Formalizing the Logic-Automaton Connection

Stefan Berghofer  Markus Reiter

Institut für Informatik
Technische Universität München

TPHOLs, 20.8.2009
Which of these formulae are true (for natural numbers)?

\[ \forall x \geq 7. \ \exists y \ z. \ 3 \cdot y + 5 \cdot z = x \]

\[ \forall x \geq 8. \ \exists y \ z. \ 3 \cdot y + 5 \cdot z = x \]

\[ \forall x \geq 8. \ \exists y \ z. \ 4 \cdot y + 5 \cdot z = x \]
Which of these formulae are true (for natural numbers)?

\[ \forall x \geq 7. \ \exists y, z. \ 3 \cdot y + 5 \cdot z = x \]

\[ \forall x \geq 8. \ \exists y, z. \ 3 \cdot y + 5 \cdot z = x \]

\[ \forall x \geq 8. \ \exists y, z. \ 4 \cdot y + 5 \cdot z = x \]
Which of these formulae are true (for natural numbers)?

\[ \forall x \geq 7. \ \exists y \ z. \ 3 \ast y + 5 \ast z = x \]

\[ \forall x \geq 8. \ \exists y \ z. \ 3 \ast y + 5 \ast z = x \]

\[ \forall x \geq 8. \ \exists y \ z. \ 4 \ast y + 5 \ast z = x \]

Stamp problem

Any postage of 8 cents or more can be made up using stamps of the denominations 3 cents and 5 cents.
Outline

1. Introduction
2. Basic Concepts
3. Automata Construction
4. The Decision Procedure
5. Conclusion
1. Introduction

2. Basic Concepts

3. Automata Construction

4. The Decision Procedure

5. Conclusion
Quantifier elimination method
Quantifier elimination method

Algebraic e.g. Cooper’s algorithm
Quantifier elimination method

Algebraic  e.g. Cooper’s algorithm
Semantic  e.g. using automata on bitstrings (this talk)
          (also works for WS1S, see \texttt{Mona})
**Quantifier elimination method**

**Algebraic**  e.g. Cooper’s algorithm

**Semantic**  e.g. using automata on bitstrings *(this talk)*

(also works for WS1S, see **Mona**)

**Implementation in a theorem prover**
Decision Procedures for Presburger Arithmetic

**Quantifier elimination method**

*Algebraic* e.g. Cooper’s algorithm  
*Semantic* e.g. using automata on bitstrings (*this talk*)  
(also works for WS1S, see \texttt{Mona})

**Implementation in a theorem prover**

*Oracle-based* Use an external tool such as \texttt{Mona}, and simply trust the answer of the tool.
Decision Procedures for Presburger Arithmetic

Quantifier elimination method

Algebraic e.g. Cooper’s algorithm
Semantic e.g. using automata on bitstrings (this talk)
(also works for WS1S, see Mona)

Implementation in a theorem prover

Oracle-based Use an external tool such as Mona, and simply trust the answer of the tool.
Certificate-based Use an external tool, but try to reconstruct a proof inside the theorem prover from a certificate (or trace) returned by the tool, rather than just trusting it.
Quantifier elimination method

Algebraic  e.g. Cooper’s algorithm
Semantic  e.g. using automata on bitstrings (this talk)
          (also works for WS1S, see \textsc{Mona})

Implementation in a theorem prover

Oracle-based  Use an external tool such as \textsc{Mona}, and simply trust the answer of the tool.
Certificate-based  Use an external tool, but try to reconstruct a proof inside the theorem prover from a certificate (or trace) returned by the tool, rather than just trusting it.
Derived rule  Write a decision procedure in the implementation language of the theorem prover (e.g. ML or OCaml) that constructs a proof by applying primitive inference rules.
Decision Procedures for Presburger Arithmetic

Quantifier elimination method

Algebraic  e.g. Cooper’s algorithm
Semantic  e.g. using automata on bitstrings \((\text{this talk})\)
(also works for WS1S, see \textit{Mona})

