The Nuts and Bolts of Yices

Bruno Dutertre
SRI International

SMT 2016
Coimbra, Portugal

Yices 2

Ancestors

- ICS (Rueß & de Moura, 2002)
- Yices (de Moura, 2005) and Simplics (Dutertre, 2005)
- Yices 1 (de Moura & Dutertre, 2006)

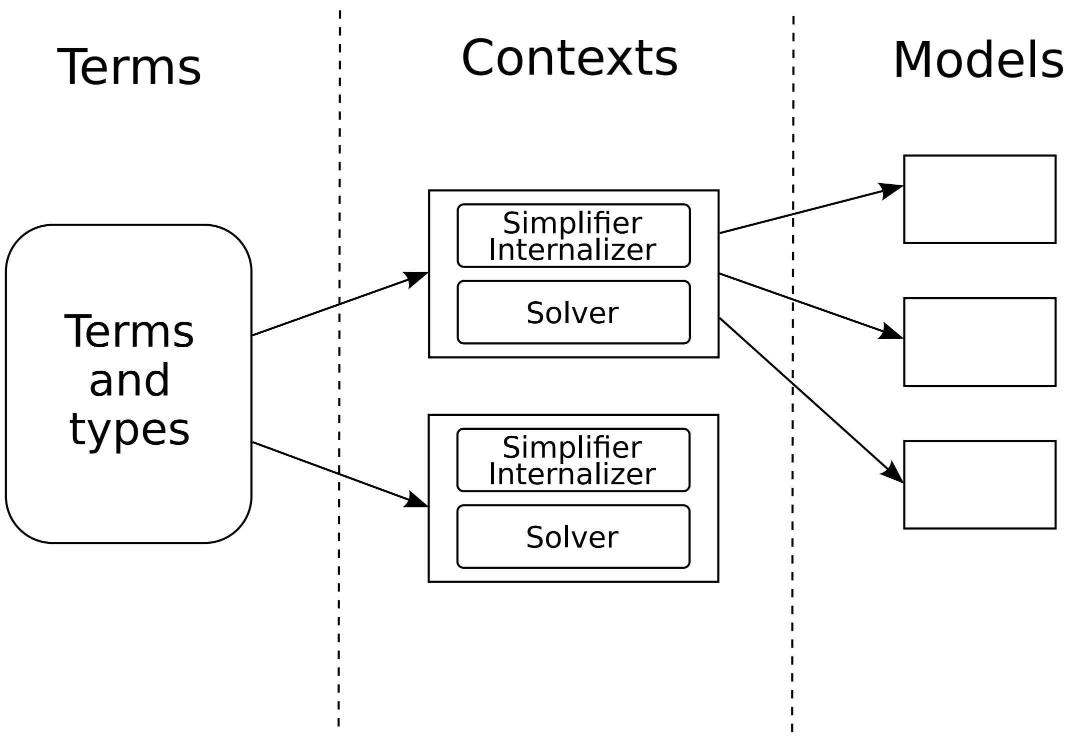
Current Status

- Yices 2.4.2, released in December 2015
- Supports linear and non-linear arithmetic, arrays, UF, bitvectors
- Limited quantifier reasoning: $\exists\forall$ fragments for bitvector, LRA
- Includes two types of solvers: classic DPPL(T) + MC-SAT

