4. Quantifiers

Program Verification

ETH Zurich, Spring Semester 2017
Alexander J. Summers

Why Quantifiers?

- Quantifiers show up *everywhere*, especially in software verification
- Used in *user-provided assertions* (e.g. program specifications)

  $\forall$i:Int, $\forall$j:Int. $0 \leq i < j < \mathrm{length}(a) \Rightarrow \mathrm{lookup}(a,i) \leq \mathrm{lookup}(a,j)$

- Used to model *additional theories* (e.g. those not natively supported)

  $\forall$l:IntList, $\forall$i:Int. $\mathrm{head}(\mathrm{cons}(i,l))=i$

  $\forall s_1$:Set. s_2:Set. $\mathrm{card}(\mathrm{union}(s_1,s_2)) = \mathrm{card}(s_1) + \mathrm{card}(s_2) + \mathrm{card}(\mathrm{inter}(s_1,s_2))$

- Used (with uninterpreted functions) to model *memory*

  $\forall$f:Field. $\mathrm{select}(\mathrm{heap},\mathbf{x},f)=0$

  $\forall$a:Addr. $\forall$f:Field. $\neg a=x \Rightarrow \mathrm{select}(h_1,\mathbf{a},f)=\mathrm{select}(h_2,\mathbf{a},f)$

- The bad news: first-order logic (no theories) is only *semi-decidable*
  - we need theories; first-order logic with e.g. linear arithmetic is *undecidable*

Approaches for Handling Quantifiers

- Select quantifier instances: $\forall x{:}T.A \;\equiv\; A[t_1/x] \wedge A[t_2/x] \wedge \ldots$
- Some quantifier techniques focus on *model finding*
  - good for some specific first-order fragments
  - important if we are particularly interested in `sat` results (and their models)
- As with other SAT extensions, both *eager* and *lazy* approaches exist
  - We will cover Z3's main *lazy model-finding* approach *(MBQI)*
- A widely-used alternative technique is *E-matching*
  - uses syntactic cues (*triggers*) to add partial quantifier instantiations
  - requires introspection: *when* will a quantifier instantiation be *relevant*?
  - doesn't guarantee to generate models, but (potentially) `unsat` results
- The plan: 1. *eager approach*, 2. *lazy model-finding*, 3. *E-matching*

Recap: Eager, Lazy and Hybrid Theory Integration

- We have a (propositional) DPLL/CDCL search engine
- An *eager* approach to integrating a theory T
  - desugars all T-literals *in advance* (adding new propositional literals for DPLL)
  - e.g. lecture 2 (Ackermannization, finite quantifiers, bit-blasting)
- An *lazy* approach to integrating a theory T
  - uses *purification, propositional abstraction* on T-literals, runs DPLL search
  - possible models must be *checked* T-*consistent* (if not, DPLL must backtrack)
- Two main *hybrid* approaches we've seen so far:
  - theory interacts with DPLL search (*theory deduction*, *theory conflict clauses*)
  - eagerly add only *partial* theory information, omit the rest: terminate if `unsat`, otherwise potentially refine problem and retry (e.g. incremental bit-blasting)

First-Order Logic Equivalences

- Note: we will sometimes omit sorts from quantifiers when irrelevant

$$\forall x_1.\forall x_2.\ A \ \equiv\ \forall x_2.\forall x_1.\ A$$

$$\exists x_1.\exists x_2.\ A \ \equiv\ \exists x_2.\exists x_1.\ A$$

$\neg\forall x.A \ \equiv\ \exists x.\neg A$ and $\neg\exists x.A \ \equiv\ \forall x.\neg A$

$$\forall x.(A \wedge B) \ \equiv\ (\forall x.A) \wedge (\forall x.B)$$

$$\exists x.(A \vee B) \ \equiv\ (\exists x.A) \vee (\exists x.B)$$

If $x \notin \mathrm{FV}(A)$ then $\exists x.A \ \equiv\ A \ \equiv \forall x.A$ and
$\forall x.(A \vee B) \ \equiv\ A \vee (\forall x.B)$ and $\exists x.(A \wedge B) \ \equiv\ A \wedge (\exists x.B)$