Implementation in a theorem prover

Oracle-based  Use an external tool such as \textit{Mona}, and simply trust the answer of the tool.
Certificate-based  Use an external tool, but try to \textit{reconstruct} a proof inside the theorem prover from a \textit{certificate} (or \textit{trace}) returned by the tool, rather than just trusting it.
Derived rule  Write a decision procedure in the implementation language of the theorem prover (e.g. ML or OCaml) that constructs a proof by applying \textit{primitive inference rules}.
Reflection  Write \textit{and verify} the decision procedure as a recursive function in HOL itself \((\text{this talk})\).
Related Work


[Norris, TPHOLs 2003] Cooper’s algorithm in HOL (derived rule)

[Norrish, TPHOLs 2003] Cooper’s algorithm in HOL (derived rule)

[Chaieb and Nipkow, JAR 2008] Cooper’s / Ferrante and Rackoff’s algorithm in Isabelle (reflection / derived rule)

[Norrish, TPHOLs 2003] Cooper’s algorithm in HOL (derived rule)

[Chaieb and Nipkow, JAR 2008] Cooper’s / Ferrante and Rackoff’s algorithm in Isabelle (reflection / derived rule)

[Nipkow, IJCAR 2008] Quantifier elimination for discrete linear orders, linear real, and Presburger arithmetic (reflection)

[Norrish, TPHOLs 2003] Cooper’s algorithm in HOL (derived rule)

[Chaieb and Nipkow, JAR 2008] Cooper’s / Ferrante and Rackoff’s algorithm in Isabelle (reflection / derived rule)

[Nipkow, IJCAR 2008] Quantifier elimination for discrete linear orders, linear real, and Presburger arithmetic (reflection)

Hooking an ‘oracle’ to a theorem prover is risky business. The oracle could be buggy [...]. The only way to avoid a buggy oracle is to reconstruct a proof in the theorem prover based on output from the oracle, or perhaps verify the oracle itself. For a semantics based decision procedure, proof reconstruction is not a realistic option: one would have to formalize the entire automata-theoretic machinery within HOL [...].

[Basin and Friedrich, FroCoS 2000]
Does Reflection work?

Hooking an ‘oracle’ to a theorem prover is risky business. The oracle could be buggy [...]. The only way to avoid a buggy oracle is to reconstruct a proof in the theorem prover based on output from the oracle, or perhaps verify the oracle itself. For a semantics based decision procedure, proof reconstruction is not a realistic option: one would have to formalize the entire automata-theoretic machinery within HOL [...].

[Basin and Friedrich, FroCoS 2000]

Our Claim

Verifying automata-based decision procedures in HOL is not as unrealistic as it may seem!
Syntax (using de Bruijn indices)

\[
\text{datatype } \textit{pf} = \text{Eq (int list) int} \mid \text{Le (int list) int} \mid \text{And pf pf} \\
\quad \mid \text{Or pf pf} \mid \text{Imp pf pf} \mid \text{Forall pf} \mid \text{Exist pf} \mid \text{Neg pf}
\]
Syntax (using de Bruijn indices)

```
datatype pf = Eq (int list) int | Le (int list) int | And pf pf
    | Or pf pf | Imp pf pf | Forall pf | Exist pf | Neg pf
```

Example: Stamp problem

```
∀ x ≥ 8. ∃ y z. 3 * y + 5 * z = x
```
Presburger Arithmetic

Syntax (using de Bruijn indices)

```plaintext
datatype pf = Eq (int list) int | Le (int list) int | And pf pf
| Or pf pf | Imp pf pf | Forall pf | Exist pf | Neg pf
```

Example: Stamp problem

```plaintext
\forall x \geq 8. \exists y z. 3 \ast y + 5 \ast z = x
```

Encoding

```plaintext
Forall (Imp (Le [-1] -8) (Exist (Exist (Eq [5, 3, -1] 0))))
```
Diophantine (In)Equations

eval-dioph :: int list ⇒ nat list ⇒ int

eval-dioph (k · ks) (x · xs) = k * int x + eval-dioph ks xs

eval-dioph [] xs = 0

eval-dioph ks [] = 0
Evaluation

Diophantine (In)Equations

eval-dioph :: int list ⇒ nat list ⇒ int
eval-dioph (k · ks) (x · xs) = k * int x + eval-dioph ks xs
eval-dioph [] xs = 0
eval-dioph ks [] = 0