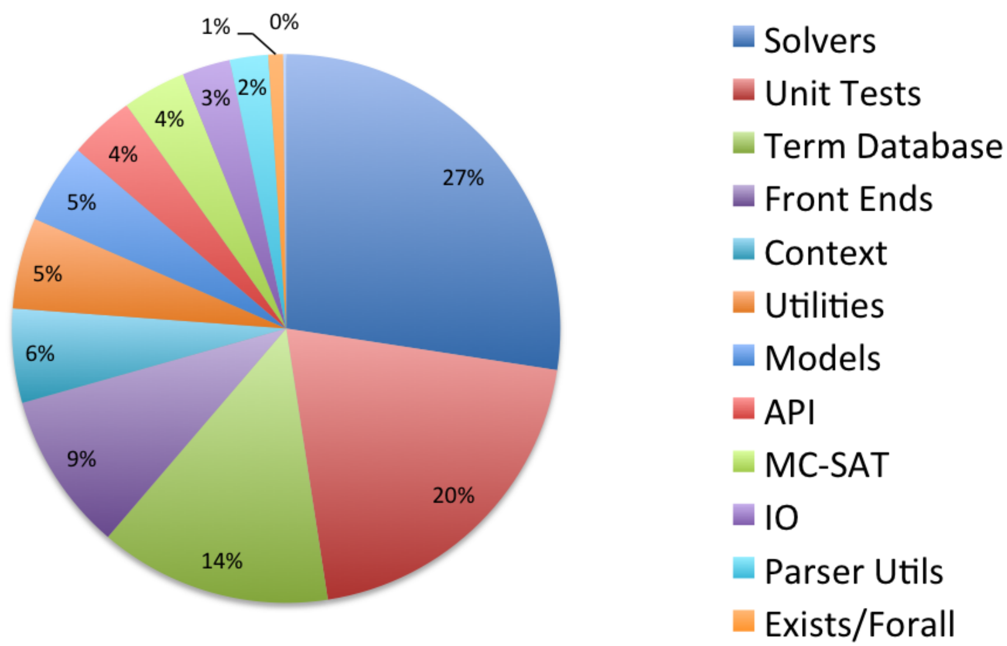
Distributions

- Free for non-commercial use
- Source + binaries distributed at (`http://yices.csl.sri.com`)

Overall Architecture



Code Breakdown



About 220K lines of C code total (C99)

Common Patterns

Tables

- Many objects are identified by an integer index i
- Then a table stores descriptors for this object at index i
- Example: term table
  - For a term t, the table stores:

    $\mathsf{kind}[t]$: tag such as `ITE_TERM`

    $\mathsf{type}[t]$: type of t (an integer index in the type table)

    $\mathsf{desc}[t]$: pointer to t's descriptor.
  - The descriptor includes arity + children (represented as integer indices)
- Benefit:
  - compact representation, small descriptors

Common Patterns

Implicit Negation

- No explicit NOT operator, we use a polarity bit (as in SAT solvers)
- Given a Boolean term t, we represent two variants of t
  - positive variant t^+ denotes t, negative variant t^- represents $\neg t$
  - the polarity is added to the term index (in the low-order bit):

```
static inline term_t pos_term(int32_t i) {
  return (i << 1);
}

static inline term_t neg_term(int32_t i) {
  return (i << 1) | 1;
}
```

Common Data Structures

Utilies

- many variants of hash tables and hash maps
- vectors, queues, stacks
- basic algorithm: sorting + a few others

Exact Rational Arithmetic

- small rationals are common
- we use our own implementation of rationals (as pairs of 32-bit integers)
- we convert to GMP rational when 32 bits are not enough

