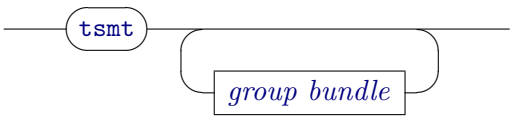
Let's consider the last point in more detail. Assume we are matching a trigger $P\ (f\ x)$. To initially match against that trigger we must find an occurrence of an application of P to some argument. This application is our initial outermost term. Let it be $P\ u$. Then we should match against the trigger $f\ x$ modulo equality to the term u. Let's assume we found a term $f\ b$ occurring in the formula and the equality $f\ b = u$ is implied by may-congruence. So to finally instantiate the property with $x = b$ we have to check that $size\ (P\ (f\ b)) \leq size\ (P\ u)$ for some fixed definition of the function *size*. We can already see that traditional structural definition of size would be too restrictive here, but practical experience also showed that without imposing any similar restriction the size of the formula can grow too quickly during E-matching even in absence of instantiation loops (so that a lot of irrelevant, but exceedingly large instances are generated during instantiation).

For this reason TSMT implements three special heuristics to define the size function in a practically suitable way:

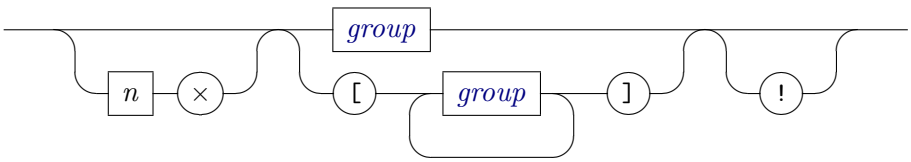
- First, the size is computed just as normal structural size of the term, but certain symbols can be assigned size greater than *1*. For instance, normally the size of the term $P\ u$ is equal to $size\ P + size\ u = 1 + 1 = 2$, but if we allow the size of symbol u to be equal to *2*, then $size\ (P\ u) = 1 + 2 = 3 \geq size\ (P\ (f\ b)) = 1 + 1 + 1 = 3$ and the desired instantiation is allowed. This heuristic is especially relevant for functions specified by their definitions. For instance, if we have a definition $u\ x \equiv f\ (g\ x)$, it is natural to assign the symbol u the size *2* so that the term $u\ x$ has the same size as the term $f\ (g\ x)$ for any x.
- Second, TSMT computes maximal sizes of terms modulo must-congruence. So if we have an equality $u\ x = f\ (g\ x)$ implied by the must-congruence, then the size of the term $u\ x$ is regarded to be not less than (i. e. at least) the size of the term $f\ (g\ x)$.
- Third, TSMT increases the size of every term by a special *margin* value. The margin is computed at definite explicitly specified points during instantiation and is equal to the maximal size of any non-Boolean term occurring in the formula (including all previous instantiations) at that point. Also, by default the margin is always computed at least on the original formula before starting any instantiations. Then it can be recomputed on demand.

5 The *tsmt* method

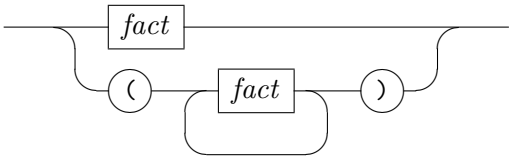
Let's now consider the main proof method of the TSMT toolset. The syntax of *tsmt* method is the following:



group bundle



group



Properties for instantiation are specified as a sequence of groups. Each group either consists of a single property that is instantiated against the goal conjoined with all previously obtained instances, or a set of properties for iterated instantiation. In case of iterated instantiation, the instances obtained during instantiation of the properties in the group are conjoined with the formula and are fed again for instantiation of the properties from the group. The process is repeated until no new instantiations can be obtained. It's important that the instantiation is always performed using bounded E-matching with sizes of terms computed as described in the previous section. The must-congruence and the margin are not recomputed during instantiation of properties from an iterated instantiation group. This allows for iterative instantiation of some non-trivial property groups without the risk of looping. However, in this tutorial we do not describe any precise termination criteria that can be used to detect possible looping. Currently TSMT itself does *not* check or inforce termination within iterated instantiation groups. Nonetheless, a formal proof of one such possible termination criterion is presented in session `TSMT_Termination`. In future there is a plan to implement automatic splitting of interative property groups according to this conservative termination criterion in a separate proof method or command, so that every resulting property instantiation group can be made necessarily terminating. But since the criterion is still only an approximation there's no plan to prohibit possibly non-terminating instantiation groups altogether.

A subsequence of non-iterated single-property *group*s is specified as a *fact*, i. e. a standard reference to a list of theorems in Isabelle. Thus a *fact* inside an application of *tsmt* method is treated as a sequence of single-property groups. Unlike most proof methods in Isabelle, TSMT currently only accepts either schematic rules with explicit triggers or rules fully instantiated using the attribute *of*. No meta or object logic quantifiers ($\bigwedge$, $\forall$, $\exists$ etc.), or schematic rules without explicitly specified triggers are allowed. All quantification must be represented using only schematic variables (e. g. *?a*). In future there is a plan to implement automatic prenex normalization, skolemization and heuristical inference of triggers to provide some support for nested quantifiers and usual Isabelle rules.

An iterated instantiation *group* is specified as a set of *facts* that is treated as the flat set of the corresponding properties. The set comprising an iterated group is specified in parenthesis.

Every instantiation group can finish with a bang (`!`) designating a recomputation of the term size margin. Also, for convenience it is possible to bundle a sequence of either non-iterated or iterated instantiation groups together using square brackets (`[` and `]`) and repeat such *group bundle* an explicitly specified number of times (*n*).

Once instantiation is finished, the resulting instances are ultimately supplied as chained facts to the *smt* proof method [2].

6 TSMT by example

Here we present several examples to illustrate the use of *tsmt* proof method and some of its features that are beyond simple transformations of the current goals and the proof context.

Let's assume we have the following definitions:

definition *"f x ≡ x"*

definition *"P x ≡ x < 0"*

Now we want to prove the sample property illustrating the E-matching modulo must- and may-congruence as was presented in Section 4. First we need to specify the property:

$$\forall x.\ (\textbf{when}\ P\ (f\ x)\textbf{:}\ P\ (f\ x) \longrightarrow x < 0).$$

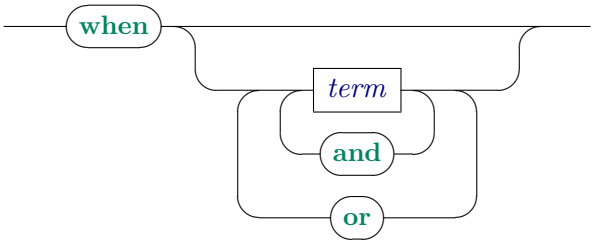
To do this we first prove the corresponding lemma:

lemma *sample_prop_lemma*: *"P (f x) ⟹ x < 0"* **unfolding** *f_def P_def* **.**

[2] In reality TSMT partially re-implements *smt* method, but this only concerns model extraction and proof optimization facilities presented further.