Formulae

eval-pf :: pf ⇒ nat list ⇒ bool
eval-pf (Eq ks l) xs = (eval-dioph ks xs = l)
eval-pf (Le ks l) xs = (eval-dioph ks xs ≤ l)
eval-pf (Neg p) xs = (¬ eval-pf p xs)
eval-pf (And p q) xs = (eval-pf p xs ∧ eval-pf q xs)
eval-pf (Or p q) xs = (eval-pf p xs ∨ eval-pf q xs)
eval-pf (Forall p) xs = (∀ x. eval-pf p (x · xs))
eval-pf (Exist p) xs = (∃ x. eval-pf p (x · xs))
Input symbols of an automaton corresponding to a formula with $n$ free variables $x_0, \ldots, x_{n-1}$ are bit lists of length $n$:

$$
\begin{align*}
  x_0 & \quad \begin{bmatrix} \vdots \ldots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{0,0} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{0,j} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{0,m-1} \vdots \vdots \vdots \end{bmatrix} \\
  \vdots & \quad \begin{bmatrix} \vdots \ldots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{i,0} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{i,j} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{i,m-1} \vdots \vdots \vdots \end{bmatrix} \\
  x_i & \quad \begin{bmatrix} \vdots \ldots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,0} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,j} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,m-1} \vdots \vdots \vdots \end{bmatrix} \\
  \vdots & \quad \begin{bmatrix} \vdots \ldots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,0} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,j} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,m-1} \vdots \vdots \vdots \end{bmatrix} \\
  x_{n-1} & \quad \begin{bmatrix} \vdots \ldots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,0} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,j} \vdots \vdots \vdots \end{bmatrix} \quad \begin{bmatrix} b_{n-1,m-1} \vdots \vdots \vdots \end{bmatrix}
\end{align*}
$$
Automata on Bit Vectors

Input symbols of an automaton corresponding to a formula with \( n \) free variables \( x_0, \ldots, x_{n-1} \) are bit lists of length \( n \):

\[
\begin{align*}
&x_0 \quad \begin{bmatrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{bmatrix} \quad \begin{bmatrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{bmatrix} \quad \begin{bmatrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{bmatrix} \\
&x_i 
\end{align*}
\]

- \( i \)-th row: value of \( i \)-th variable (natural number)
Input symbols of an automaton corresponding to a formula with $n$ free variables $x_0, \ldots, x_{n-1}$ are bit lists of length $n$:

$$
\begin{align*}
x_0 & \ \begin{bmatrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{bmatrix} & \begin{bmatrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{bmatrix} & \begin{bmatrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{bmatrix} \\
\end{align*}
$$

- $i$-th row: value of $i$-th variable (natural number)
- $j$-th column: $j$-th bit of variables
Input symbols of an automaton corresponding to a formula with \( n \) free variables \( x_0, \ldots, x_{n-1} \) are bit lists of length \( n \):

\[
\begin{align*}
  x_0 & \quad \begin{bmatrix} b_{0,0} \\ \vdots \\ b_{i,0} \end{bmatrix} \\  \vdots & \quad \vdots \\  x_i & \quad \begin{bmatrix} b_{0,j} \\ \vdots \\ b_{i,j} \end{bmatrix} \\  \vdots & \quad \vdots \\  x_{n-1} & \quad \begin{bmatrix} b_{0,m-1} \\ \vdots \\ b_{n-1,j} \\ \vdots \\ b_{n-1,m-1} \end{bmatrix}
\end{align*}
\]