Apart from GMP (and libpoly), Yices doesn't use third-party libraries

DPLL(T) Basics

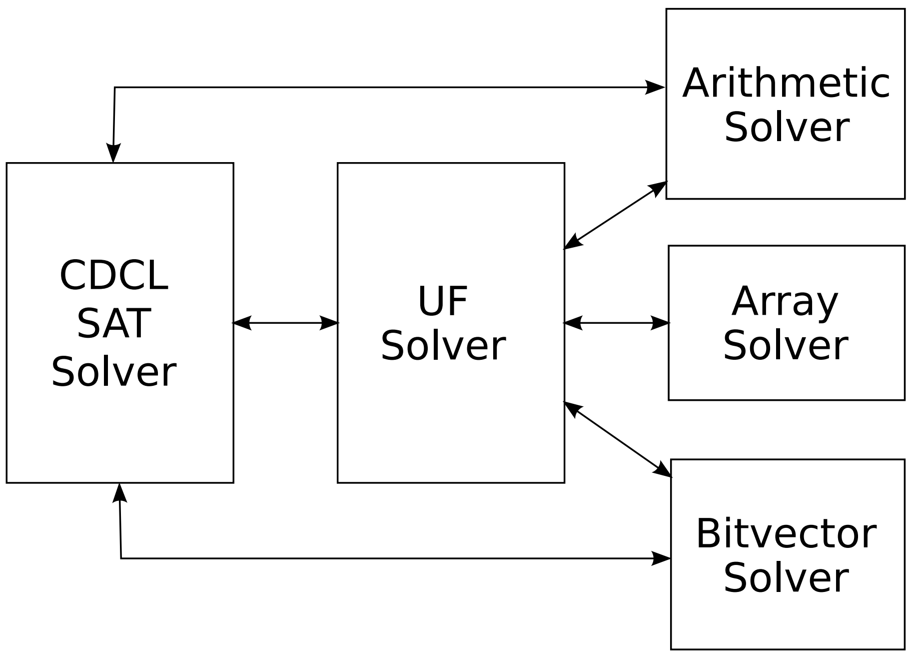
Basic ideas

- Combination of a CDCL-based SAT solver and a theory solver
- Boolean variables in the SAT solver are mapped to atoms in theory T
- The SAT solver assigns truth-values to the atoms.
- The theory solver checks whether the truth assignment is consistent in T

(Minimial) Theory Solver

- Checks whether a conjunction of literals $\phi_1 \wedge \ldots \wedge \phi_n$ is satisfiable in theory T
- If not, produces an explanation: subset of $\phi_1, \ldots, \phi_n$ that's inconsistent.

DPLL(T) Architecture in Yices



Common Features of Real Theory Solvers

Theory Propagation

- set the truth value of an atom in the SAT solver when it's implied in T

$$\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \phi'$$

Dynamic Clauses and Variables

- splitting on demand (Barrett, et al., 2006): add new atoms on the fly
- in UF theory: "dynamic Ackermannization" (de Moura & Bjørner, 2007)
- array theory: lazy instantiation of array axioms

The SAT solver must support these features. This goes beyond what off-the-shelf SAT solvers provide.

DPLL(T) Core in Yices 2

SAT Solver Interface

```
create_boolean_variable(...)
attach_atom_to_bvar(...)
add_clause(...)
propagate_literal(...)
record_theory_conflict(....)
```

Theory Solver Interface

```
assert_atom(...)
propagate(...)
expand_explation(...)
backtrack(...)
final_check(...)
```

Rules

- The theory solver can call propagate_literal only within propagate.
- The theory solver can't add clauses or variables within assert_atom (i.e., during BCP).

Lazy Explanations

Goal

- Avoid the cost of constructing clauses for every propagation (because that can be expensive)
- Only propagations involved in a conflict need such a clause

Two Step Approach

- at propagation time: the theory solver calls

  ```
  propagate_literal(core, l, exp)
  ```

  where `exp` is anything the solver may later need to generate the explanation.
- during conflict resolution, the SAT solver calls

  ```
  expand_explanation(solver, l, exp, &vector)
  ```

  to expand the explanation into a conjunction of literals (that implies `l`).

Dynamic Clause Addition

l_0	l_1	l_2							l_n

Normal SAT Solving

- Clauses are added before the search
- All literals are unassigned, we can pick any two as watch literals

In SMT Context

- Clauses are added during the search
- Some literals may be assigned (true or false)
- Need to search for two watch literals in the clause