6.1 *when* attribute

Now we need to turn this lemma into a property with explicit triggers. To do this TSMT provides a special attribute *when* with the following syntax:



This attribute provides a way to specify several sets of conjunctive triggers. Sets are separated by the **or** keyword while the triggers in a set are separated by the **and** keyword. The special case of empty set of triggers is treated as unconditional instantiation and is only applicable to ground properties that have no occurrences of schematic variables. This way we can, for instance, unconditionally insert definitions of some constants e. g. $MAX\_BYTE \equiv 255$. To attach our trigger $P\ (f\ x)$ to the lemma we transform it with the *when* attribute:

lemmas *sample_prop* = *sample_prop_lemma*[**when** "$P\ (f\ x)$"]

Now we can try to prove our sample property. Let's first formulate it in the following way:
lemma "$(a = b \vee a = c) \wedge a = d \wedge u = f\ a \wedge f\ b < f\ d \wedge P\ u \Longrightarrow b < 0$"
We use the *tsmt* method to instantiate our sample property and prove the goal:

lemma "$(a = b \vee a = c) \wedge a = d \wedge u = f\ a \wedge f\ b < f\ d \wedge P\ u \Longrightarrow b < 0$"
  apply (*tsmt sample_prop*)
  oops

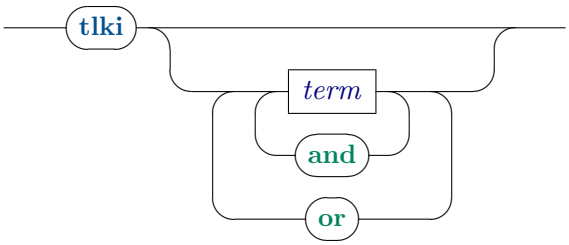
6.2 Counterexample report

However, we can see that the application of the method is not successful. Upon unsuccessful invocation *tsmt* method outputs a report on the results of instantiation and a reconstructed counterexample model. The report includes the following parts:

- The goal state printed with counterexample model highlighting available upon mouse hover with the key `Ctrl` pressed and held;
- The report on the number of instantiations for every property, which is shown separately for each the set of conjunctive triggers. If the same property with the same set of conjunctive triggers occurs several times in the sequence supplied to the *tsmt* method, the report distinguishes the corresponding occurrences and their corresponding instantiations by the occurrence index (the "**at** *n*" part) counted from *1*;
- The propositions characterizing Skolem constants introduced during instantiation. More about introduction of Skolem constants is explained in the further examples, namely in Subsection 6.8;
- The list of additional may-equalities propagated by theory-specific decision procedures in the SMT solver. Since TSMT itself is only able to non-conservatively *approximate* may-congruence, the concrete congruence relation implied by the reconstructed counterexample model may contain additional equalities that are not part of the may-approximation. This well illustrates the fact the the may-approximation used by TSMT is non-conservative and that the E-matching implemented in a traditional way inside the solver could potentially provide more relevant instantiations by employing the congruence returned by the combination of other theories implemented in the solver, such as linear arithmetic. If such a theory-implied equality is needed to be accounted in the may-congruence to produce the necessary instantiations, it can be supplied to TSMT by using a dummy *hint* property instantiated with the additional equality. E. g. *hint*[*of* "$f\ a = 0$"].

6.3 tlki command

Now let's first ensure that our sample property was instantiated as expected. To do this we can use the **tlki** command that is applicable in a proof state resulting from unsuccessful application of the *tsmt* method. The command has the following syntax:



The command selects from all resulting instantiations of all properties the ones that contain occurrences of terms matching the patterns specified in the command. The command supports conjunctive and disjunctive constraints, but not negations (e. g. excluding instances that match some patterns is not supported). Using the command we can query instantiations containing e .g. the variable a. However, here we should also note that unlike HOL (a Higher-Order Logic), the logic internally used by TSMT is essentially first-order. This particularly implies that a function application e. g. $g\ a\ b$ cannot be further split into currified subterms such as g and $g\ a$. The command **tlki** looks for subterms, so a query P will not result in any matching instances, because P does not occur as a separate first-order term, but only as a function in a full (uncurrified) application of the form $P\ x$. However, if there are indeed actual higher-order occurrences of the predicate P such as *filter* P [], then only the instances containing those higher-order occurences would be recognized and matched by the **tlki** command. To look up any full application of the predicate P we can use a *wildcard* _ as in P _. The command also recognizes schematic variables such as *?a*, but does not perform non-linear matching i. e. the patterns _ = _ and *?a* = *?a* are treated similarly [3]. This is of course in contrast to TSMT itself that really supports arbitrary patterns, including non-linear ones. Let's summarize all of the above:

```
lemma "(a = b ∨ a = c) ∧ a = d ∧ u = f a ∧ f b < f d ∧ P u ⟹ b < 0"
  apply (tsmt sample_prop)
  tlki "?a < ?a"
  tlki "a" or "b"
  tlki "P _" and "_ < 0"
  tlki "a"
  oops
```

We can inspect the effects of various uses of the **tlki** command by looking at the output in the interactive document inside Isabelle/jEdit.

Moreover, looking at both the output of the **tlki** command and the *tsmt* proof method itself we can notice that in fact both of them highlight predicates and even display some values for subterms inside tooltips available on mouse hovering with `Ctrl` pressed and held. Those values are extracted from the SMT solver upon returning the counterexample model. Predicates that hold in the counterexample model are highlighted in green and the predicates that do not hold are highlighted in red. The colors are actually the ones used for Markdown list markup, so they are configurable in the Isabelle plugin options (`Markdown bullet1` for truth and `Markdown bullet4` for falsehood). The model is consistent with all property instantiations obtained by *tsmt*.

After looking closer at the provided counterexample model, we can notice that in fact in that model $b > 0$ and $c < 0$ (the precise numbers depend on the particular solver implementation). So we can now come up with the correct formulation of the lemma and finally prove it with *tsmt*:

```
lemma "(a = b ∨ a = c) ∧ a = d ∧ u = f a ∧ f b < f d ∧ P u ⟹ b < 0 ∨ c < 0"
  by (tsmt sample_prop)
```

[3] *Linear* patterns contain no more than a single occurrence of any schematic variable, while *non-linear* patterns can generally contain any number of occurrences of the same variable.