- \( i \)-th row: value of \( i \)-th variable (natural number)
- \( j \)-th column: \( j \)-th bit of variables
- column 0: least significant bit
Input symbols of an automaton corresponding to a formula with \( n \) free variables \( x_0, \ldots, x_{n-1} \) are bit lists of length \( n \):

\[
\begin{array}{cccc}
  x_0 & \left[ \begin{array}{c} b_{0,0} \\ \vdots \\ b_{i,0} \\
  \vdots \\
  b_{n-1,0} \end{array} \right] & \cdots & \left[ \begin{array}{c} b_{0,j} \\ \vdots \\ b_{i,j} \\
  \vdots \\
  b_{n-1,j} \end{array} \right] & \cdots & \left[ \begin{array}{c} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\
  \vdots \\
  b_{n-1,m-1} \end{array} \right] \\
  \end{array}
\]

- \( i \)-th row: value of \( i \)-th variable (natural number)
- \( j \)-th column: \( j \)-th bit of variables
- column 0: least significant bit

[Boudet and Comon, CAAP 1996]
List of variables (encoded as list of column vectors)

\[
nats-of-boolss :: nat \Rightarrow bool \ list \ list \Rightarrow nat \ list
\]
\[
nats-of-boolss \ n \ [] = replicate \ n \ 0
\]
\[
nats-of-boolss \ n \ (bs \cdot bss) =
\]
\[
map (\lambda (b, x). \ nat-of-bool b + 2 \ast x)\]
\[
(\ zip \ bs \ (nats-of-boolss \ n \ bss))
\]
**Values of Variables**

**List of variables (encoded as list of column vectors)**

\[ \text{nats-of-bools} :: \text{nat} \Rightarrow \text{bool list list} \Rightarrow \text{nat list} \]

\[ \text{nats-of-bools} \ n \ [] = \text{replicate} \ n \ 0 \]

\[ \text{nats-of-bools} \ n \ (\text{bs} \cdot \text{bss}) = \]

\[ \text{map} \ (\lambda (b, x). \text{nat-of-bool} \ b + 2 \ast x) \]

\[ (\text{zip} \ \text{bs} \ (\text{nats-of-bools} \ n \ \text{bss})) \]

**Single variable (encoded as row vector)**

\[ \text{nat-of-bools} :: \text{bool list} \Rightarrow \text{nat} \]

\[ \text{nat-of-bools} \ [] = 0 \]

\[ \text{nat-of-bools} \ (b \cdot \text{bs}) = \text{nat-of-bool} \ b + 2 \ast \text{nat-of-bools} \ \text{bs} \]
Deterministic Automata

Represented by type

\[
dfa = \text{nat bdd list} \times \text{bool list}
\]

transition table \hspace{1cm} accepting states

Note:
start state = 0

Transition table
Deterministic Automata

Represented by type

\[ dfa = \text{nat bdd list} \times \text{bool list} \]

transition table \hspace{5em} accepting states

Note: start state = 0
Deterministic Automata

Represented by type

\[
dfa = \text{nat list} \times \text{bool list}
\]

transition table \hspace{1cm} accepting states

Note: start state = 0

Transition table
Represented by type

\[ nfa = \text{bool list bdd list} \times \text{bool list} \]

transition table accepting states
Represented by type

\[ nfa = \text{bool list bdd list} \times \text{bool list} \]

\underline{transition table} \quad \underline{accepting states}

Note: (finite) sets of states represented as bitstrings
Outline

1. Introduction
2. Basic Concepts
3. Automata Construction
4. The Decision Procedure
5. Conclusion
Automata Construction

- Complement: for negation
Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\lor$, $\land$, and $\rightarrow$
Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\lor$, $\land$, and $\rightarrow$
- Projection: for existential quantifiers
Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\lor$, $\land$, and $\rightarrow$
- Projection: for existential quantifiers
- Atomic formulae: Diophantine (in)equations
Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\lor$, $\land$, and $\rightarrow$
- Projection: for existential quantifiers
- Atomic formulae: Diophantine (in)equations

**Naive implementation**

- Product automaton: $m \cdot n$ states
Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\lor$, $\land$, and $\rightarrow$
- Projection: for existential quantifiers
- Atomic formulae: Diophantine (in)equations

**Naive implementation**

- Product automaton: $m \cdot n$ states
- DFA from NFA: $2^n$ states
Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\lor$, $\land$, and $\rightarrow$
- Projection: for existential quantifiers
- Atomic formulae: Diophantine (in)equations