Two Watch Literals in Dynamic Clauses

Preference Relation

- For every literal l_i in the clause, let v_i be the value assigned to l_i and k_i the decision level of l_i (if assigned)
- Preference relation: $\sqsubset$ defined by

$$v_i = \mathsf{undef} \ \wedge \ v_j = \mathsf{false} \ \Rightarrow \ l_i \sqsubset l_j \qquad v_i = \mathsf{true} \ \wedge \ v_j = \mathsf{undef} \ \Rightarrow \ l_i \sqsubset l_j$$

$$v_i = v_j = \mathsf{false} \ \wedge \ k_i > k_j \ \Rightarrow \ l_i \sqsubset l_j \qquad v_i = v_j = \mathsf{true} \ \wedge \ k_i < k_j \ \Rightarrow \ l_i \sqsubset l_j$$

Dynamic Clause Addition

- Pick two smallest literals for $\sqsubset$. If neither is false, they can be watched literals.
- If one is false and the other is undef backtrack and perform an Boolean propagation.
- If both are false, backtrack and resolve the conflict.

A Trick: Heuristic Caching of Theory Lemmas

Lemma Caching

- Theory explanations and conflicts are converted to clauses during conflict resolution.
- Normally, these clauses are not stored in the SAT solver.
- Caching is a heuristic that selects theory lemmas and keep them as learned clauses.

Heuristic

- Cache only small theory lemmas (max size is a parameter)
- Cache only lemmas for which we can find two watch literals without backtracking

Congruence Closure and E-Graph

Congruence Closure

- Basic theory: deals with equalities and uninterpreted functions
- Well-known implementations:
  - Build an equivalence relation between term
  - Merge two classes when they contain congruent terms:

$$x = y \land t = u \ \Rightarrow f(x, t) = f(y, u)$$

  - In SMT, bookkeeping to generate explanations (Nieuwenhuis & Olivera, 2006)

Yices Implementation

- Congruence closure extended to deal with Boolean terms
- Handles equalities as terms
- Efficient data structures for maintaining use lists (a.k.a. parents)

Congruence-Closure: Terms

Terms and Occurrences

- Terms are denoted by integers from 0 to $\mathsf{nterms} - 1$
- For a Boolean terms t, we distinguish between positive t^+ and negative t^- occurrences (t^- is the same as $\neg t$).
- For non-Boolean terms, all occurrences are positive.

Term Descriptors

- Each term t has a descriptor $\mathsf{body}[t]$ that can be of the following forms:
  - $(\mathsf{apply}\ f\ t_1 \ldots t_n)$: uninterpreted function application where f, t_1, . . . , t_n are term occurrences.
  - $(\mathsf{eq}\ t_1\ t_2)$: equality
  - $\mathsf{variable}$: atomic, uninterpreted term
- Term $t = 0$ represents the Boolean constant. (0^+ is true and 0^- is false)

Congruence Closure: Classes

Equivalence Class

- Identified by an integer between 0 and $\mathsf{nclasses} - 1$
- A class stores a set of term occurrences known to be equal
- These are stored in a circular list:
  - $\mathsf{label}[t]$: class to which term t belongs (with a polarity bit)
  - $\mathsf{next}[t]$: successor of t in the circular list (with a polarity bit)
- For a class of Boolean terms, there's an implicit complementary class that contains the same terms with opposite polarities

Example

- If t, $\neg u$, and $\neg v$ are in the same class C

$$\begin{array}{ll} \mathsf{next}[t] = u^- & \mathsf{label}[t] = C^+ \\ \mathsf{next}[u] = v^+ & \mathsf{label}[u] = C^- \\ \mathsf{next}[v] = t^- & \mathsf{label}[v] = C^- \end{array}$$

Two classes: $C^+ = \{t^+, u^-, v^-\}$ and $C^- = \{t^-, u^+, v^+\}$.