Let's now move on to a more realistic example: Let's assume we are verifying some program manipulating fixed-size machine integers. Take their size to be *64* bits. First, let's introduce abbreviation for an auxiliary constant:

abbreviation *"MAX_ULONG ≡ 18446744073709551615"*

Now let's assume we want to prove the following useful property of integer division:

lemma *"a < MAX_ULONG div 2 ⟹ uint (a div 2 * 2) ≤ uint (a * 2 div 2)"* **for** *a b :: "64 word"*
oops

First let's try to prove it using existing capabilities provided by the `HOL-Word` library. We try to prove the lemma with *uint_arith* method for arithmetic on bounded integers and then using Sledgehammer.

lemma *"a < MAX_ULONG div 2 ⟹ uint (a div 2 * 2) ≤ uint (a * 2 div 2)"* **for** *a b :: "64 word"*
apply *uint_arith*
oops
lemma *"a < MAX_ULONG div 2 ⟹ uint (a div 2 * 2) ≤ uint (a * 2 div 2)"* **for** *a b :: "64 word"*
sledgehammer [*timeout=10*, *provers="cvc4 z3"*]
oops

Both attempts are unsuccessful. So now we switch to the TSMT-based supplementary theory *TSMT_Tutorial.TSMT_Bounded* providing a locale with a relatively complete set of properties for deciding satisfiability problems for ground formulas in the theory of bounded integers using existing SMT solvers. The implementation currently provided by the *ubound* locale is compete for the following operations on bounded integers: $+$, $-$, $*$ (linear multiplication by a constant),*div*, *mod*, $+$, *uint* (conversion from bounded integer to a mathematical integer), and *word_of_int* (the reverse conversion, generally has wrap-around semantics). To use the locale we instantiate it with the particular bounded integer type of interest and provide the required constant value:

type_synonym *ulong = "64 word"*
interpretation *ubound "TYPE(64)" MAX_ULONG* **by** *unfold_locales simp_all*

Now let's consider some of the properties provided by the locale. The first and simplest property is just another notation for the upper bound of our bounded integer type (`ulong`):

thm *U*

when : $U = MAX\_ULONG$

The next property formalizes the notion of integer division and remainder. Note that whenever we encounter either of the corresponding operators (functions) *div* or *mod*, we want to instantiate both of the two properties unambiguously characterizing the integer division with remainder, so we need two sets of triggers, one for the division and one for the remainder. To achieve this we use the **or** keyword in the *when* attribute:

thm *div_modT*

when *a div numeral k* **or** *a mod numeral k*:
$numeral\ k \leq U \longrightarrow$
$uint\ (a\ div\ numeral\ k) * numeral\ k + uint\ (a\ mod\ numeral\ k) = uint\ a$
when *a div numeral k* **or** *a mod numeral k*:
$0 < numeral\ k \longrightarrow numeral\ k \leq U \longrightarrow uint\ (a\ mod\ numeral\ k) < numeral\ k$

We also note that we only support integer division by a *syntactic constant*. The non-linear case of division/multiplication by a variable is not fully supported by the solvers and even in some special cases, when the variable is propagated with a constant value by congruence or linear arithmetic, although the solver can be able to prove unsatisfiability, the proof may still fail to replay within Isabelle/HOL (because Isabelle's tactics for linear arithmetic are less flexible than those of SMT solvers). Therefore we restrict our reasoning to syntactic constants that are represented in Isabelle using the *numeral* function that takes binary representation of the number as argument. Other notations, such as decimal, octal an hexadecimal numbers are

implemented in Isabelle as parse/print translations. For completeness sake we notice that the numbers *0* and *1* may also be represented in Isabelle using special polymorphic constants $0::'a$ and $1::'a$ and negative numbers do not have special constant notation, so a number -2 is not recognized as constant, but as an application of a prefix function $(-)$ to the constant *2*. Those details are also accounted for in the *ubound* locale.

The next property formalizes multiplication of a bounded integer by a constant:

thm *lin_mul'T*

when $a * numeral\ k$**:** $uint\ a * numeral\ k \leq U \longrightarrow uint\ (a * numeral\ k) = uint\ a * numeral\ k$

Actually, there are two such properties, one for multiplication with constant on the left (*lin_mulT*) and one for the constant on the right (this *lin_mul'T*). Also note the convention to give TSMT properties names ending with *T*. This is just a convention introduced by analogy to Isabelle's *I* for introduction, *E* for elimination rules etc. The next property formalizes semantics of bounded integer constants:

thm *numeralT*

when $numeral\ k$**:** $numeral\ k \leq U \longrightarrow uint\ (numeral\ k) = numeral\ k$

The remaining properties straightforwardly formalize the semantics of comparison $(\leq)$ between bounded integers and the properties of the conversion of a bounded integer to a mathematical integer (*int*):

thm *ltT*

when $a < b$**:** $(a < b) = (uint\ a < uint\ b)$

thm *uintT*

when $uint\ a$**:** $0 \leq uint\ a$
when $uint\ a$**:** $uint\ a \leq U$

Using the properties presented above we can immediately prove our target lemma:

```
lemma "a < MAX_ULONG div 2 ⟹ uint (a div 2 * 2) ≤ uint (a * 2 div 2)" for a b :: ulong
  by (tsmt div_modT lin_mul'T numeralT ltT uintT U)
```

Now let's test the capabilities of counterexample model extraction using this sample lemma. Pretend we have the following wrong formulation of the lemma:

$$a < MAX\_ULONG \Longrightarrow uint\ (a\ div\ 2 * 2) \leq uint\ (a * 2\ div\ 2).$$

Upon invoking the *tsmt* tactic with the same arguments as above for this modified lemma, we get the usual counterexample report:

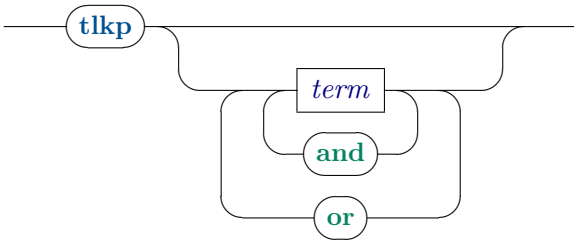
```
lemma "a < MAX_ULONG ⟹ uint (a div 2 * 2) ≤ uint (a * 2 div 2)" for a b :: ulong
  apply (tsmt div_modT lin_mul'T numeralT ltT uintT U) — Inspect in isabelle/jEdit
  oops
```