Make sure that you're comfortable with understanding and using these

Skolemization

- Existential quantifiers can be eliminated by *Skolemization*
- Idea: $\exists x.A$ is equisatisfiable with $A[c/x]$ where c is a fresh constant
  - no matter what the sort is (note: we don't allow "empty" sorts – cf. slide 36)
- We can eliminate (only) existentials in this way
  - e.g. $\exists x{:}\text{Int}.(x>4 \wedge x<5)$ is rewritten to $c>4 \wedge c<5$ for a fresh constant c
- This can only be done for existentials in *positive positions*
  - e.g.$\neg\exists x{:}\text{Int}.(x>4 \wedge x\leq 5)$ (false) must not be rewritten to $c>4 \wedge c\leq 5$ (sat)
  - easiest: apply CNF transformation first; use $\neg\exists x$ / $\forall x\neg$, $\neg\forall x$ /$\exists x\neg$ dualities
- Even in positive positions, we need more for quantifier alternations:
  - e.g. $\forall x{:}\text{Int}.\exists y{:}\text{Int}.(y>x)$ (true) would be rewritten to $\forall x{:}\text{Int}.(c>x)$ (unsat)
  - In general, Skolemization replaces an $\exists$-bound variable with a fresh *function* of the $\forall$*-bound variables enclosing it*: e.g. $\forall x{:}\text{Int}.(f(x)>x)$ (satisfiable)

Eager Quantifier Elimination

- For *some logical fragments*, quantifiers can be eliminated *eagerly*
- e.g. *Effectively PRopositional logic (EPR)* allows only formulas:
  - $\exists x_1 .\dots \exists x_m .\ \forall y_1 .\dots \forall y_n .\ A$ where A is quantifier-free, ($m,n \geq 0$)
  - and which contain no function symbols except *constants and equality*
- We can apply *Skolemization* to remove the existential quantifiers
  - note: this will (finitely) expand the set of constant symbols in the formula
- It is then sufficient to instantiate the $\forall$ quantifiers *for each constant*
  - i.e. replace $\forall x{:}T.A$ with $A[c_1/x] \wedge A[c_2/x] \wedge \dots$
- The resulting formula is equi-satisfiable and quantifier-free
  - Any models will include the introduced *Skolem constants* (can be removed)
- Approach is similar to *eager SMT*. What is the analogous *lazy* idea?

Quantified Literals

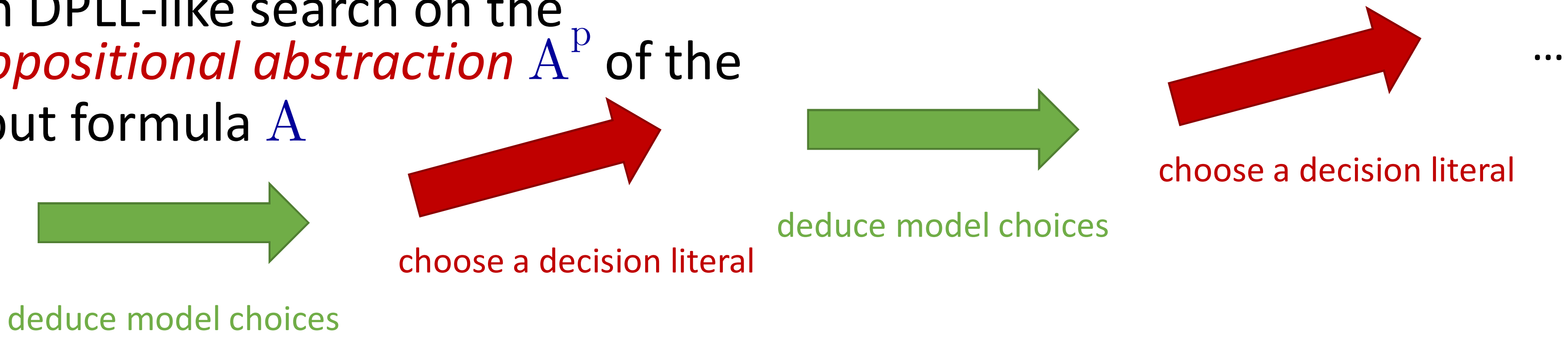
- Let's imagine *quantifiers as a theory*: consider lazy theory integration
- A *quantified literal* is a formula $\forall x{:}T.A$ or its negation $\neg\forall x{:}T.A$
  - note: $\neg\forall x{:}T.A \equiv \exists x{:}T.\neg A$
- An *extended clause* is a disjunction of (any number of) *first-order literals* (slide 62) and (any number of) *quantified literals*
- A formula is in *extended CNF* iff it is a conjunction of *extended clauses*
  - we can rewrite negative quantified literals via *Skolemization*
  - e.g. $(\forall x{:}T.p(x)) \Rightarrow \forall y{:}T.q(y)$ becomes $(\neg\forall x{:}T.p(x)) \vee \forall y{:}T.q(y)$
  - equivalently $(\exists x{:}T.\neg p(x)) \vee \forall y{:}T.q(y)$, Skolemization: $\neg p(c) \vee \forall y{:}T.q(y)$
  - we obtain a formula with *only positive quantified literals*
- Extend *propositional abstraction* to also abstract quantified literals
  - e.g. above becomes $\neg a \vee b$ where a,b abstract $p(c)$,$\forall y{:}T.q(y)$ respectively

