

CVC Lite: Selected Stories from the Trenches

Clark Barrett

New York University

Outline

- *Nelson-Oppen in CVC Lite*
- Leveraging Proofs
- Hybrid SAT Methods
- Partial Functions

The Nelson-Oppen Method

We have seen the Nelson-Oppen method many times. Now I get to give my version. First a definition: If S is a set of terms and \sim is an equivalence relation on S , then the *arrangement of S induced by \sim* is

$$Ar_{\sim} = \{x = y \mid x \sim y\} \cup \{x \neq y \mid x \not\sim y\}.$$

Suppose that $\mathcal{T}_1, \dots, \mathcal{T}_N$ are N signature-disjoint stably-infinite first-order theories, with signatures $\Sigma_1, \dots, \Sigma_N$. Let $\mathcal{T} = \bigcup \mathcal{T}_i$ and $\Sigma = \bigcup \Sigma_i$.

Let Sat_i be a decision procedure \mathcal{T}_i -satisfiability. Suppose we wish to determine the satisfiability of a conjunction Φ of quantifier-free Σ -formulas.

1. Purify Φ to obtain a separate form $\bigwedge \Phi_i$.
2. Let S be the set of variables shared among the Φ_i (appearing in at least two).
Guess an equivalence relation \sim on S .
3. For each i , run Sat_i on $\Phi_i \cup Ar_{\sim}$.

If there exists an equivalence relation \sim such that Sat_i returns true for each i , then Φ is \mathcal{T} -satisfiable. Otherwise, Φ is \mathcal{T} -unsatisfiable.

The Nelson-Oppen Method

It has long been known that the purification step is not really necessary. However, it was not obvious (to me) how to formalize this at first. Here is a variation of Nelson-Oppen without purification.

Suppose, as before, we wish to determine the satisfiability of a conjunction Φ of quantifier-free Σ -formulas.

1. Partition Φ into N sets Φ_i such that Φ_i contains literals whose topmost non-logical symbol is from Σ_i .
2. Let S be the set of all terms which occur i -alien (for some i) in some literal in Φ or in some sub-term of some literal in Φ . We call S the set of *shared* terms. Guess an equivalence relation \sim on S .
3. For each i , run Sat_i on $\Phi_i \cup Ar_{\sim}$, treating terms beginning with symbols from other signatures as variables.

As before, if there exists an equivalence relation \sim such that Sat_i returns true for each i , then Φ is \mathcal{T} -satisfiable. Otherwise, Φ is \mathcal{T} -unsatisfiable.

A More Concrete Algorithm

In practice, we want an incremental algorithm. Suppose Φ is a set of literals. We will process Φ one literal at a time. We repeat the following process until an inconsistency is detected or Φ is empty.

1. Remove a literal ϕ from Φ .
2. Simplify ϕ : traverse ϕ , applying two important transformations:
 - (a) Each sub-term is replaced by its equivalence class representative.
 - (b) Rewrites are applied locally to each sub-term:
 - i. Canonization / normalization
 - ii. Boolean simplifications
3. If ϕ is an equality,
 - (a) Apply a solver to ϕ if applicable;
 - (b) Record ϕ in the global union-find data structure;
 - (c) Notify theories on the *notify list* of the left-hand side of ϕ .
4. Identify shared terms in ϕ and send ϕ to the appropriate theory.
5. Each theory checks the satisfiability of its formulas together with the arrangement of shared terms induced by the global union-find database. Each theory either returns satisfiable, returns unsatisfiable, or generates a new formula which constrains the global union-find database.

Outline

- Nelson-Oppen in CVC Lite
- *Leveraging Proofs*
- Hybrid SAT Methods
- Partial Functions

Using Proofs

Some benefits of proofs we've already discussed

- Checking “soundness” on the fly
- Using assumptions for conflict analysis
- Some users want proof objects for various reasons

Another way to use proofs

I'd like to talk about another use of proofs: using proofs to import CVC Lite results into other theorem provers. This provides two benefits:

- The other theorem prover can function as a proof-checker for CVC Lite.
- CVC Lite can be integrated as an *untrusted oracle* within the theorem prover.

I will discuss a case study done with Sean McLaughlin on integrating CVC Lite with HOL Light.

HOL Light

HOL Light is an interactive theorem prover written by John Harrison descended from the LCF projects and the HOL4 theorem prover.

All the theorems are created by a core set of *10 primitive inference rules* such as modus ponens and reflexivity. The core consists of just over *300 lines* of OCaml.

All other rules of inference are derived *conservatively* from these rules. With OCaml as the metalanguage, the user may program arbitrary new rules that cannot compromise the correctness of the system.

Because of its transparent design and minimal base of trusted code, HOL Light was chosen by Thomas Hales as the system in which to formalize his proof of the Kepler Conjecture.

One of the reasons for combining CVC Lite and HOL Light was to see if CVC Lite could help with this project.

Proofs in CVC Lite

Proofs are represented in CVCL by *sequents*.

A sequent is a pair $\Gamma \vdash \phi$, where Γ is a set of *assumptions* and ϕ is a formula.

A sequent is valid iff $\mathcal{T} \cup \Gamma \models \phi$, where \mathcal{T} is the union of all theories participating in the cooperating framework of CVCL.

A *proof rule*, or *inference rule* is denoted as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the P_i 's are *premises* and C is the conclusion of the rule (all are sequents).

A rule is *sound* if the validity of all premises implies the validity of the conclusion.

The set of premises may be empty, in which case the rule is called an *axiom*.

A *proof* or *derivation* of a sequent C is a sequence of proof rule applications that forms a finite proof tree with C as the root and axioms on the leaves.

Translating Proofs

A proof tree in CVCL is an expression. Each node is labeled with the inference rule and its children are the relevant premises.

Proof translation consists of traversing the expression bottom-up, replacing nodes with HOL Light theorems.

To do this, each proof rule of CVCL must be mapped to an equivalent inference in HOL Light.

We will look at a simple example.