However, upon closer inspection of the counterexample model we notice that it does not provide the desired values for the variable *a* that satisfy the negated formula. Instead, we just see that the value of the term *a* in the counterexample model is just itself, i .e. *a*. This is what TSMT produces by default when it is not able to directly reconstruct the model of a term. The reason for this failure lies in the use of particular underlying theories in the back-end SMT solver, more particularly, the QF_UFLIA logic that is currently picked by TSMT for all formulas regardless of their syntax or semantics. TSMT favours QF_UFLIA because of its full support provided both by major SMT solvers (primarily, Z3) and Isabelle proof reconstruction implementation. In practice this most surely guarantees that any proof produced by the SMT solver will be definitely replayed by the Isabelle inference kernel. Yet QF_UFLIA only meaningfully interprets mathematical integers and linear arithmetic operations on them as well as predicates. So an SMT solver implementing a complete decision procedure for this logic will only produce concrete models for terms of types *int* and *bool*. Since the variable *a* is of type *ulong* i. e. is not a mathematical integer, it does not have an explicit model, hence the dummy value. However, the fact that TSMT preforms instantiation

ahead of satisfiability check provides one crucial advantage: together with instantiation it produces a lot of auxiliary terms (the subterms of the obtained instances) that complement the terms directly present in the initial formula. These terms can be later employed to enhance the capabilities of the counterexample model reconstruction.

6.4 tlkp command

More specifically, TSMT requests the model for all terms occurring in the resulting instantiations as well as in the initial formula. Those models are accessible with the **tlkp** command, which has the following syntax



entirely analogous to that of the **tlki** command. Unlike **tlki** though, it searches not for instantiations, but for subterms occurring in them. The found matching subterms are printed in alphabetical order along with the values returned by the solver. Here it's important to note that for terms of types not interpreted in the QF_UFLIA logic, the solver returns Skolem constants instead of real values as it does not ascribe them any particular interpretation. However, since QF_UFLIA includes congruence closure, those constants still provide some useful information about equalities between those values. Equal terms are assigned the same Skolem constant. Since the constants returned by the solver are given quite arbitrary meaningless names, TSMT replaces them with the shortest (least-size) representants from their corresponding equivalence classes and shows us those representants instead. Therefore, not only the variable a itself, but also all other terms equal to it in the produced counterexample will be assigned the default value a in the resulting model. However, in our particular example it's more important that our properties actually relate the bounded integer numbers to the corresponding mathematical integers using the functions *uint* and *word_of_int*. Since those functions occur in the resulting instantiations and even more so, our set of properties is designed in a way that a term of the form *uint x* will inevitably appear in the instantiations for *any* bounded integer term x occurring in the initial formula, we can be sure that any such term is related to the corresponding integer term *uint x* by our property instances. Thus to obtain some useful hint on the value of the term a, or any other bounded integer term occurring in the instantiated formula, we can request the value of the corresponding integer term, e. g. *uint a*. Let's do that:

```
lemma "a < MAX_ULONG ⟹ uint (a div 2 * 2) ≤ uint (a * 2 div 2)" for a b :: ulong
  apply (tsmt div_modT lin_mul'T numeralT ltT uintT U)
  tlkp "uint a" or "uint (a * 2)" or "uint (a div 2)"
  tlkp "a div 2 * 2" or "a * 2 div 2"
  oops
```

6.5 Custom model extractors

In this output we see the value assigned to the *uint a* is equal to *9223372036854775808*. Also, we can see that the term *uint* $(a * 2)$ is assigned an arbitrary value (it is *0* for my implementation) not equal to $2 *$ *uint a*, that reflects the fact that in our theory of bounded integers an overflow during integer multiplication results in an undefined value. This way we can ultimately detect the flaw in our second erroneous formulation. However, it would be more convenient if the *tsmt* method itself could automatically perform some transformations of the extracted counterexample model to present us more helpful and readable output. This is nontheless not entirely trivial since TSMT does not have an obvious way to know about all the possible theories (such as bounded integer arithmetic) we might want to reason about in advance. For this reason we decided to make the model extraction capabilities maximally flexible and equip the users with

some basic means to define and plug in their own model transformations. The particular functions are defined in the ML structure `TSMT_Model` and are not covered in this tutorial as this requires some additional introduction to Isabelle/ML programming and some common practices of pure functional programming. We suggest more advanced users to figure out the semantics of basic model extraction combinators directly from their definitions as they are all relatively small. Here we only present the particular code for registering the extractors for values of type *'a word* (of which our type *ulong* is an instance):

setup ‹*TSMT_Model.add_extractor* ***type_name***‹*word*› *"reconstruct"* (*K extract_word_reconstruct*)›
setup ‹*TSMT_Model.add_extractor* ***type_name***‹*word*› *"lookup"* (*K extract_word_lookup*)›

The first extractor implements reconstruction of the bounded integer value x based on the value of the corresponding term of the form *uint x* while the second extractor looks for the terms of the form *word_of_int y* in the equivalence class of the term x. The extractors are applied in the *reverse* order relative to the order of their registration, and the value returned by the first successful extractor is selected, so the extractor based on *word_of_int y* is preferred in out setting. After registering the extractors, we can finally get the nice reconstructed model we might originally hoped for:

lemma *"**a** < **MAX_ULONG** ⟹ **uint** (**a** **div** **2** * **2**) ≤ **uint** (**a** * **2** **div** **2**)"* **for** *a b* :: *ulong*
  apply (*tsmt div_modT lin_mul'T numeralT ltT uintT U*) — Try it out in Isabelle/jEdit!
  tlkp *"**a** **div** **2** * **2**"* **or** *"**a** * **2** **div** **2**"*
  oops

6.6 *tsmt add/del* attribute

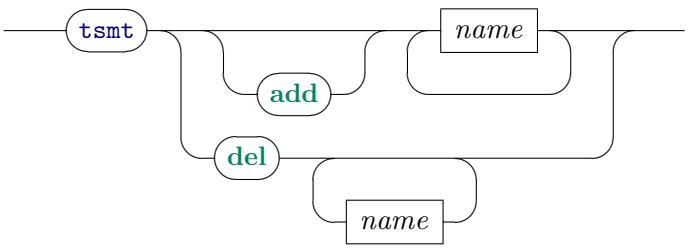
There is, however, an inherent inconvenience in the need to provide the sequence of instantiated properties explicitly at every invocation of the *tsmt* method. This can be addressed by first defining a named fact comprising the carefully designed set of properties e. g.:

lemmas *bounded* = *div_modT lin_mul'T numeralT ltT uintT U*
lemma *"**a** < **MAX_ULONG** **div** **2** ⟹ **uint** (**a** **div** **2** * **2**) ≤ **uint** (**a** * **2** **div** **2**)"* **for** *a b* :: *ulong*
  by (*tsmt bounded*)