**Naive implementation**

- Product automaton: $m \cdot n$ states
- DFA from NFA: $2^n$ states

**Better:** only generate reachable states (using DFS)
Existential quantifiers

- Convert DFA to NFA (trivial)
Existential quantifiers

- Convert DFA to NFA (trivial)
- Project away variable(s) to be quantified (yields NFA)
Existential quantifiers

- Convert DFA to NFA (trivial)
- Project away variable(s) to be quantified (yields NFA)
- Convert NFA to DFA (subset construction)
Existential quantifiers
- Convert DFA to NFA (trivial)
- Project away variable(s) to be quantified (yields NFA)
- Convert NFA to DFA (subset construction)

Universal quantifiers
Note: \( (\forall x. P x) = (\neg (\exists x. \neg P x)) \)
quantify-bdd :: nat ⇒ bool list bdd ⇒ bool list bdd
quantify-bdd i (Leaf q) = Leaf q
quantify-bdd 0 (Branch l r) = bdd-binop bv-or l r
quantify-bdd (Suc i) (Branch l r) =
    Branch (quantify-bdd i l) (quantify-bdd i r)
Projection

quantify-bdd :: nat ⇒ bool list bdd ⇒ bool list bdd
quantify-bdd i (Leaf q) = Leaf q
quantify-bdd 0 (Branch l r) = bdd-binop bv-or l r
quantify-bdd (Suc i) (Branch l r) =
    Branch (quantify-bdd i l) (quantify-bdd i r)
Projection

\[ L_1 \cup L_2 \quad R_1 \cup R_2 \quad L_3 \cup L_4 \quad R_3 \cup R_4 \]

**quantify-bdd :: nat ⇒ bool list bdd ⇒ bool list bdd**

\[
\begin{align*}
\text{quantify-bdd } i \ (\text{Leaf } q) &= \text{Leaf } q \\
\text{quantify-bdd } 0 \ (\text{Branch } l \ r) &= \text{bdd-binop } \text{bv-or } l \ r \\
\text{quantify-bdd } (\text{Suc } i) \ (\text{Branch } l \ r) &= \text{Branch } (\text{quantify-bdd } i \ l) \ (\text{quantify-bdd } i \ r)
\end{align*}
\]
Method by Boudet and Comon

\[
\begin{align*}
\text{eval-dioph } ks \ x s &= l \\
\text{eval-dioph } ks \ (\text{map } (\lambda x. x \mod 2) \ x s) \mod 2 &= l \mod 2 \land \\
\text{eval-dioph } ks \ (\text{map } (\lambda x. x \div 2) \ x s) &= \\
(1 - \text{eval-dioph } ks \ (\text{map } (\lambda x. x \mod 2) \ x s)) \div 2
\end{align*}
\]
Diophantine Equations

Method by Boudet and Comon

\[(\text{eval-dioph } ks \; xs = l) = \]
\[(\text{eval-dioph } ks \; (\text{map } (\lambda x. \; x \mod 2) \; xs) \; \mod 2 = l \; \mod 2 \land \]
\[\text{eval-dioph } ks \; (\text{map } (\lambda x. \; x \div 2) \; xs) = \]
\[(l - \text{eval-dioph } ks \; (\text{map } (\lambda x. \; x \mod 2) \; xs)) \; \div 2)\]

xs is a solution iff...

- ...it is a solution modulo 2, and...
Diophantine Equations

Method by Boudet and Comon

\[
\text{(eval-dioph } ks \ x s = l) = \\
\text{(eval-dioph } ks (\text{map (} \lambda x. x \ \text{mod} \ 2) \ x s) \ \text{mod} \ 2 = l \ \text{mod} \ 2 \ \land \\
\text{eval-dioph } ks (\text{map (} \lambda x. x \ \text{div} \ 2) \ x s) = \\
(l - \text{eval-dioph } ks (\text{map (} \lambda x. x \ \text{mod} \ 2) \ x s)) \ \text{div} \ 2) \\
\]

\( x s \) is a solution iff...