Translating Proofs

```
iff-mp(NOT FALSE, (x = x),
  neg-intro(NOT FALSE,
    (LAMBDA (assump-1: FALSE): assump-1))),
iff-symm((x = x), NOT FALSE,
  iff-contrapositive((NOT (x = x) <=> FALSE),
    iff-trans(NOT (x = x), NOT TRUE, FALSE,
      optimized-subst-op(NOT (x = x), NOT TRUE,
        rewrite-eq-refl(REAL, x))),
      rewrite-not-true))))
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    (LAMBDA (assump-1: FALSE): assump-1)),  
  iff-symm(  
    iff-contrapositive(  
      iff-trans(  
        optimized-subst-op(NOT (x = x)),  
        rewrite-eq-refl(x)),  
        rewrite-not-true))))
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    (LAMBDA (assump-1: FALSE): assump-1)),  
  iff-symm(  
    iff-contrapositive(  
      iff-trans(  
        optimized-subst-op(NOT (x = x),  
           $\vdash x = x \leftrightarrow true$ ),  
        rewrite-not-true))))
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    (LAMBDA (assump-1: FALSE): assump-1)),  
  iff-symm(  
    iff-contrapositive(  
      iff-trans(  
         $\vdash \neg(x = x) \leftrightarrow \neg true,$   
         $\vdash \neg true \leftrightarrow false$ ))))))
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    (LAMBDA (assump-1: FALSE): assump-1)),  
  iff-symm(  
    iff-contrapositive(  
       $\vdash \neg(x = x) \leftrightarrow \text{false}$ )))
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    (LAMBDA (assump-1: FALSE): assump-1)),  
  iff-symm(  
     $\vdash x = x \leftrightarrow \neg \text{false}$ ))
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    (LAMBDA (assump-1: FALSE): assump-1)),  
   $\vdash \neg \text{false} \leftrightarrow x = x$ )
```

Translating Proofs

```
iff-mp(  
  neg-intro(  
    false  $\vdash$  false)  
   $\vdash \neg$ false  $\leftrightarrow x = x$ )
```

Translating Proofs

iff-mp (

$\vdash \neg \text{false}$)

$\vdash \neg \text{false} \leftrightarrow x = x$)

Translating Proofs

$\vdash x = x$

Translating Proofs

Typically, proof rules in CVCL are small and simple and can be easily translated. There are a few exceptions. For example, in CVCL,

$$\begin{aligned} & (0 + 1y_4^2x_4^2 + 1y_4^2x_3^2 + 1y_4^2x_2^2 + -2y_4y_3x_2x_1 + 2y_4y_2x_3x_1 + \\ & \quad - 2y_4y_1x_4x_1 + 1y_3^2x_4^2 + 1y_3^2x_3^2 + 1y_3^2x_1^2 + 2y_3y_2x_3x_2 + \\ & \quad - 2y_3y_1x_4x_2 + 1y_2^2x_4^2 + 1y_2^2x_2^2 + 1y_2^2x_1^2 + 2y_2y_1x_4x_3 + 1y_1^2x_3^2 + \\ & \quad 1y_1^2x_2^2 + 1y_1^2x_1^2 + 0 + 1y_4^2x_1^2 + 2y_4y_3x_2x_1 + \\ & \quad - 2y_4y_2x_3x_1 + 2y_4y_1x_4x_1 + 1y_3^2x_2^2 + -2y_3y_2x_3x_2 + \\ & \quad 2y_3y_1x_4x_2 + 1y_2^2x_3^2 + -2y_2y_1x_4x_3 + 1y_1^2x_4^2) \\ & = \\ & (0 + 1y_4^2x_4^2 + 1y_4^2x_3^2 + 1y_4^2x_2^2 + 1y_4^2x_1^2 + 1y_3^2x_4^2 + 1y_3^2x_3^2 + \\ & \quad 1y_3^2x_2^2 + 1y_3^2x_1^2 + 1y_2^2x_4^2 + 1y_2^2x_3^2 + 1y_2^2x_2^2 + \\ & \quad 1y_2^2x_1^2 + 1y_1^2x_4^2 + 1y_1^2x_3^2 + 1y_1^2x_2^2 + 1y_1^2x_1^2) \end{aligned}$$

is a primitive inference.

This corresponds to **101,359** HOL Light primitive inference rule applications!

Boolean Reasoning in HOL Light

CVC Lite includes advanced Boolean SAT techniques while HOL Light does not. We can use proof translation to leverage CVC Lite's advanced Boolean capabilities.

The following table compares performance on a simple pigeonhole problem of various sizes. The columns show times in seconds for CVC Lite running alone (but still producing proofs), HOL Light running alone, and HOL Light using CVC Lite and performing the translation.

n	CVC Lite	HOL Light	CVC_PROVE
2	0.10	4.5	1.75
3	0.18	13	10
4	0.90	34	43
5	2.9	> 10000	210
6	19	> 10000	980
7	238	> 10000	4308

Adding Arrays to HOL Light

CVC Lite has a well developed theory of arrays. This theory does not exist in the current HOL Light version.

A simple way to do this is to add axioms in HOL Light for the array theory. Though a conservative extension would be better, this was an easy way to get something working quickly.

Consider the following formula in HOL Light:

```
`((S1:(real,real)array) = S2) ==>
  (write S1 i (read S2 i)) = S1)`;;
```

Given the axioms, the built-in HOL Light decision procedure can solve this problem in 56 seconds. CVC_PROVE takes .015 seconds.

Even slightly more difficult problems such as the following are intractable for HOL Light. By contrast, CVC_PROVE solved it easily.

```
`(((write (S1:(real,real)array) i v) = (write S2 j w)) /\
  (read S1 i = v) /\ (read S2 j = w)) ==>
  ((S1 = S2) /\ ((i = j) ==> (v = w)) /\
  (~ (i = j) ==> (read S1 j = w)))`;;
```

Outline

- Nelson-Oppen in CVC Lite
- Leveraging Proofs
- *Hybrid SAT Methods*
- Partial Functions

Hybrid SAT

Because CVCL does not use an external SAT solver, it does not have to translate its input into CNF, but can instead work with the original formula.

Often decision heuristics work better when the original structure of the formula is taken into account.