Nonetheless, this approach has at least two disadvantages:

- First, it's impossible to extend the previously defined fact afterwards, when some additional properties become relevant for further proofs. This is especially inconvenient in the context of incremental property development, where some previously established properties are applied to prove the later, derived and often more complicated ones.
- Second, there is a small somewhat nasty detail about the Isabell's treatment of facts defined using the **lemma** command: The original names of the theorems comprising the fact are not preserved. This prevents some nice features from being based on such fact bundles, for instance name-annotated statistics output for unsuccessful proof attempts as well as automated proof optimization with filtering of relevant theorems, while reporting their original names in the output.

For this reason TSMT toolset provides a special set of attributes with the following syntax:



The form *add* $name_1$ ... $name_n$ adds the theorem being modified by the attribute to the named property groups with the specified names. If some of the groups don't previously exist, it also creates them. The *add* token can be omitted, so by default the *tsmt* attribute is treated similarly to *tsmt add*. The *del* form

without name deletes the theorem being modified from all previously created property groups, while the form *del* $name_1$ ... $name_n$ deletes the theorem only from the specified groups. The order of the properties in the group is the order in which they are added to the group.

Let's now consider one another aspect of defining and using the properties to prove propositions with *tsmt*. First let's assume we defined the following theorem group using the *tsmt* attribute:

lemmas [*tsmt bounded'*] = *word_of_intT uintT*(*2*)

Later we proved and extended the property group with the following additional lemmas:

lemmas [*tsmt bounded'*] =
leT subT addT zero one numeralT lin_mulT lin_mul'T lin_mul0T lin_mul0'T lin_mul1T lin_mul1'T U

Now we want to prove a sample property about bounded integers with those properties. The proposition is well-understood and is a typical minimal example of a property of bounded integers from the decidable fragment, where the set of properties defined above is complete. However, upon trying to apply the *tsmt* method we get an unexpected result:

lemma *simple_example*:
"*MAX_ULONG* ≤ *a* ⟹ *1* ∗ *a* − *MAX_ULONG* = *0*" for *a* :: *ulong*
apply (*tsmt bounded'*)
tlkp "*uint* _" tlkp "*word_of_int* (*uint* _)"
oops

Let's inspect what has just happened in some more detail. Our intent in approaching the proof was to establish the bijection between the bounded terms occurring in the formula and their mathematical unbounded counterparts. A crucial part in establishing this bijeciton is instantiation of the property *word_of_intT* (**when** *uint a*: *word_of_int* (*uint a*) = *a*) for every unbounded integer counterpart of the form *uint x*. However, to successfully perform this instantiation as intended we need to have all the necessary terms of this form to occur in the current formula. However, if we instantiate the property *word_of_intT* *before* instantiating the properties that establish the correspondence for all the operations occurring in the formula, we might not yet have all the necessary terms. In our example the initial formula does not contain any subterm of the form *uint x* at all, all of them should be added by instantiating the preceding properties. So the property *word_of_intT* should be placed *after* the properties characterizing the operations (*subT*, *addT* etc.). Thus the reason of this failure is the wrong ordering of properties in our group. For this reason, the *TSMT_Tutorial.TSMT_Bounded* theory defines the correctly-ordered group *ubound*. However, if we want to care somewhat less about the order of property instantiations we might also use an iterated instantiation group as follows:

lemma *simple_example*: **"*MAX_ULONG* ≤ *a* ⟹ *1* ∗ *a* − *MAX_ULONG* = *0*" for** *a* :: *ulong*
by (*tsmt* (*bounded'*)) — Note the parenthesis

Here we should warn that as TSMT does *not enforce* any termination criterion on such groups, and thus can potentially loop. Therefore, carefully crafted (and, particularly, well-ordered) property groups are more favourable in general.

Now we move to some different kind of example lemmas, namely the theory of interpreted sets to demonstrate other important features provided by TSMT. The theory of interpreted sets, presented, for instance, in [3] includes the following most basic operations on sets: ∪, ∩, − (the set difference as it is denoted in Isabelle/HOL), {·} (the singleton set), as well as the constant {} (an empty set), and a predicate ∈. The paper actually presents a complete set of properties that can be used to obtain the corresponding decision procedure for the formulas in the theory using the tools provided by Isabelle and TSMT. Let's present here most of the resulting TSMT properties that we ended up with after optimizing somewhat the set of rules presented in the paper for the context of trigger-based quantifier instantiation:

lemma *emptyT*[**when "*u*" and "*{}* :: *'a set*"**]: **"*u* ∉ *{}*" by** *simp*
lemma *member_splitT*[**when "*t*" and "*u*"**]: **"*u* ∈ *t* ∨ *u* ∉ *t*" by** *simp*
lemma *UnT*[**when "*u* ∈ *s* ∪ *t*"**]: **"*u* ∈ *s* ∪ *t* ⟷ *u* ∈ *s* ∨ *u* ∈ *t*" using** *Un_iff* **.**
lemma *IntT*[**when "*u* ∈ *s* ∩ *t*"**]: **"*u* ∈ *s* ∩ *t* ⟷ *u* ∈ *s* ∧ *u* ∈ *t*" using** *Int_iff* **.**
lemma *DiffT*[**when "*u* ∈ *s* − *t*"**]: **"*u* ∈ *s* − *t* ⟷ *u* ∈ *s* ∧ *u* ∉ *t*" using** *Diff_iff* **.**
lemma *singletonT*[**when "*u* ∈ {*v*}"**]: **"*u* ∈ {*v*} ⟷ *u* = *v*" using** *singleton_iff* **.**

6.7 Skolemization of existential quantifiers

The presented properties are mostly just typical TSMT properties with triggers that do not demonstrate any especially interesting proof patterns. The only notable thing here is the use of variable triggers e. .g *"u"*, which correspond to instantiation of the property with every term of the corresponding type (e. g. a set or an element of a set) occurring in the current formula (the set of terms available after instantiating the preceding properties). However, one last property presented in the paper requires some more attention. We can represent it in HOL as follows:

$$s \neq t \Longrightarrow \exists w.\ w \in s \wedge w \notin t \vee w \notin s \wedge w \in t.$$