Class Attributes

Parent Vector

- $\mathsf{parents}[C]$: vector of term descriptors (pointers)
- Each element in $\mathsf{parents}[C]$ is a composite term, parent of a term of class C
- Example:

  if t^+ is in C, then $\mathsf{parents}[C]$ contains terms in which t occurs, e.g.,

  $$(\mathsf{apply}\ f\ t\ u) \quad (\mathsf{eq}\ z\ t) \quad (\mathsf{apply}\ g\ u\ t\ t)$$

Root

- $\mathsf{root}[C]$: class representative = an element of C
- This is also the root of C's merge tree

Congruence Roots

Congruent Terms

- $(\text{apply } f\ t_1 \ldots t_n)$ is congruent with $(\text{apply } g\ u_1 \ldots\ u_n)$ if
  $\text{label}[f] = \text{label}[g],\ \text{label}[t_1] = \text{label}[u_1],\ \ \ldots,\ \ \text{label}[t_n] = \text{label}[u_n]$
- $(\text{eq } t_1\ t_2)$ is congruent with $(\text{eq } u_1\ u_2)$ if
  $\text{label}[t_1] = \text{label}[u_1]$ and $\text{label}[t_2] = \text{label}[u_2]$ or
  $\text{label}[t_1] = \text{label}[u_2]$ and $\text{label}[t_2] = \text{label}[u_1]$.

Congruence Roots

- For every class of congruent terms, exactly one representative is stored in a hash table. It's the congruence root.

Simplifications for Equalities

- $(\text{eq } t_1\ t_2)$ simplifies to true if $\text{label}[t_1] = \text{label}[t_2]$
- $(\text{eq } t_1\ t_2)$ simplifies to false if $\text{label}[t_i] = \neg\text{label}[t_2]$.

Congruence Closure

Based on Merging Classes

- When C_1 and C_2 are merged, we must visit all parents of, say, C_1 to check whether they have become congruent to some other term.
- For each p in $\mathsf{parent}[C_1]$:
  - If p is not a congruent root, skip it.
  - Otherwise:
    1. remove p from the hash table
    2. compute p's new signature
    3. search for a q with the same signature in the hash table
    4. if such a q exists then p is congruent to q, merge their classes
    5. otherwise p is a congruence root, put it back in the hash table.

Performance Issue

- How to avoid visiting terms that are not congruence roots?
- Need to remove p from all its parent vectors in step 4 above.