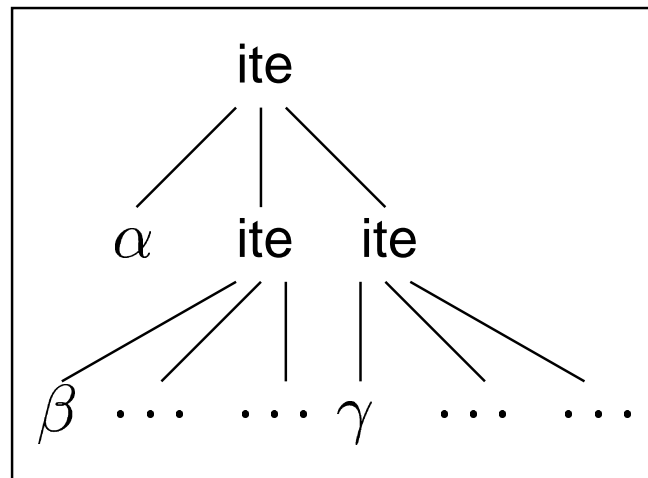
A separate database of learned conflict clauses is maintained and processed using fast BCP.

Thus, CVCL can use structure-based decision heuristics while still benefiting from learned conflict clauses.

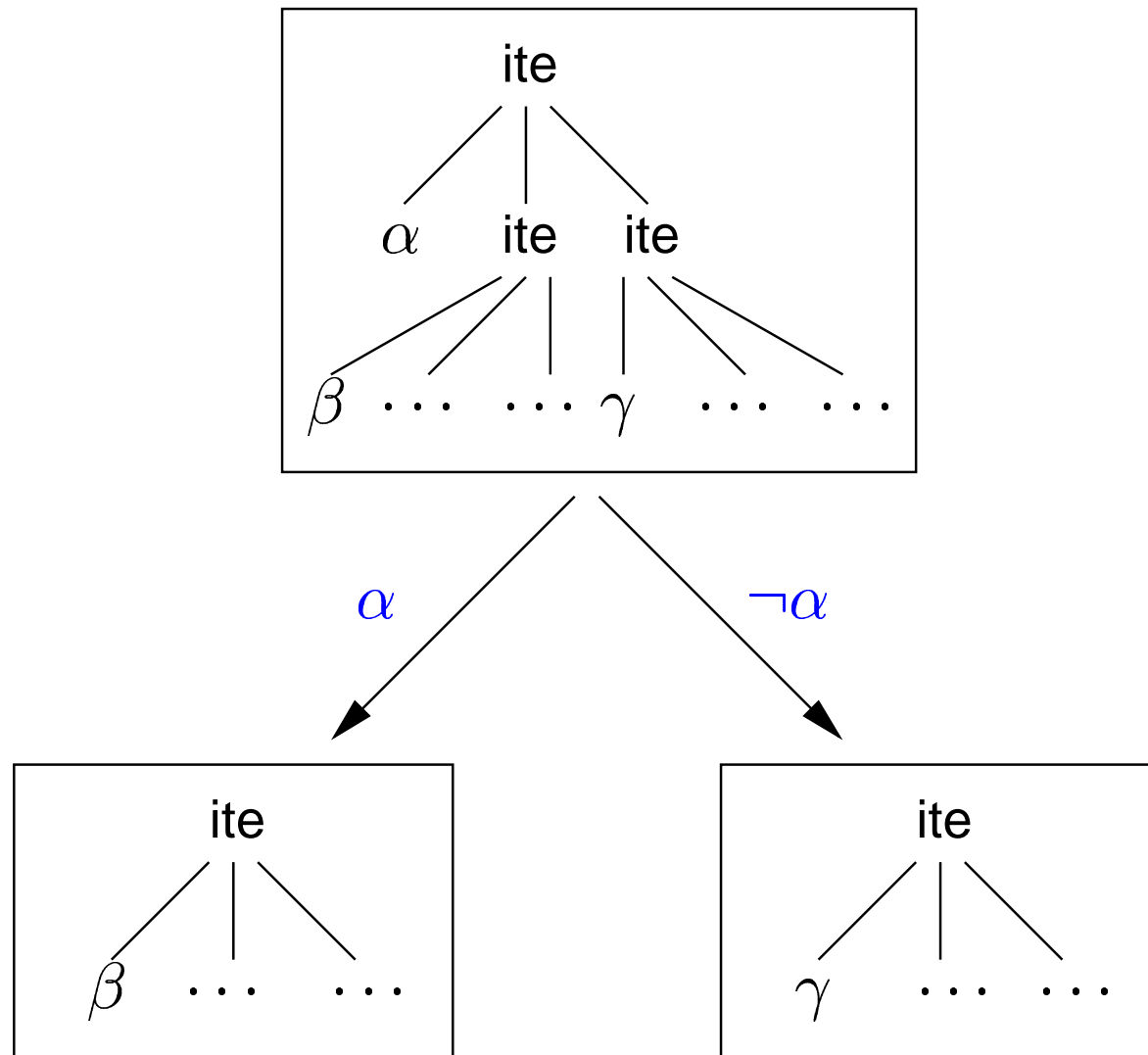
CVCL's structural decision heuristics are based on a simple depth-first search (DFS) heuristic.

DFS together with simplification of the formula works well on examples with lots of conditionals. These examples tend to perform poorly if translated into CNF.

Depth-first-search decision heuristic



Depth-first-search decision heuristic



Caching decision heuristic

The simple DFS technique can be enhanced by favoring decisions that were successful in the past.

The *caching* heuristic, keeps an LRU cache of successful splitters which is updated whenever a branch of the search is closed.

Each splitter in the cache also has a *trust* level which increases when it is used successfully and decreases over time (much like the score in Chaff).

Whenever a new decision must be made, CVCL consults the cache to see if there are any relevant splitters which have a sufficient trust level. If so, that splitter is used. If not, CVCL reverts to DFS.