Checking $\forall$-Quantifiers in a Candidate Model

- Given a candidate model M (only non-quantified literals), suppose want to check whether it also satisfies $\forall \mathrm{x}{:}\mathrm{T}.\mathrm{A}$ (for quantifier-free A)
- The formula is true in M exactly when $\exists \mathrm{x}{:}\mathrm{T}.\neg\mathrm{A}$ is *unsatisfiable in* M
- Via Skolemization, we reduce this to: $\neg\mathrm{A}[\mathrm{c}/\mathrm{x}]$ *unsatisfiable in* M
  - we can ask suitable theory solver(s) to check consistent (if so, give a model)
- If unsatisfiable, we know M satisfies the quantified formula
- If satisfiable, we get some value v such that $\mathrm{M}[\mathrm{c}{\mapsto}\mathrm{v}]{\vDash}\neg\mathrm{A}[\mathrm{c}/\mathrm{x}]$
- Idea: adding the constraint $\mathrm{A}[\mathrm{v}/\mathrm{x}]$ to our problem rules out model M
  - v is not a term: we pick term(s) t equal to v in the current candidate model
  - Ideally an existing term, but can be an interpreted constant / newly added
  - $\mathrm{A}[\mathrm{t}/\mathrm{x}]$ is *false in model* M (we don't fix *how* to select suitable term(s) t)

Model-Based Quantifier Instantiation (MBQI)

run DPLL-like search on the *propositional abstraction* A^p of the input formula A



- when a quantified literal is added to the candidate model:
  - *record the quantifier* for later checking (note: it must be positive $\forall x{:}T.A$)
- when a candidate model M is found:
  - is it a model for *all of the recorded quantifiers*? Check them (cf. last slide)
  - If all true, we are done. Otherwise, generate formula(s) $A[t/x]$ corresponding to a quantifier instantiation that is false in M
  - restart entire algorithm, conjoining $A[t/x]$ to the input formula
  - note: term t may or may not have been present in the original formula

MBQI Example

- Consider $f(b)=a+1 \wedge (\forall x{:}\text{Int}.f(x)<b) \wedge \forall x{:}\text{Int}.(x=a \vee f(x)>a+1)$
  Is the formula satisfiable? Try MBQI on the example:
- Find a candidate model M for non-quantified literal $f(b)=a+1$
  - e.g. $M(a)=0,\ M(b)=0,\ M(f)=(\lambda z.1)$ (Z3 initially guesses constant functions)
- Check quantified literals:
  - is $\neg(f(c)<b)$ satisfiable in M? *Yes: e.g.* if c gets value 0 (note: $M(a)=0$)
  - Conjoin e.g. $f(a)<b$ with the original problem, and try again
- e.g. new candidate model $M(a)=0,\ M(b)=2,\ M(f)=(\lambda z.1)$
  - is $\neg(f(c)<b)$ satisfiable in M? *No: first quantifier is true in the model*
  - is $\neg(c=a) \wedge \neg(f(c)>a+1)$ satisfiable in M? *Yes:* e.g. for $M(c)=M(b)$
  - conjoin $b=a \vee f(b)>a+1$ with the original problem, and try again
  - we now get `unsat`