Composite and Parent Vector Implementation

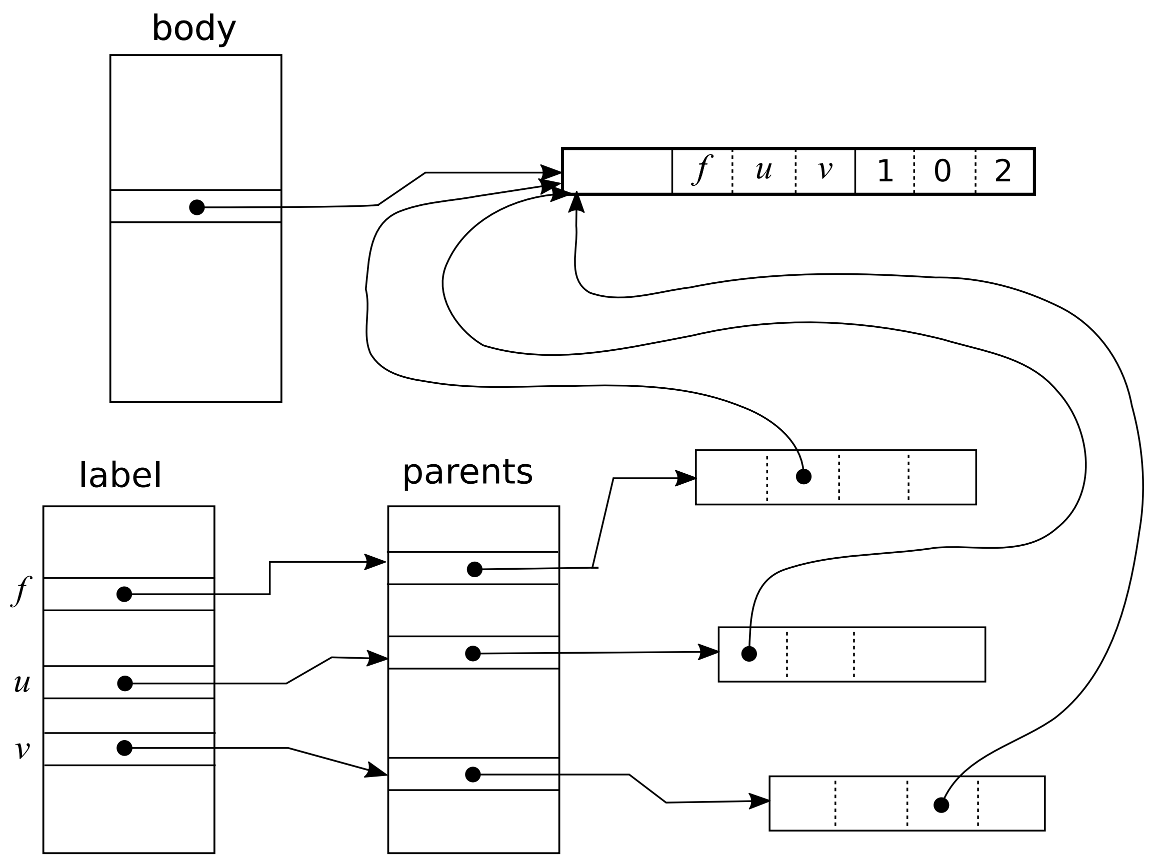
Composite Stucture

- a header: tag + arity, hash, term id
- an array of n children
- an array of n integer indices (hooks)

Invariant

- If the i-th child of p is in class C, then p is stored in $\mathsf{parents}[C]$ at some index k and we have $p \mapsto \mathsf{hook}[i] = k$.
- From p, we can find the parent vectors that contain p and the positions in each vectors where p is stored.
- This allows p to be removed from all its parent vectors, without scanning the vectors.

Composite and Parent Vector Implementation



Preprocessing and Simplification

Preprocessing and formula simplification are not glamorous but they are critical to SMT solving:

- Many SMT-LIB benchmarks are accidently hard: they become easy (sometimes trivial) with the right simplification trick
  - Examples: `eq_diamond`, `nec-smt` problems, `rings` problems, `unconstrained` family
- This is not just in the SMT-LIB benchmarks:
  - Bitvector problems are typically solved via bit-blasting (i.e., converted to Boolean SAT). But without simplification, bit-blasting can turn easy problems into exponential search.
  - There are other problems that just can't be solved without the right simplifications.

Example: Nested if-then-elses

How do we deal with non-boolean if-then-else?

- Lifting:
  - Rewrite `(>= (ite c t1 t2) u)` to `(ite c (>= t1 u) (>= t2 u))`
  - Risk exponential blow up if `u` is an if-then-else term

- Use an auxiliary variable
  - Replace `(ite c t1 t2)` by a fresh variable `z` and add constraints. For example, `(>= (ite c t1 t2) u)` is converted to

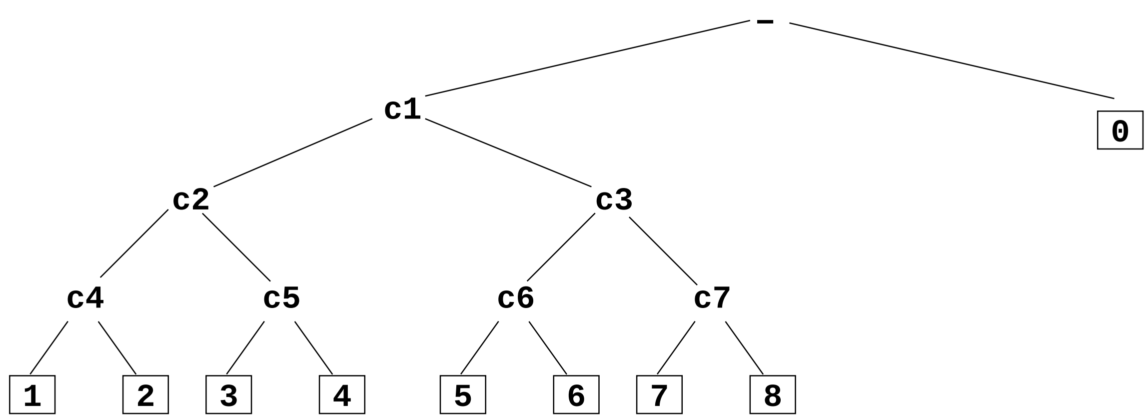
    ```
    (>= z u)
    (implies c (= z t1))
    (implies (not c) (= z t2))
    ```