Caching decision heuristic

$$(p \rightarrow x = 1) \wedge x = ite(q, 2, 3) \wedge \dots$$

Caching decision heuristic

$$(p \rightarrow x = 1) \wedge x = ite(q, 2, 3) \wedge \dots$$

p

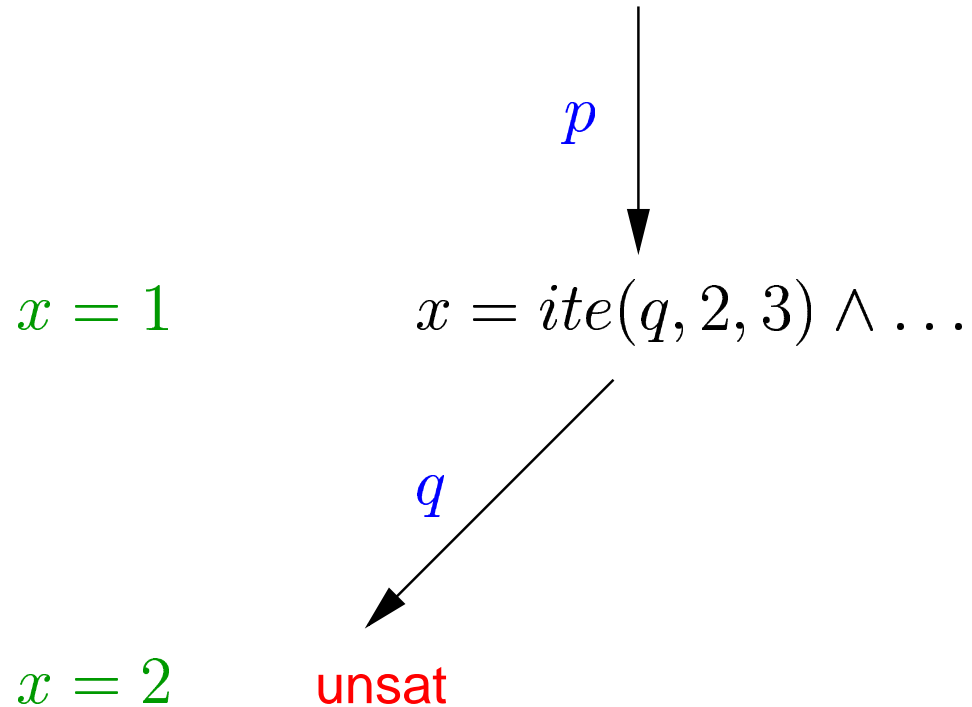


$$x = 1$$

$$x = ite(q, 2, 3) \wedge \dots$$

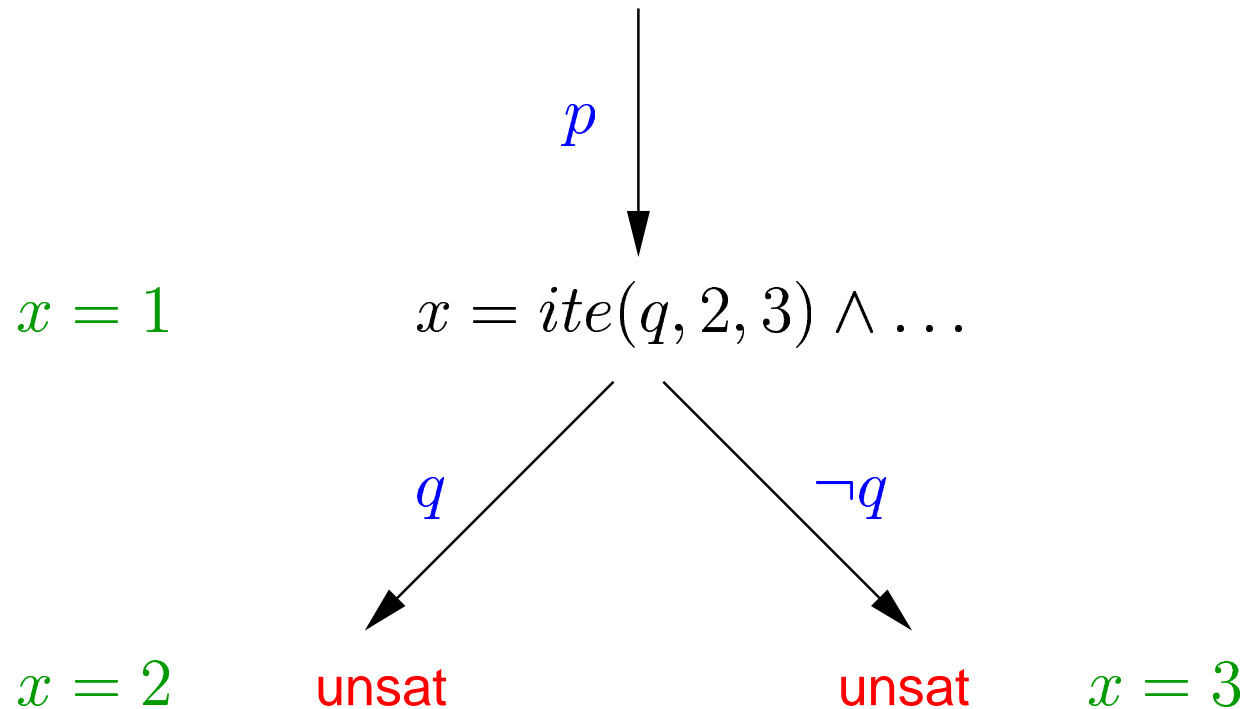
Caching decision heuristic

$$(p \rightarrow x = 1) \wedge x = ite(q, 2, 3) \wedge \dots$$



Caching decision heuristic

$$(p \rightarrow x = 1) \wedge x = \text{ite}(q, 2, 3) \wedge \dots$$



Combining caching heuristic with SAT methods

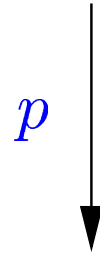
In order to combine the caching heuristic with conflict learning, we have to adjust slightly what we mean by a branch being closed.