MBQI (Non-)example

- Consider $\forall x{:}\text{Int}.\ f(x)>f(x\text{-}1)$
- Is the formula satisfiable? *Yes*
  - but we won't find the function definition by enumerating (counter-)examples
- e.g. a candidate model $M(f)=(\lambda z.0)$ doesn't work: $\neg f(1)>f(0)$
  - Conjoin e.g. $f(1)>f(0)$ with the original problem, and try again
  - Candidate model $M(f)=(\lambda z.(z=1?1{:}0))$ doesn't work: $\neg f(2)>f(1)$
  - Candidate model $M(f)=(\lambda z.(z=2?2{:}(z=1?1{:}0)))$ doesn't work: $\neg f(3)>f(2)$
  - … this continues forever (depending on the model-guessing approach)

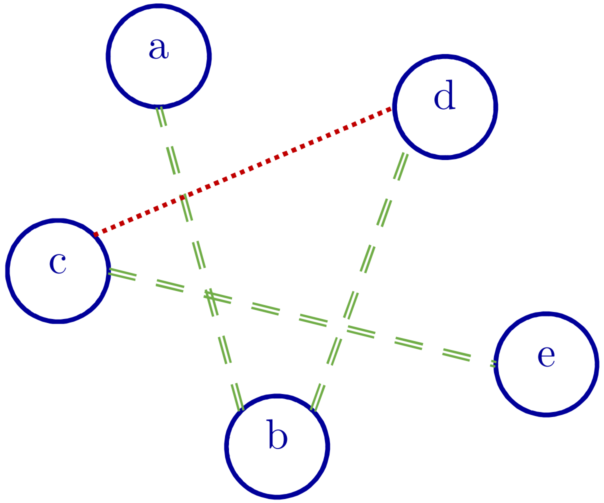
MBQI - Summary

- *Model-Based Quantifier Instantiation* can generate models
  - similar to a lazy SMT approach to integrating a theory
  - for certain decidable fragments, it provides termination guarantees
  - lazy approach may be faster in some cases than an eager approach
  - can also be applied outside of these fragments, at risk of non-termination
- MBQI may explore an *infinite space* of possible instantiations/models
  - because the exploration is *lazy* we may even find answers in an infinite space
  - but termination is guaranteed only for some decidable fragments (e.g. EPR)
  - a satisfiable first-order problem may sometimes have no finite model
  - this is common for program verification problems (e.g. recursive definitions)
- For general program verification, we'll need an alternative approach
  - we'll look at the most widely-used approach: *E-matching*

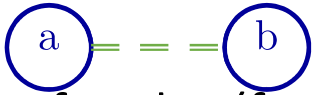
Representing Equalities and Disequalities

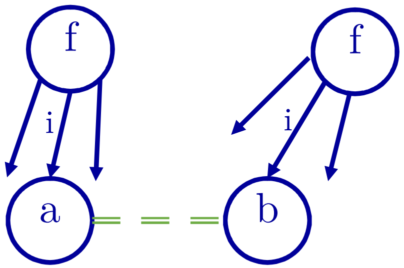
- Recall: SMT solver must maintain *(dis)equality information*
- Over constants (only), we can represent this using:
  - equality *equivalence classes*
  - tracking *disequal* pairs
  - The former can be implemented e.g. via *union-find* data structure
- Theory solver for equality/constants
  - model consistent iff no pair of unequal terms is in same equivalence class

e.g. $a=b,\ d=b,\ \neg(d=c),\ c=e$



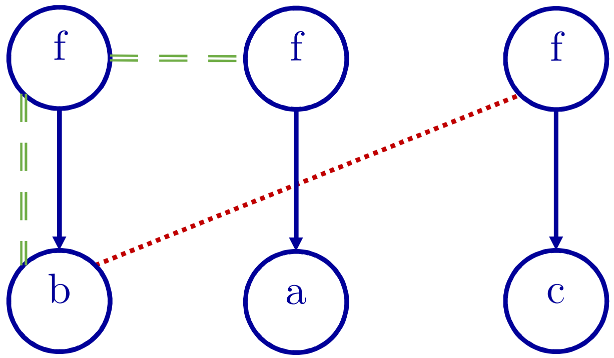
Congruence Closure: The E-Graph

- An E-graph is a generalisation of this idea, adding *uninterpreted functions*
- Node labelled f for each term $\mathrm{f}(\ldots)$
  - directed, indexed edges to each function argument  (omitted where clear)
  - equality and disequality edges as before
- On adding  equality edge:
  - find pairs of nodes (for each function f):