- ...it is a solution modulo 2, and...
- ...quotient of \( x s \) and 2 is a solution of another equation with same coefficients, but different right-hand side.
Diophantine Equations

Method by Boudet and Comon

\[
\text{(eval-dioph } \text{ ks } \text{ xs } = \ l) = \\
\text{(eval-dioph } \text{ ks } \text{(map } (\lambda x. \ x \mod 2) \text{ xs}) \mod 2 = \ l \mod 2 \land \\
\text{eval-dioph } \text{ ks } \text{(map } (\lambda x. \ x \div 2) \text{ xs)} = \\
(\ l - \text{ eval-dioph } \text{ ks } \text{(map } (\lambda x. \ x \mod 2) \text{ xs})) \div 2)
\]

\(\text{xs}\) is a solution iff . . .

- . . . it is a solution modulo 2, and . . .
- . . . quotient of \(\text{xs}\) and 2 is a solution of another equation with same coefficients, but different right-hand side.

Reachable right-hand sides are bounded

If \(|m| \leq \max |l| (\sum k \leftarrow \text{ks. } |k|)\) then

\(|(m - \text{eval-dioph } \text{ ks } \text{(map } (\lambda x. \ x \mod 2) \text{ xs})) \div 2| \\
\leq \max |l| (\sum k \leftarrow \text{ks. } |k|).\)
**Formula:** $2x - 3y = 2$
**Diophantine Equations — Example**

**Formula:** \(2x - 3y = 2\)

Some solutions

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[7 \cdot 2 = 14\]
\[4 \cdot 3 = 12\]
**Diophantine Equations — Example**

**Formula:** \(2x - 3y = 2\)

Some solutions

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(22 \cdot 2 = 44\)
\(14 \cdot 3 = 42\)
Diophantine Equations — Example

Formula: \(2x - 3y = 2\)

Some solutions

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

\[10 \cdot 2 = 20\]
\[6 \cdot 3 = 18\]
Diophantine Equations — Example

**Formula:** \(2x - 3y = 2\)

![Diophantine Equations Diagram](image)

**Some solutions**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>102 (\cdot) 3 = 306</td>
</tr>
</tbody>
</table>
eq-dfa :: nat ⇒ int list ⇒ int ⇒ dfa

eq-dfa n ks l ≡
let (is, js) = dioph-dfs n ks l
in (map (λj. make-bdd
  (λxs. if eval-dioph ks xs mod 2 = j mod 2 then the is[int-to-nat-bij
  (j − eval-dioph ks xs) div 2])
  else |js|)
  n [[])
js ⊗
[Leaf |js|],
map (λj. j = 0) js ⊗ [False])
Strengthened statement

\[ \text{If } (l, m) \in (\text{succsr (dioph-succs n ks)})^* \text{ and } \forall bs \in bss. \text{ is-alph n bs then} \]
\[ \text{dfa-accepting (eq-dfa n ks l)} \]
\[ (\text{dfa-steps (eq-dfa n ks l)} \]
\[ (\text{the (fst (dioph-dfs n ks l))[int-to-nat-bij m]} \] \[ bss) = (eval-dioph ks (nats-of-boolss n bss) = m). \]
Corollary \((l = m)\)

If \(\forall bs \in bss. \text{is-\text{alph}}~n~bs\) then

\[
\text{dfa-accepts}~(\text{eq-dfa}~n~ks~l)~bss = (\text{eval-dioph}~ks~(\text{nats-of-boolss}~n~bss) = l).
\]
The Decision Procedure

dfa-of-pf :: nat ⇒ pf ⇒ dfa

dfa-of-pf n (Eq ks l) = eq-dfa n ks l

dfa-of-pf n (Le ks l) = ineq-dfa n ks l

dfa-of-pf n (Neg p) = negate-dfa (dfa-of-pf n p)

dfa-of-pf n (And p q) =
  binop-dfa (∧) (dfa-of-pf n p) (dfa-of-pf n q)