Combining caching heuristic with SAT methods

$$(p \rightarrow x = 1) \wedge x = ite(q, 2, 3) \wedge \dots$$

Combining caching heuristic with SAT methods

$$(p \rightarrow x = 1) \wedge x = ite(q, 2, 3) \wedge \dots$$

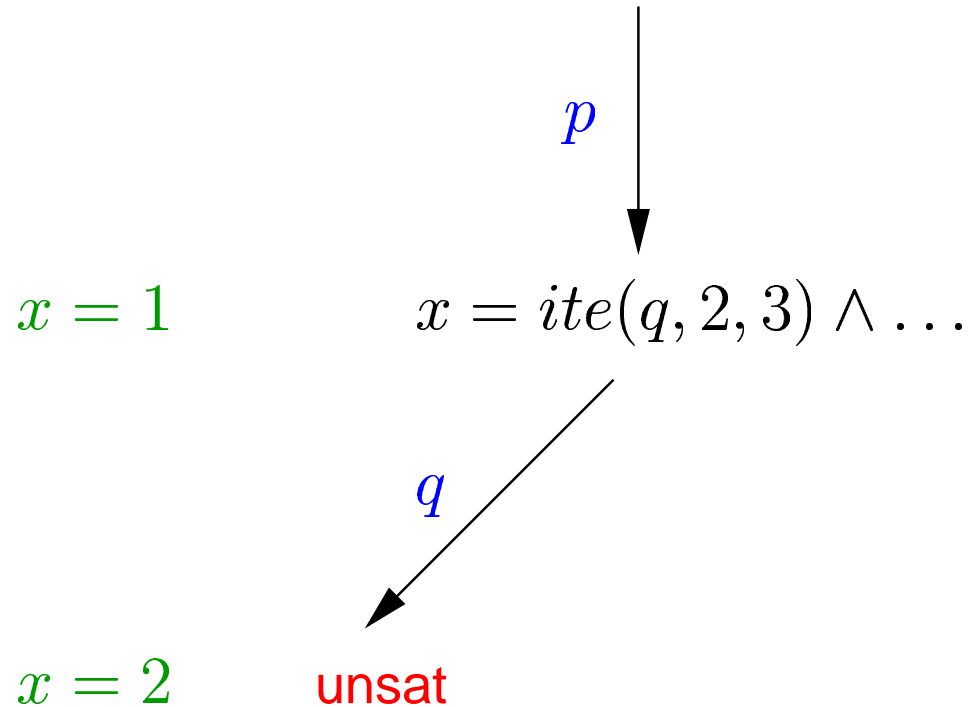


$$x = 1$$

$$x = ite(q, 2, 3) \wedge \dots$$

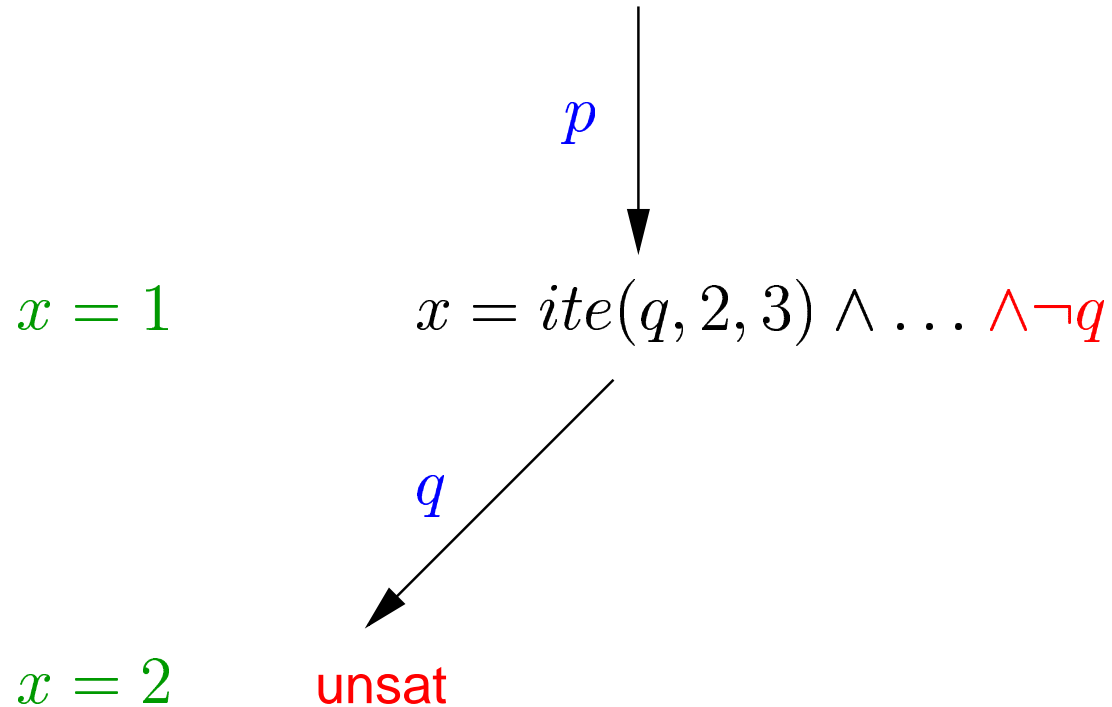
Combining caching heuristic with SAT methods

$$(p \rightarrow x = 1) \wedge x = ite(q, 2, 3) \wedge \dots$$



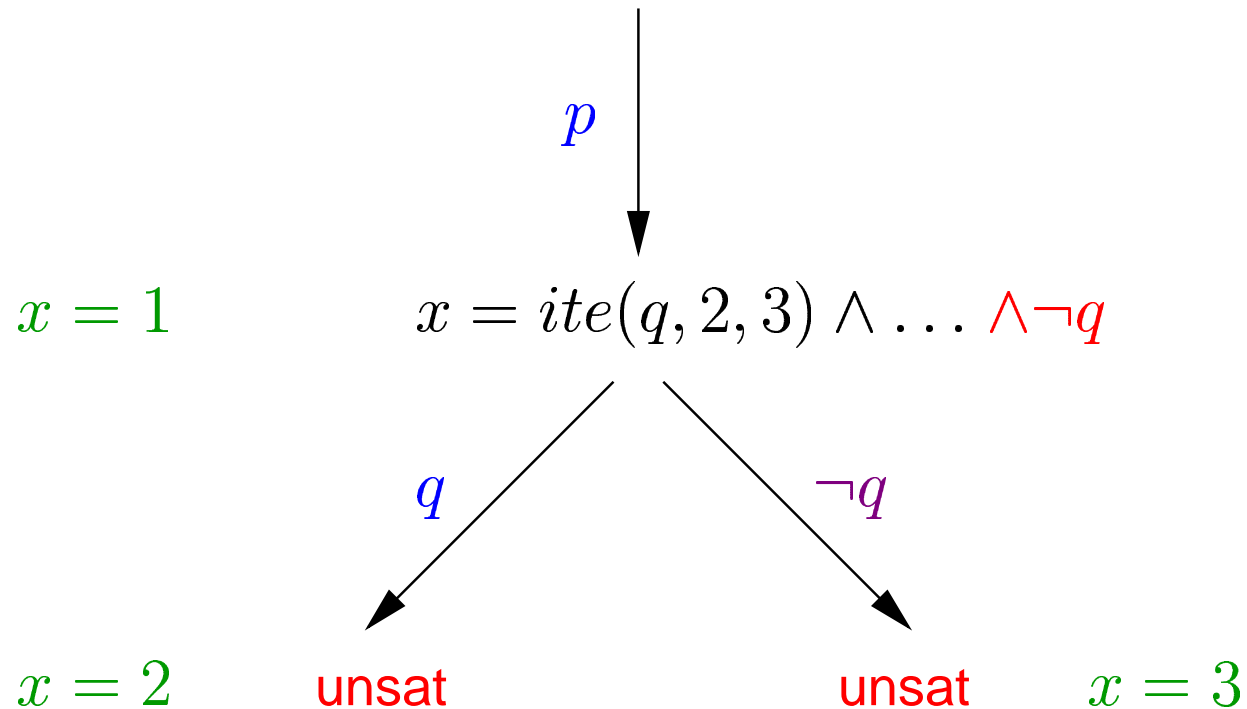
Combining caching heuristic with SAT methods