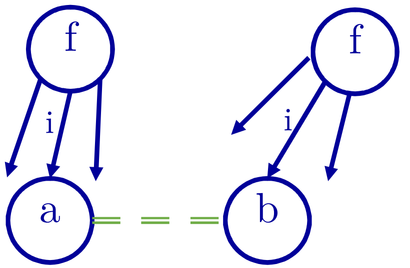
- if arguments of the two f nodes are *pairwise equal*, equate them too

e.g. $\mathrm{f(b)=b,\ f(b)=f(a),\ \neg(f(c)=b)}$



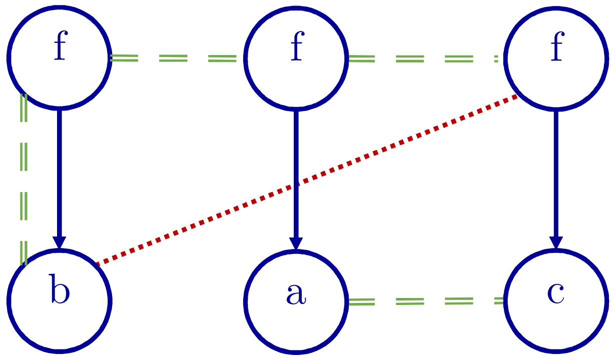
Congruence Closure: The E-Graph

- An E-graph is a generalisation of this idea, adding *uninterpreted functions*
- Node labelled f for each term $\mathrm{f}(\ldots)$
  - directed, indexed edges to each function argument $\xrightarrow{1}$ (omitted where clear)
  - equality and disequality edges as before
- On adding (a) = = (b) equality edge
  - find pairs of nodes (for each function f):



  - if arguments of the two f nodes are *pairwise equal*, equate them too
  - An efficient way to track (dis)equalities, and a built-in a theory solver for $\mathrm{T_E}$

e.g. $\mathrm{f(b)=b,\ f(b)=f(a),\ \neg(f(c)=b),\ a=c}$



Back to the Quantifiers: Introducing Triggers

- We extend the syntax of $\forall$-quantified formulas in two ways
  - Firstly, we allow multiple adjacent $\forall$-quantifiers to be *merged* into one:
  - A formula $\forall x_1.(\forall x_2.\ A)$ can be written $\forall x_1,x_2.\ A$
  - We refer to x_1, x_2 as the *variables quantified* by the (single) $\forall$-quantifier
- Now, we allow a *trigger* to be attached to any $\forall$-quantifier
  - we write e.g. $\forall x.\{t\}\ A$, in which A the *quantifier body*, t is the *trigger*
- A trigger is a term t (of any sort), satisfying the following criteria:
  - t must *contain all of the variables quantified* by the quantifier
  - t may *not contain interpreted function symbols* (except for constants)
  - t must contain *at least one non-constant function symbol* (it cannot just be x)
- For example, $\forall x{:}\mathrm{Int}.\{f(x)\}\ f(x)<b$ is a quantifier with a trigger $f(x)$
- Triggers are *typically* terms in the quantifier body, but *not necessarily*

Ground Terms and Trigger Matching

- A *ground term* is a term containing no variables
  - e.g. $f(3)$ is a ground term, in $\forall x{:}Int.\{f(x)\}\ f(x)<b$ the subterm $f(x)$ is not
- Triggers provide a mechanism for controlling quantifier instantiations
- The idea: a quantifier $\forall x.\{t\}\ A$ will (only) be instantiated when:
  - a *ground term* $t[t'/x]$ occurs in our current formula to satisfy / current model
  - in this case, the corresponding quantifier instantiation $A[t'/x]$ will be made
  - this instantiated formula $A[t'/x]$ is *conjoined* to the current formula to satisfy
  - the instantiation will only be made *once* (for the same quantifier and term t')
- e.g. given $g(f(a))=0 \wedge \forall x{:}Int.\{g(f(x))\}\ g(f(x))=1$ the term $g(f(a))$ *matches the trigger* $g(f(x))$, causing the instantiation $g(f(a))=1$
- Without a matching ground term, *no information* will be deduced from the quantifier body: e.g. $(\forall x.\{p(x)\}\ p(x)) \wedge \forall x.\{p(x)\}\ \neg p(x)$