Here we have a problem: a nested existential quantifier that is not supported by TSMT. Although nested quantifiers are not directly supported by TSMT, which currently only supports schematic quantification, from general mathematical logic we know that there's a process transforming any formula in first-order logic to a normal form without any nested quantifiers: It's prenex normalization followed by Skolemization. The precise rules of prenex normlization are widely known and can be found in many general sources. Same is true for Skolemization, so here we only mention that as in our case we actually already have a prenex normal form with two implicit universal quantifiers $\forall s$ and $\forall t$, to eliminate the existentially quantified variable w we introduce a function of two variables (say, s and t) that would return an arbitrary element satisfying the body of the eliminated existential quantifier. Let's denote the Skolem function w_e and use the Hilbert epsilon operator (denoted as *SOME* _. _ in Isabelle/HOL) to obtain the required arbitrary value:

```
definition "we s t ≡ SOME x. x ∈ s − t ∨ x ∈ t − s"
```

Now we are ready to formulate and prove the final property from the minimal theory of interpreted sets:

```
lemma set_eqT[when "s = t", tsmt sets]:
  "s ≠ t ⟹ we s t ∈ s ∧ we s t ∉ t ∨ we s t ∉ s ∧ we s t ∈ t"
  unfolding we_def Diff_iff by auto (smt someI)+
lemmas [tsmt sets] = member_splitT emptyT UnT IntT DiffT singletonT
```

We note that our choice of the name for the Skolem function w_e is not entirely arbitrary. The convention of starting Skolem functions with the symbol w is currently relied upon in TSMT when inventing the names of the corresponding Skolem variables occurring in the automatically constructed optimized proofs, that are presented and discussed further in this tutorial (Subsection 6.11). For now, let's use our resulting property group *sets* to prove two example propositions from the paper:

```
lemma "x ∈ {1 :: int} ∪ {2} ⟹ x ≤ 2" by (tsmt sets)
lemma "x ∈ {1 :: int} ∪ {2} ⟹ x ≤ 2" by (tsmt (sets))
lemma "4 < u ⟹ u < 6 ⟹ u ∈ x ∪ y ⟹ x ∩ {5 :: int} ≠ {5} ⟹ z = {} ⟹ y − z ≠ {}"
  by (tsmt sets)
lemma "4 < u ⟹ u < 6 ⟹ u ∈ x ∪ y ⟹ x ∩ {5 :: int} ≠ {5} ⟹ z = {} ⟹ y − z ≠ {}"
  by (tsmt (sets))
```

Here we once again emphasize the need for careful ordering of the instantiated properties (or the use of iterated instantiation, which terminates for this property system as is shown in the paper). We also note that we can install the counterexample model extractor for the theory of interpreted sets using our pre-defined function provided in the `TSMT_Model` ML structure:

```
setup ‹TSMT_Model.add_extractor type_name‹set› "fold" TSMT_Model.set_extractor›
```

Here's an example use:

```
lemma "3 < u ⟹ u < 6 ⟹ u ∈ x ∪ y ⟹ x ∩ {5 :: int} ≠ {5} ⟹ z = {} ⟹ y − z ≠ {}"
  apply (tsmt sets) — Inspect interactively in Isabelle/jEdit
  oops
```

The extractor is based on exhaustively probing the model value for every predicate of the form $\_ \in s$ (for a set s) occurring in the resulting instantiated formula. The terms t such that the predicate $t \in s$ holds in the counterexample model are included in the reconstructed model for the set s.

To elaborate a bit more on the use of Skolem functions, let's consider a slightly more complicated proposition:

```
typedef object = "UNIV :: nat set" morphisms to_nat object by simp
typedef container = "UNIV :: nat set" morphisms to_nat container by simp
typedef name = "UNIV :: nat set" morphisms to_nat name by simp
datatype entity =
  Object (object : object) |
  Container (container : container)
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
  oops
```

We have several ways to prove this proposition with TSMT. First, if we attempt to prove it using Sledgehammer, we obtain an efficient proof:

```
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
  by (smt image_mono sup_mono)
```

If we want to replay this proof within TSMT, we need to augment the properties *image_mono* and *sup_mono* with explicit triggers as currently TSMT does not implement automatic trigger inference:

```
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
  by
    (tsmt
      image_mono[when "?A ⊆ ?B" and "?f ` ?A"]
      sup_mono[where ?'a="'a set", when "?a ⊆ ?c" and "?b ⊆ ?d"])
```

Even though in this example we were able to harness existing properties such as *image_mono*, it may be more reliable to ensure we have a relatively complete set of properties to reason about certain classes of propositions, because the suitable auxiliary lemmas such as *image_mono* are not guaranteed to be present in Isabelle libraries for every proposition we might need to prove. For the *image* operation (`) we suggest the following set of properties:

```
definition "w_i x f s ≡ SOME y. y ∈ s ∧ x = f y"
lemma imageT[when "x ∈ f ` s", tsmt sets]: "x ∈ f ` s ⟹ w_i x f s ∈ s ∧ x = f (w_i x f s)"
  unfolding w_i_def by auto (smt someI)+
lemma imageIT[when "y ∈ s" and "f y" and "f ` s", tsmt sets]:
  "y ∈ s ⟹ f y ∈ f ` s"
  using imageI .
```

We also add two properties for the (⊂) operation (⊆):

```
definition "w_s s t ≡ SOME x. x ∈ s - t"
lemma subsetIT[when "s ⊆ t", tsmt sets]:
  "¬ s ⊆ t ⟹ w_s s t ∈ s ∧ w_s s t ∉ t"
  unfolding w_s_def subset_iff by auto (smt someI)+
lemma subsetT[when "s ⊆ t" and "x ∈ s", tsmt sets]:
  "s ⊆ t ⟹ x ∈ s ⟹ x ∈ t"
  using set_mp .
```

We already notice how two of the four additional properties we defined involve Skolem functions. And indeed, when we try to gradually prove our lemma by iteratively inspecting the resulting counterexample

model and adding the suitable missing properties on each step, we can notice how the terms involved in the resulting instances are becoming larger and larger due to growing nesting of the Skolem function applications:

lemma
"entities = Object ` objects ∪ Container ` containers ⟹
entities' = Object ` objects' ∪ Container ` containers' ⟹
objects' ⊆ objects ⟹
containers' ⊆ containers ⟹
entities' ⊆ entities"
apply (*tsmt subsetIT UnT imageT subsetT*)
— Inspect the model in jEdit. Notice the model of *objects'* is {w_i (w_s *entities' entities*) *Object objects'*}, i. e. involves two allications of Skolem functions w_s and w_i
oops