$$(p \rightarrow x = 1) \wedge x = \text{ite}(q, 2, 3) \wedge \dots \wedge (\neg p \vee \neg q)$$



Combining caching heuristic with SAT methods

$$(p \rightarrow x = 1) \wedge x = \text{ite}(q, 2, 3) \wedge \dots \wedge (\neg p \vee \neg q)$$



Conversion to CNF

One of the benefits of converting a formula to CNF is that you effectively get propagation both up and down the original formula tree.

By not converting to CNF, we lose some of this propagation.

A solution is to do both: keep the original formula, but also convert it to CNF and use those clauses simply for propagating information.

Results

Simple		no BCP, no conflict clauses
Fast		BCP, conflict clauses

Results

Simple		no BCP, no conflict clauses
Fast		BCP, conflict clauses

Decisions	
Simple DFS	1.00
Fast DFS	0.47
Simple caching	0.41
Fast DFS + CNF	0.10
Fast caching + CNF	0.04

Results

Simple		no BCP, no conflict clauses
Fast		BCP, conflict clauses

Decisions	
Simple DFS	1.00
Fast DFS	0.47
Simple caching	0.41
Fast DFS + CNF	0.10
Fast caching + CNF	0.04

Time	
Fast DFS	1.38
Fast DFS + CNF	1.08
Simple DFS	1.00
Fast caching + CNF	0.55
Simple caching	0.53

Outline

- Nelson-Oppen in CVC Lite
- Leveraging Proofs
- Hybrid SAT Methods
- *Partial Functions*

Motivation

FOL and Partial Functions

- *First Order Logic (FOL)* is a powerful tool for modeling computer systems.
- Standard models of *FOL* assume that all functions are total.
- However, many applications are more naturally modeled using partial functions.

Motivation

FOL and Partial Functions

- *First Order Logic (FOL)* is a powerful tool for modeling computer systems.
- Standard models of *FOL* assume that all functions are total.
- However, many applications are more naturally modeled using partial functions.

Formalizing Partial Functions

- A variety of approaches have been proposed for accommodating partial functions.
- In order to be clear and rigorous, it is desirable to develop a semantics that allows *undefined* values. There are two main approaches:
 - Allow terms to be undefined, but require formulas to be either true or false.
 - Allow both terms and formulas to be undefined.
- We chose the second approach based on a *principle of least surprise*: *a formula should be valid only if there is no disagreement on whether it should be a valid formula.*

Three-Valued Logic: Syntax

Let $\Sigma = (S, F, P, C, \Delta)$ be a signature, where

- $S = \{s_1, \dots\}$ is a set of sort symbols,
- $F = \{f_1, \dots\}$ is a set of function symbols,
- $P = \{p_1, \dots\}$ is a set of predicate symbols,
- $C = \{c_1, \dots\}$ is a set of constant symbols, and
- Δ is a set of *domain formulas* which we will explain shortly.

Three-Valued Logic: Syntax

Let $\Sigma = (S, F, P, C, \Delta)$ be a signature, where

- $S = \{s_1, \dots\}$ is a set of sort symbols,
- $F = \{f_1, \dots\}$ is a set of function symbols,
- $P = \{p_1, \dots\}$ is a set of predicate symbols,
- $C = \{c_1, \dots\}$ is a set of constant symbols, and
- Δ is a set of *domain formulas* which we will explain shortly.

Each symbol in F , P , and C has a *type* built out of the sort symbols in S . Terms and formulas are defined in a standard way:

$$\begin{aligned} t & ::= x \mid c \mid f(t_1, \dots, t_n) \mid \mathbf{ite}(\phi, t_1, t_2), \\ \phi & ::= \mathbf{true} \mid \mathbf{false} \mid p(t_1, \dots, t_n) \mid t_1 = t_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi_1 \mid \\ & \quad \mathbf{ite}(\phi_0, \phi_1, \phi_2) \mid \exists x : s. \phi_1. \end{aligned}$$

Three-Valued Logic: Syntax

Let $\Sigma = (S, F, P, C, \Delta)$ be a signature, where

- $S = \{s_1, \dots\}$ is a set of sort symbols,
- $F = \{f_1, \dots\}$ is a set of function symbols,
- $P = \{p_1, \dots\}$ is a set of predicate symbols,
- $C = \{c_1, \dots\}$ is a set of constant symbols, and
- Δ is a set of *domain formulas* which we will explain shortly.

Each symbol in F , P , and C has a *type* built out of the sort symbols in S . Terms and formulas are defined in a standard way:

$$\begin{aligned} t & ::= x \mid c \mid f(t_1, \dots, t_n) \mid \mathbf{ite}(\phi, t_1, t_2), \\ \phi & ::= \mathbf{true} \mid \mathbf{false} \mid p(t_1, \dots, t_n) \mid t_1 = t_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \mid \\ & \quad \mathbf{ite}(\phi_0, \phi_1, \phi_2) \mid \exists x : s. \phi_1. \end{aligned}$$

Other operators are considered to be abbreviations in the usual way: $\phi_1 \wedge \phi_2$, $\phi_1 \Rightarrow \phi_2$, $\phi_1 \Leftrightarrow \phi_2$, and $\forall x : s. \phi$.