dfa-of-pf n (Or p q) =
  binop-dfa (∨) (dfa-of-pf n p) (dfa-of-pf n q)

dfa-of-pf n (Exist p) =
  rquot (det-nfa (quantify-nfa 0 (nfa-of-dfa (dfa-of-pf (Suc n) p)))))))
  n

dfa-of-pf n (Forall p) = dfa-of-pf n (Neg (Exist (Neg p)))
The Decision Procedure

dfa-of-pf :: nat ⇒ pf ⇒ dfa

dafo-of-pf n (Eq ks l) = eq-dfa n ks l

dafo-of-pf n (Le ks l) = ineq-dfa n ks l

dafo-of-pf n (Neg p) = negate-dfa (dfa-of-pf n p)

dafo-of-pf n (And p q) =
  binop-dfa (∧) (dfa-of-pf n p) (dfa-of-pf n q)

dafo-of-pf n (Or p q) =
  binop-dfa (∨) (dfa-of-pf n p) (dfa-of-pf n q)

dafo-of-pf n (Exist p) =
  rquot (det-nfa (quantify-nfa 0 (nfa-of-dfa (dfa-of-pf (Suc n) p))))
  n

dafo-of-pf n (Forall p) = dfa-of-pf n (Neg (Exist (Neg p)))

Theorem (Correctness)

If ∀ bs∈bss. is-alph n bs then
dafo-accepts (dfa-of-pf n p) bss = eval-pf p (nats-of-boolss n bss).
Outline

1. Introduction
2. Basic Concepts
3. Automata Construction
4. The Decision Procedure
5. Conclusion
Algorithm can compete quite well with standard decision procedure for Presburger arithmetic available in Isabelle.
• Algorithm can compete quite well with standard decision procedure for Presburger arithmetic available in Isabelle.
• Size of DFA for stamp problem (without minimization): 6 states
Performance

- Algorithm can compete quite well with standard decision procedure for Presburger arithmetic available in Isabelle.
- Size of DFA for stamp problem (without minimization): 6 states
- Size of DFAs for subformulae:

<table>
<thead>
<tr>
<th>Forall</th>
<th>Imp</th>
<th>Exist</th>
<th>Exist</th>
<th>Eq [5, 3, −1]</th>
<th>Le [−1] − 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
<td>13</td>
<td>9</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>
Performance

- Algorithm can compete quite well with standard decision procedure for Presburger arithmetic available in Isabelle.
- Size of DFA for stamp problem (without minimization): 6 states
- Size of DFAs for subformulae:

<table>
<thead>
<tr>
<th>Forall</th>
<th>Imp</th>
<th>Exist</th>
<th>Exist</th>
<th>Eq [5, 3, −1] 0</th>
<th>Le [−1] −8</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
<td>13</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

- Use of DFS pays off!
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first
Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first

Other representations for BDDs
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first
- Other representations for BDDs
  - ROBDDs with sharing [Verma, Asian 2000]
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first
- Other representations for BDDs
  - ROBDDs with sharing [Verma, Asian 2000]
  - Requires memory for storing BDDs
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first
- Other representations for BDDs
  - ROBDDs with sharing [Verma, Asian 2000]
  - Requires memory for storing BDDs
  - Algorithms no longer purely functional
    (more challenging to reason about)
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first
- Other representations for BDDs
  - ROBDDs with sharing [Verma, Asian 2000]
  - Requires memory for storing BDDs
  - Algorithms no longer purely functional
    (more challenging to reason about)
- Presburger Arithmetic on reals (using Büchi automata)
Future Work

- Other methods for constructing DFAs for Diophantine equations, e.g. [Wolper and Boigelot, TACAS 2000]
  - can directly deal with variables over integers
  - sign bit / most significant bit comes first

- Other representations for BDDs
  - ROBDDs with sharing [Verma, Asian 2000]
  - Requires memory for storing BDDs
  - Algorithms no longer purely functional
    (more challenging to reason about)

- Presburger Arithmetic on reals (using Büchi automata)

- Extend to WS1S, and apply it to circuit verification problems described in [Basin and Friedrich, FROCOS 2000].