6.8 Skolem variables and *tsmt skolem* attribute

To alleviate this problem TSMT provides the ability to introduce auxiliary *Skolem variables* during instantiation of properties involving Skolem functions. The variables simply abbreviate concrete applications of Skolem functions occurring in the resulting instantiations. Since the Skolem variables are introduced automatically, their names follow a particular naming convention. They all have names of the form $w_{lab\_n\_m}$, where n is a unique number identifying the property containing the corresponding application of the Skolem function w_{lab}, and the number m identifies the particular instance of this property (simply increasing index starting from *0*). To enable the introduction of Skolem variables for applications of a Skolem function, the definition of that function should be transformed with the *tsmt skolem* attribute:

declare w_s_def[*tsmt skolem*] w_i_def[*tsmt skolem*]

Now we can finish our proof of the example lemma:

lemma
"entities = Object ` objects ∪ Container ` containers ⟹
entities' = Object ` objects' ∪ Container ` containers' ⟹
objects' ⊆ objects ⟹
containers' ⊆ containers ⟹
entities' ⊆ entities"
apply (*tsmt subsetIT UnT imageT subsetT*)
— Notice the Skolem variable $w_{i\_7\_0}$ in the model of *objects'*
oops
lemma
"entities = Object ` objects ∪ Container ` containers ⟹
entities' = Object ` objects' ∪ Container ` containers' ⟹
objects' ⊆ objects ⟹
containers' ⊆ containers ⟹
entities' ⊆ entities"
by (*tsmt subsetIT UnT imageT subsetT imageIT*)

6.9 *tsmt def* and *tsmt size* attributes

A small note on the lemma we've just proved: Its premises contain an equality *entities = Object ` objects ∪ Container ` containers*. Since the equality is present in the original formulation on the lemma, it is part of the initial must-congruence and this in particular implies that size size of the term *entities* is considered to be not less than that of the term *Object ` objects ∪ Container ` containers*. For that reason a trigger $x \in s \cup t$ can match on the term $x \in$ *entities* modulo congruence. However, if we had the constants *objects*, *containers* and *entities* as parts of the context (e. g. in the context of a locale), we might want to ensure the size of the term *entities* is always considered to be at least the same as the size of the term *Object ` objects ∪ Container ` containers*. Although we could add the property *entities = Object ` objects ∪ Container ` containers* (with empty triggers) somewhere close the beginning of the instantiation group and follow the group in the *tsmt* instantiation sequence by the size margin re-computation (denoted as "!"), this would

also increase the size of all other terms in the formula. For a more targeted control of the size, TSMT has two special attributes: *tsmt def* that is applied to the definitions of the form $a \equiv b$ and "*tsmt size c n*" that is used directly and ensures the size of the constant c to be at least n. Both attributes can only *increase* the size of the constant. If the size is already set to a greater value, the attributes do nothing. We can apply the attributes in the following way:

```
locale sets =
  fixes objects :: "object set"
    and containers :: "container set"
begin
definition [tsmt def]: "entities ≡ Object ` objects ∪ Container ` containers"
```

or

```
declare [[tsmt size "entities" 7]]
```

After our successful attempt at careful interactive proof of our property with manual inspection of the produced counterexamples let's now attempt a more effortless approach to proving the lemma. Let's try to directly use our set of properties *sets*:

```
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
  apply (tsmt sets)
  oops
```

The first attempt failed. The second attempt at trying the iterated instantiation would also fail, because the instantiation would loop (we suggest the reader to figure out the precise reason for this). We can apply bounded instantiation deep enough to finally prove the lemma without elaborating the particular properties involved:

```
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
  by (tsmt 3×sets)
```

Yet as we can see this proof leads to a significant overhead stemming from excessive instantiation of irrelevant properties. We would like a way to somehow quickly identify the relevant properties automatically. Actually, often, there's such a way. So here we suggest another approach to significantly reduce the proof effort when using TSMT:

6.10 Applying sledgehammer in a failing TSMT proof state

In fact, as TSMT performs all instantiation internally and produces the set of resulting instances that are actually valid Isabelle theorems, it's possible to use these instances to help **sledgehammer** automatically identify the relevant properties to finish the proof. We suggest the following **sledgehammer** configuration for that purpose:

sledgehammer_params [*provers="cvc4 z3 spass e remote_vampire", dont_preplay, dont_try0, no_isar_proofs*]

The instantiations produced by TSMT are bound to the fact named *this* available in the context of a failed proof. Therefore we can apply **sledgehammer** in this state as follows:

```
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
```

```
  containers' ⊆ containers ⟹
  entities' ⊆ entities"
 apply (tsmt sets) using this sledgehammer [timeout=10] (sets)
 oops
```

sledgehammer helps figuring out the relevant properties, so we append them to the end of our instantiation sequence:

```
lemma
 "entities = Object ` objects ∪ Container ` containers ⟹
  entities' = Object ` objects' ∪ Container ` containers' ⟹
  objects' ⊆ objects ⟹
  containers' ⊆ containers ⟹
  entities' ⊆ entities"
 apply (tsmt sets sets(4,8,9,11)) oops
```

This does not lead us directly to a successful proof, but we can repeat the attempt:

```
lemma
 "entities = Object ` objects ∪ Container ` containers ⟹
  entities' = Object ` objects' ∪ Container ` containers' ⟹
  objects' ⊆ objects ⟹
  containers' ⊆ containers ⟹
  entities' ⊆ entities"
 apply (tsmt sets sets(4,8,9,11)) using this sledgehammer [timeout=10] (sets)
 oops

lemma
 "entities = Object ` objects ∪ Container ` containers ⟹
  entities' = Object ` objects' ∪ Container ` containers' ⟹
  objects' ⊆ objects ⟹
  containers' ⊆ containers ⟹
  entities' ⊆ entities"
 by (tsmt sets sets(4,8,9,11) sets(9))
```

We finally obtained a working and relatively efficient proof. However, in case of more complicated lemmas the resulting proofs obtained in this manner may be still not efficient enough, especially in a typical "interactive proof document" setting of the Isabelle/PIDE workflow that implies multiple repetitions of the same proof upon every slight change made to the active document. For that reason TSMT provides a special proof method that automatically optimizes the resulting successful proof attempt based on the capabilities of modern SMT solvers, namely on the extraction of *unsatisfiable cores*.