E-Matching

- Consider a slight variant on the previous example:

$$g(b)=0 \wedge b=f(a) \wedge \forall x{:}\mathrm{Int}.\{g(f(x))\}\ g(f(x))=1$$

- This contains no ground terms of the shape $g(f(x))$; according to the rules of the previous slide, we would not instantiate the quantifier
- Improvement: a quantifier $\forall x.\{t\}\ A$ will (only) be instantiated when:
  - there are *ground terms* t' and t'' such that t'' occurs in our current formula to satisfy / current model and $t[t'/x]=t''$ is *true in our current model*
  - in this case, the corresponding quantifier instantiation $A[t'/x]$ will be made
- Triggers are matched *modulo equalities* (*E-matching*)
  - E-matching can be efficiently implemented by *pattern-matching* triggers against the current E-graph (exploring known equivalence classes on terms)
  - In some tools (and in SMT-LIB), triggers are called *patterns*

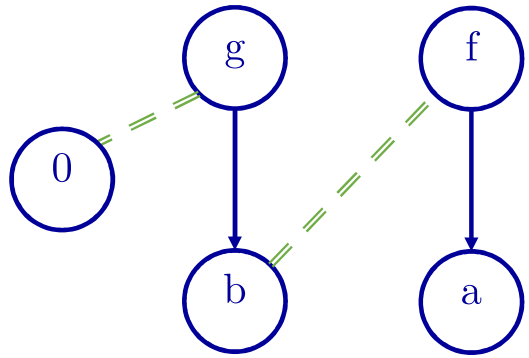
E-Matching in Action

- Consider the previous example:

$$g(b)=0 \land b=f(a) \land$$
$$\forall x{:}\mathrm{Int}.\{g(f(x))\}\ g(f(x))=1$$

- DPLL search will add $g(b)=0,\ b=f(a)$ as (ground) literals, and record the quantifier as necessarily true
- Build the E-graph for ground literals
- Match trigger $g(f(x))$ against E-graph:
  - start from each g node
  - for their (only) argument nodes, search the equivalence class for each f nodes
  - use argument of (each) f for instantiation

$g(b)=0,\ b=f(a)$



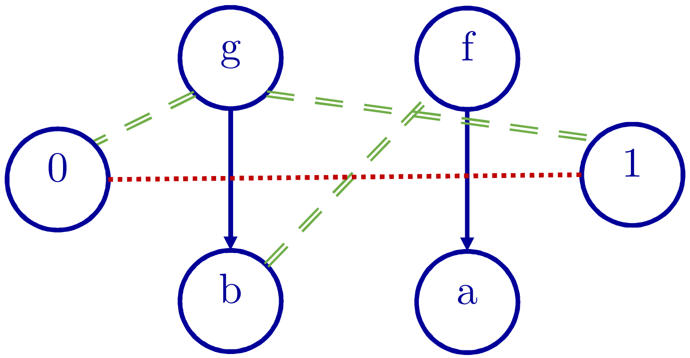
E-Matching in Action

- Consider the previous example:

$$g(b)=0 \land b=f(a) \land$$
$$\forall x{:}\text{Int}.\{g(f(x))\}\ g(f(x))=1$$

- DPLL search will add $g(b)=0,\ b=f(a)$ as (ground) literals, and record the quantifier as necessarily true
- Build the E-graph for ground literals
- Match trigger $g(f(x))$ against E-graph:
  - start from each g node
  - for their (only) argument nodes, search the equivalence class for each f nodes
  - use argument of (each) f for instantiation
  - We get a match for $g(f(a))$

$g(b)=0,\ b=f(a),\ g(f(a))=1$



Integrating E-Matching for Quantifier Instantiation

run DPLL-like search on the *propositional abstraction* A^p of the input formula A

deduce model choices

choose a decision literal

deduce model choices

choose a decision literal

…

- maintain current (dis)-equality information in an E-graph
  - recall: this information is also needed for theory combination
- when a quantified literal is added to the candidate model:
  - *record the quantifier* for potential E-matching
- periodically, run an E-matching engine on the current E-graph
  - look for new instantiations of recorded quantifiers, and add them
  - this expands the current formula for DPLL search (new clauses)
  - note: we never check the truth of quantifiers in a model (unlike in MBQI)