Domain Formulas

The set Δ of *domain formulas* contains a formula for each function and predicate symbol in Σ .

- The domain formula for a *function symbol* f is a Σ -formula with k free variables where k is the arity of f and is denoted $\delta_f[x_1, \dots, x_k]$.
- The domain formula for a *predicate symbol* p of arity k is defined similarly and is denoted $\delta_p[x_1, \dots, x_k]$.
- An *instantiation* of a domain formula δ_f with terms t_1, \dots, t_k is written $\delta_f[t_1, \dots, t_k]$ and denotes the result of replacing each x_i with t_i in the domain formula $\delta_f[x_1, \dots, x_k]$.

Domain Formulas

The set Δ of *domain formulas* contains a formula for each function and predicate symbol in Σ .

- The domain formula for a *function symbol* f is a Σ -formula with k free variables where k is the arity of f and is denoted $\delta_f[x_1, \dots, x_k]$.
- The domain formula for a *predicate symbol* p of arity k is defined similarly and is denoted $\delta_p[x_1, \dots, x_k]$.
- An *instantiation* of a domain formula δ_f with terms t_1, \dots, t_k is written $\delta_f[t_1, \dots, t_k]$ and denotes the result of replacing each x_i with t_i in the domain formula $\delta_f[x_1, \dots, x_k]$.

Intuitively, the domain formula for f defines the *set of points where f is defined*.

Domain Formulas

The set Δ of *domain formulas* contains a formula for each function and predicate symbol in Σ .

- The domain formula for a *function symbol* f is a Σ -formula with k free variables where k is the arity of f and is denoted $\delta_f[x_1, \dots, x_k]$.
- The domain formula for a *predicate symbol* p of arity k is defined similarly and is denoted $\delta_p[x_1, \dots, x_k]$.
- An *instantiation* of a domain formula δ_f with terms t_1, \dots, t_k is written $\delta_f[t_1, \dots, t_k]$ and denotes the result of replacing each x_i with t_i in the domain formula $\delta_f[x_1, \dots, x_k]$.

Intuitively, the domain formula for f defines the *set of points where f is defined*.

Note that this assumes the set is always *first-order definable*. Fortunately, this does not seem to be a serious restriction for practical applications.

Domain Formulas

The set Δ of *domain formulas* contains a formula for each function and predicate symbol in Σ .

- The domain formula for a *function symbol* f is a Σ -formula with k free variables where k is the arity of f and is denoted $\delta_f[x_1, \dots, x_k]$.
- The domain formula for a *predicate symbol* p of arity k is defined similarly and is denoted $\delta_p[x_1, \dots, x_k]$.
- An *instantiation* of a domain formula δ_f with terms t_1, \dots, t_k is written $\delta_f[t_1, \dots, t_k]$ and denotes the result of replacing each x_i with t_i in the domain formula $\delta_f[x_1, \dots, x_k]$.

Intuitively, the domain formula for f defines the *set of points where f is defined*.

Note that this assumes the set is always *first-order definable*. Fortunately, this does not seem to be a serious restriction for practical applications.

In order to have an unambiguous semantics, *it is important that the domain formulas themselves always be defined*. One simple way to ensure this is to require that if s is a function or predicate symbol appearing in a domain formula, then $\delta_s[x_1, \dots, x_n] = \text{true}$.

Example

A signature $\Sigma = (S, F, P, C, \Delta)$ for arithmetic in which division by 0 is undefined is given below:

- $S = \{Q\}$
- $F = \{+, -, \times, /\}$
- $P = \{<, >\}$
- $C = \{0, 1, \dots\}$
- $\delta_+ = \delta_- = \delta_\times = \delta_< = \delta_> = \text{true}$
- $\delta_/[x_1, x_2] = x_2 \neq 0$

Three-Valued Logic: Semantics

Given a signature $\Sigma = (S, F, P, C, \Delta)$, a model is a pair $M = \langle A, I \rangle$, where

- A is an S -indexed family of nonempty carrier sets $A = \{A_s \mid s \in S\}$ and
- I is an *interpretation*, which is a mapping:
 - from constant symbols $c : s$ to elements $c^M \in A_s$,
 - from function symbols $f : s_1 \times \cdots \times s_n \rightarrow s$ to *partial functions*
 $f^M : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$
 - from predicate symbols $p : s_1 \times \cdots \times s_n$ to relations
 $p^M \subseteq A_{s_1} \times \cdots \times A_{s_n}$.

Three-Valued Logic: Semantics

Given a signature $\Sigma = (S, F, P, C, \Delta)$, a model is a pair $M = \langle A, I \rangle$, where

- A is an S -indexed family of nonempty carrier sets $A = \{A_s \mid s \in S\}$ and
- I is an *interpretation*, which is a mapping:
 - from constant symbols $c : s$ to elements $c^M \in A_s$,
 - from function symbols $f : s_1 \times \cdots \times s_n \rightarrow s$ to *partial functions*
 $f^M : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$
 - from predicate symbols $p : s_1 \times \cdots \times s_n$ to relations
 $p^M \subseteq A_{s_1} \times \cdots \times A_{s_n}$.

Given a model M and a variable assignment e which maps each variable to an element of some A_s , the value of an expression (a term or a formula) α is denoted $\llbracket \alpha \rrbracket_{Me}$.

Three-Valued Logic: Semantics

Given a signature $\Sigma = (S, F, P, C, \Delta)$, a model is a pair $M = \langle A, I \rangle$, where

- A is an S -indexed family of nonempty carrier sets $A = \{A_s \mid s \in S\}$ and
- I is an *interpretation*, which is a mapping:
 - from constant symbols $c : s$ to elements $c^M \in A_s$,
 - from function symbols $f : s_1 \times \cdots \times s_n \rightarrow s$ to *partial functions*
 $f^M : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$
 - from predicate symbols $p : s_1 \times \cdots \times s_n$ to relations
 $p^M \subseteq A_{s_1} \times \cdots \times A_{s_n}$.

Given a model M and a variable assignment e which maps each variable to an element of some A_s , the value of an expression (a term or a formula) α is denoted $\llbracket \alpha \rrbracket_{Me}$.