  - Benefit: this does not blow up

Nested if-then-else (cont'd)

But lifting may still work better

- Example: `(= t1 a)` when `t1` is a nested if-then-else with all leaves trivially distinct from `a`.



Approach in Yices

Special ITE

- If all leaves of an if-then-else term t are constant, it's marked as special
- We can then compute the domain of t: finite set of constant values:

$$\begin{aligned} \mathsf{dom}((\mathsf{ite}\ c\ t_1\ t_2)) &= \mathsf{dom}(t_1) \cup \mathsf{dom}(t_2) \\ \mathsf{dom}(a) &= \{a\} \ \text{ if } a \text{ is a constant} \end{aligned}$$

Example Simplification Rules

$$\begin{aligned} \mathsf{dom}(t) = a &\longrightarrow \mathsf{false} && \text{if } a \notin \mathsf{dom}(t) \\ \mathsf{dom}((\mathsf{ite}\ c\ t_1\ t_2)) = a &\longrightarrow c \wedge\ t_1 = a && \text{if } a \notin \mathsf{dom}(t_2) \\ \mathsf{dom}((\mathsf{ite}\ c\ t_1\ t_2)) = a &\longrightarrow \neg c \wedge\ t_2 = a && \text{if } a \notin \mathsf{dom}(t_1) \end{aligned}$$

Flattening to Avoid Auxiliary Variables

Direct translation for $(\mathsf{ite}\ c_1\ (\mathsf{ite}\ c_2\ a_2\ b_2)(\mathsf{ite}\ c_3\ a_3\ b_3))$

- Introduces one variable for each ite term:

$$x_1 = (\mathsf{ite}\ c_1\ x_2\ x_3) \quad x_2 = (\mathsf{ite}\ c_2\ a_2\ b_2) \quad x_3 = (\mathsf{ite}\ c_3\ a_3\ b_3)$$

- Converts to six clauses:

$$\begin{array}{ccc} c_1 \Rightarrow x_1 = x_2 & \neg c_1 \Rightarrow x_1 = x_3 & c_2 \Rightarrow x_2 = a_2 \\ \neg c_2 \Rightarrow x_2 = b_2 & c_3 \Rightarrow x_3 = a_3 & \neg c_3 \Rightarrow x_3 = b_3 \end{array}$$

Better Translation

- Don't introduce x_2 and x_3 and produce fewer clauses:

$$\begin{array}{cc} c_1 \wedge\ c2 \Rightarrow x_1 = a_2 & c_1 \wedge \neg c_2 \Rightarrow x_1 = b_2 \\ \neg c_1 \wedge c_3 \Rightarrow x_1 = a_3 & \neg c_1 \wedge \neg c_3 \Rightarrow x_1 = b_3 \end{array}$$

- Must be applied carefully if some sub-terms have several occurrences
- Very useful for problems that combine UF and arithmetic: removing auxiliary variables helps the E-graph generate short explanations

Conclusion

SMT Solvers

- A lot more than an SAT solver + theory solvers
- Parsing, term representation, simplification, preprocessing represent more code in Yices
- Engineering matters: low-level details make a difference

Other People Involved