Selecting Triggers I

- Choosing appropriate triggers for quantifiers can be a difficult task
- Triggers may be *too restrictive*: we may miss relevant instances
  - e.g. we don't get `unsat` (using E-matching) on the following example:
    $\neg(\mathrm{a}=\mathrm{b}) \wedge \mathrm{f}(\mathrm{a})=\mathrm{f}(\mathrm{b}) \wedge \forall \mathrm{x}$:Int.$\{\mathrm{g}(\mathrm{f}(\mathrm{x}))\}\ \mathrm{g}(\mathrm{f}(\mathrm{x}))=\mathrm{x}$ ("g is the inverse of f")
  - Changing the trigger to be just $\mathrm{f}(\mathrm{x})$ will get us `unsat`
- Triggers may be *too permissive*: we may get too many instantiations
  - e.g. what triggers could we choose for $\forall \mathrm{x}$:Int. $\neg\ \mathrm{g}(\mathrm{f}(\mathrm{x}))=\mathrm{g}(\mathrm{x})$?
  - Assuming we choose a term from the quantifier body (not a requirement):
  - Choosing $\mathrm{g}(\mathrm{f}(\mathrm{x}))$ as a trigger again seems too restrictive (e.g. add $\mathrm{f}(\mathrm{a})=\mathrm{a}$)
  - Choosing $\mathrm{g}(\mathrm{x})$ as a trigger has a different problem: any ground term $\mathrm{g}(\mathrm{a})$ will cause us to instantiate, adding a term $\mathrm{g}(\mathrm{f}(\mathrm{a}))$ which also matches the trigger…
  - This situation leads to infinite instantiations, and is called a *matching loop*

Selecting Triggers II

- Sometimes a single trigger term is hard or impossible to find
- Triggers may, in general, consist of *sets of terms* within $\{\ldots\}$
  - e.g. $\forall x{:}\mathrm{Int}.\ \{f(x),g(x)\}\ g(f(x))=g(x)$ will be instantiated only when we have *both* ground terms $f(t)$ and $g(t)$ for some term t
- We can also write multiple, alternative trigger sets on a quantifier:
  - e.g. $\forall x{:}\mathrm{Int}.\ \{f(x)\}\{g(x)\}\ g(f(x))=g(x)$ will be instantiated when we have *either* a ground term $f(t)$ or $g(t)$ for some term t
- Conceptually, we need to triggers which define *relevant instantiations*
- We must simultaneously try to avoid:
  - needing instantiations when we don't have the triggers (*too restrictive*)
  - generating too many irrelevant instantiations (triggers *too permissive*)
  - as a special case of the latter, we must avoid the potential for *matching loops*

Quantifiers - Summary

- We have seen two alternative techniques for *lazy quantifier support*
  - we have also seen an *eager approach* for Effectively Propositional Logic
- The first, *Model-Based Quantifier Instantiation* can generate models
  - for decidable fragments, termination guarantees, but not in general
- The second, *E-matching* selects instantiations based on *triggers*
  - can be applied in settings where MBQI would not terminate
  - depends heavily on carefully-chosen triggers (we will see this issue a lot)
  - because quantifiers are not checked true, models are not guaranteed
  - incomplete: with (only) E-matching, SMT solver will return `unsat` or `unknown`
- We've now covered the necessary tool support for a program verifier
  - We will heavily use *E-matching*, *uninterpreted functions* and other *theories*
  - In the next lecture, we'll look using these for *encoding problems into SMT*

Quantifiers – Some References

- *Handbook of Satisfiability.* Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (2009)
- MBQI and related techniques:
  - *Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories.* Yeting Ge, Leonardo de Moura (2009)
  - *Quantifier Instantiation Techniques for Finite Model Finding in SMT.* Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, Clark Barrett (2013)
  - *Model Finding for Recursive Functions in SMT.* Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, Cesare Tinelli (2016)
- E-Matching:
  - *Efficient E-Matching for SMT Solvers.* Leonardo de Moura, Nikolaj Bjørner (2007)
  - *Programming with Triggers.* Michał Moskal (2009)
- Effectively Propositional Logic: *On a problem in formal logic.* Frank Ramsey (1928).
  - Also, search for *"Bernays–Schönfinkel-Ramsey"*
- Other teaching material: *Quantifiers.* Leonardo de Moura (SAT/SMT Summer School 2012)
- See also: *Z3 – A Tutorial.* Leonardo de Moura, Nikolaj Bjørner (2011)
  - and `http://rise4fun.com/z3/tutorial`