6.11 Optimizing successful proofs with *topt* method

The *topt* method accepts exactly the same syntax as the basic *tsmt* method. It also supports counterexample model extraction and binds the *this* fact exactly the same way, so it can actually be used in place of *tsmt* to obtain the initial successful proof by iterative refinement with **sledgehammer**. The difference is that it *does not* actually prove the goal in case the proof attempt is successful. Instead, it generates a structured Isabell/Isar proof of the goal that ends in an application of the *tsmt* method with an optimized property sequence:

```
lemma
 "entities = Object ` objects ∪ Container ` containers ⟹
  entities' = Object ` objects' ∪ Container ` containers' ⟹
  objects' ⊆ objects ⟹
  containers' ⊆ containers ⟹
  entities' ⊆ entities"
 apply (topt sets sets(4,8,9,11) sets(9))
 oops
```

The user can either use the entire structured optimized proof by clicking on the output of the *topt* method or only the last optimized application of *tsmt* by clicking on the last **by** command:

```
lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
— After clicking somewhere in the proof and removing the initial topt invocation
proof atomize?
  assume
    "entities = Object ` objects ∪ Container ` containers"
    "entities' = Object ` objects' ∪ Container ` containers'"
    "objects' ⊆ objects"
    "containers' ⊆ containers"
  thus "entities' ⊆ entities"
    by (tsmt subsetIT sets(4, 8, 11) imageIT)
qed

lemma
  "entities = Object ` objects ∪ Container ` containers ⟹
   entities' = Object ` objects' ∪ Container ` containers' ⟹
   objects' ⊆ objects ⟹
   containers' ⊆ containers ⟹
   entities' ⊆ entities"
— After clicking on the by command and removing the initial topt invocation
  by (tsmt subsetIT sets(4, 8, 11) imageIT)
```

6.12 *topt* configuration options

The proof optimization implemented by *topt* method can be configured by the following options that can be used to make the resulting proof either shorter, more efficient or more readable:

- *minimize_core*, *on* by default. This option enables additional *local* minimization of the unsatisfiable core returned by the SMT solver. Since the unsatisfiable core is inherently ambiguous as the formula can have several alternative proofs and the solver always relies only on the first one available that it happened to obtain, it can be reasonable to try to optimize the resulting core further by successively trying to exclude every assumption included in the core one by one. This additional optimization is enabled by this option. It should be noted that it might still not obtain the shortest possible proof, because an entirely different shorter proof can still exist that includes some other assumptions (instantiations) that were not initially included in the core.

- *max_insts*, equals *0* by default. The *topt* method is not only able to filter out some irrelevant properties, but also to replace some properties with their corresponding relevant instantiations altogether. This is especially relevant for proofs involving instantiations of many properties that can still be too slow even after restricting the property sequence only to the relevant ones. This option puts a limit on the number of relevant instances that can be used in place of the original property. As a special case this option can be used to avoid instantiation altogether and obtain an entirely ground proof of the goal e. g.
  using [[*max_insts=3*]] **apply** (*topt sets sets(4,8,9,11) sets(9)*)
  would give
  ```
  define w_s w_i w_i0 where
      "w_s ≡ w_s entities' entities"
      "w_i ≡ w_i w_s Container containers'"
      "w_i0 ≡ w_i w_s Object objects'"
  ultimately show "entities' ⊆ entities"
      by
          (tsmt
              subsetIT[of "entities'" "entities"]
  ```

UnT[*of* "w_s" "**Object ` objects′**" "**Container ` containers′**"]
UnT[*of* "w_s" "**Object ` objects**" "**Container ` containers**"]
imageT[*of* "w_s" "**Object**" "**objects′**"] *imageT*[*of* "w_s" "**Container**" "**containers′**"]
subsetT[*of* "**objects′**" "**objects**" "w_{i0}"]
subsetT[*of* "**containers′**" "**containers**" "w_i"] *imageIT*[*of* "w_{i0}" "**objects**" "**Object**"]
imageIT[*of* "w_i" "**containers**" "**Container**"])

Here it's important to note that this feature relies on the conventional naming of the Skolem functions starting with w that is replaced by w to avoid conflicts in the Skolem variable definitions.

- *merge_refs*, *on* by default. The option enables merging of adjacent properties from the same group e. g. *sets* into a single fact reference with the corresponding theorem selection e. g. *sets*(*4*,*8*,*11*). Turning this option off enables output of each property separately with the corresponding original name e. g. *UnT imageT subsetT* thus making the proof longer, but arguably a bit more understandable.

- *drop_whens*, *on* by default. This option disables splitting of the resulting properties by their disjunctive triggers. For instance if only instantiations produced by some of the disjunctive triggers are relevant, the resulting proof will contain a special trigger selection attribute *when* $n_1 \ldots n_k$ that selects only the specified triggers by their corresponding indexes. This attribute causes removal of these additional selection attributes, making the proofs shorter, but also potentially slower.

7 Replaying TSMT proofs with Metis

Since TSMT implements quantifier instantiation and thus transforms the original goals by supplementing them with additional implied facts, the resulting goals may often be discharged not only by SMT solvers, but also by other decision procedures, including the internal decision procedures implemented within Isabelle itself, such as Metis and Argo. While our current experience with Argo unfortunately uncovered a number of blocker bugs that prevent its use in too many cases, the Metis solver turned out to be able to efficiently discharge many of the resulting goals. For this reason, we implemented an additional *tmetis* proof method that is fully analogous to *tsmt* except that is applies Metis as the final proof step. To augment Metis with some very limited background knowledge about linear arithmetic, *tmetis* attempts to prove pairwise disequalities between all explicit numerical constants occurring in the resulting instantiated formula and adds those disequalities to the facts supplied to *metis*.

8 More examples and planned features

The features presented in this tutorial are sufficient to prove some practically relevant lemmas from the formalization of a small hierarchical storage model available in the file `../PicoMROSL.thy`. We also recommend the reader to get familiar with the Sledgehammer tool by referring to its official documentation. In future we plan to implement at least the following extensions to TSMT:

- Automatic prenex normalization, Skolem function introduction and trigger inference. This should enable the use of many existing lemmas and nested quantified propositions as TSMT properties without the need of explicit definition of Skolem functions and corresponding triggers for every property;

- Automated ordering and grouping of properties based on a conservative termination criterion. This should make possible a fully automated reconstruction of TSMT proofs from **sledgehammer** proofs with high success rate (i. e. not guaranteed to succeed in general due to differences between TSMT and quantifier instantiation algorithms used in the solvers).

end

References

[1] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[2] M. Moskal, J. undefinedopuszaundefinedski, and J. R. Kiniry. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.*, 198(2):19–35, May 2008.

[3] C. G. Zarba. *Combining Sets with Elements*, pages 762–782. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.