A model is required to satisfy the following additional condition imposed by the domain formulas Δ :

$\llbracket \delta_f[x_1, \dots, x_k] \rrbracket_{Me} = \text{true}$ iff f^M is defined at $(\llbracket x_1 \rrbracket_{Me}, \dots, \llbracket x_k \rrbracket_{Me})$.

Three-Valued Logic: Semantics

Semantics are defined in the usual bottom-up way with the following exceptions:

- When evaluating a function or predicate, the domain formula is first evaluated for the same arguments. If the domain formula is true, the evaluation proceeds normally. Otherwise, the value is \perp .
- For all operators other than \vee and **ite**, if any child is \perp , then the value is \perp .
- \vee and **ite** are non-strict, meaning that sometimes they can be evaluated even if one of their children is \perp :

$$\begin{aligned} \llbracket \phi_1 \vee \phi_2 \rrbracket_{Me} &= \begin{cases} \text{true}, & \text{if } \llbracket \phi_1 \rrbracket_{Me} = \text{true} \text{ or } \llbracket \phi_2 \rrbracket_{Me} = \text{true}; \\ \text{false} & \text{if } \llbracket \phi_1 \rrbracket_{Me} = \text{false} \text{ and } \llbracket \phi_2 \rrbracket_{Me} = \text{false}; \\ \perp & \text{otherwise.} \end{cases} \\ \llbracket \text{ite}(\phi, \alpha, \beta) \rrbracket_{Me} &= \begin{cases} \perp, & \text{if } \llbracket \phi \rrbracket_{Me} = \perp; \\ \llbracket \alpha \rrbracket_{Me}, & \text{if } \llbracket \phi \rrbracket_{Me} = \text{true}; \\ \llbracket \beta \rrbracket_{Me}, & \text{if } \llbracket \phi \rrbracket_{Me} = \text{false}. \end{cases} \end{aligned}$$

Three-Valued Validity

Under the three-valued semantics, a formula ϕ is

- *valid* if in all models M and for all variable assignments e , $\llbracket \phi \rrbracket_{Me} = \text{true}$.
- *invalid* if there is at least one model M and variable assignmente such that $\llbracket \phi \rrbracket_{Me} = \text{false}$.
- Otherwise, (ϕ always evaluates to \perp or *true*), the validity is *undefined*.

Reduction from Three-Valued to Two-Valued Logic

Suppose we wish to determine the three-valued validity of a formula ϕ . We will show how this can be done using only standard two-valued first order logic.

The key idea is to check a separate formula which determines whether ϕ can be undefined. This formula is called a *type correctness condition (TCC)*.

The TCC for an expression α is denoted \mathcal{D}_α and is computed as follows:

$$\begin{aligned}\mathcal{D}_x &\equiv \mathcal{D}_c \equiv \text{true} \\ \mathcal{D}_{f(t_1, \dots, t_n)} &\equiv \delta_f[t_1, \dots, t_n] \wedge \bigwedge_{i=1}^n \mathcal{D}_{t_i} \\ \mathcal{D}_{p(t_1, \dots, t_n)} &\equiv \delta_p[t_1, \dots, t_n] \wedge \bigwedge_{i=1}^n \mathcal{D}_{t_i} \\ \mathcal{D}_{\text{ite}(\phi, \alpha, \beta)} &\equiv \mathcal{D}_\phi \wedge (\text{ite}(\phi, \mathcal{D}_\alpha, \mathcal{D}_\beta)) \\ \mathcal{D}_{t_1=t_2} &\equiv \mathcal{D}_{t_1} \wedge \mathcal{D}_{t_2} \\ \mathcal{D}_{\neg\phi} &\equiv \mathcal{D}_\phi \\ \mathcal{D}_{\phi_1 \vee \phi_2} &\equiv (\mathcal{D}_{\phi_1} \wedge \phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2}) \\ \mathcal{D}_{\exists x. \phi} &\equiv (\exists x. \mathcal{D}_\phi \wedge \phi) \vee (\forall x. \mathcal{D}_\phi)\end{aligned}$$

Reduction from Three-Valued to Two-Valued Logic

Suppose we have a signature Σ . Let $\hat{\Sigma}$ be equivalent to Σ except that all of its domain formulas are *true*. We will call such a signature a *total* signature and a corresponding model a *total* model.

Let M be a model of Σ and let \hat{M} be a (total) model of $\hat{\Sigma}$ whose interpretation of function and predicate symbols agrees with M wherever the domain formulas of M are true. We call \hat{M} an *extension* of M .

Finally, let $\llbracket S \rrbracket_{\hat{M}}^2 e$ denote the evaluation of an expression S in the model \hat{M} using standard two-valued semantics.

Theorem

Let S be any Σ -term or formula, and let \hat{M} denote an arbitrary extension of a Σ -model M to a total model over $\hat{\Sigma}$. Then,

$$\begin{aligned} \llbracket \mathcal{D}_S \rrbracket_{\hat{M}}^2 e = \text{true} &\Rightarrow \llbracket S \rrbracket_{\hat{M}}^2 e = \llbracket S \rrbracket_M e. \\ \llbracket \mathcal{D}_S \rrbracket_{\hat{M}}^2 e = \text{false} &\Rightarrow \llbracket S \rrbracket_M e = \perp. \end{aligned}$$

Corollary

If ϕ is a Σ -formula, then $\mathcal{D}_\phi \wedge \phi$ is (two-valued) valid iff ϕ is three-valued valid.

Reduction from Three-Valued to Two-Valued Logic

Another important property of \mathcal{D}_ϕ is that if ϕ is represented as a *DAG*, then the worst-case size of $\mathcal{D}_\phi \wedge \phi$ is linear in the size of ϕ .

This is because at each step of the computation of \mathcal{D}_ϕ , only a constant number of additional nodes are introduced in addition to those already in ϕ .

This is critical for many applications where the size of ϕ may be very large.

Sources

C. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD Thesis. Stanford University, 2003.

S. McLaughlin and C. Barrett. *Cooperating Theorem Provers: A Case Study Combining HOL Light and CVC Lite*. Unpublished, 2003.

C. Barrett and J. Donham. *Combining SAT Methods with Non-Clausal Decision Heuristics*. PDPAR 2004.

S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. Dill. *A practical approach to partial functions in CVC Lite*. PDPAR 2004.

This research was supported in part by an Intel grant for “Cooperating Decision Procedures in Formal Verification of Hardware”.