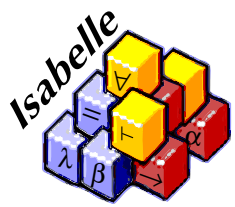
# Formalizing the Logic-Automaton Connection

Stefan Berghofer  Markus Reiter

Institut für Informatik
Technische Universität München




Uppsala, 23.4.2009

# Quiz

**Which of these formulae are true (for natural numbers)?**

$$\forall x{\geq}7.\ \exists y\ z.\ 3 * y + 5 * z = x$$

$$\forall x{\geq}8.\ \exists y\ z.\ 3 * y + 5 * z = x$$

$$\forall x{\geq}8.\ \exists y\ z.\ 4 * y + 5 * z = x$$

# Quiz

**Which of these formulae are true (for natural numbers)?**

$$\forall x{\geq}7.\ \exists y\ z.\ 3 * y + 5 * z = x$$

$$\forall x{\geq}8.\ \exists y\ z.\ 3 * y + 5 * z = x$$

$$\forall x{\geq}8.\ \exists y\ z.\ 4 * y + 5 * z = x$$

# Quiz

**Which of these formulae are true (for natural numbers)?**

$$\forall x{\geq}7.\ \exists y\ z.\ 3 * y + 5 * z = x$$

$$\forall x{\geq}8.\ \exists y\ z.\ 3 * y + 5 * z = x$$

$$\forall x{\geq}8.\ \exists y\ z.\ 4 * y + 5 * z = x$$

**Stamp problem**

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

# Outline

# Outline

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.
  - Presburger arithmetic (this talk)

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.
  - Presburger arithmetic (this talk)
  - Weak Second-order theory of One Successor (WS1S)

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.
  - Presburger arithmetic (this talk)
  - Weak Second-order theory of One Successor (WS1S)
- Approaches for implementing decision procedures for PA

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.
  - Presburger arithmetic (this talk)
  - Weak Second-order theory of One Successor (WS1S)
- Approaches for implementing decision procedures for PA

  Algebraic e.g. Cooper's algorithm

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.
  - Presburger arithmetic (this talk)
  - Weak Second-order theory of One Successor (WS1S)
- Approaches for implementing decision procedures for PA

  Algebraic e.g. Cooper's algorithm

  Semantic e.g. using automata on bitstrings

  (also works for WS1S, see MONA)

# Introduction

- Higher-order logic (HOL) is undecidable in general, but...
- ...many decidable logics can be embedded into HOL, e.g.
  - Presburger arithmetic (this talk)
  - Weak Second-order theory of One Successor (WS1S)
- Approaches for implementing decision procedures for PA

  Algebraic e.g. Cooper's algorithm

  Semantic e.g. using automata on bitstrings

  (also works for WS1S, see MONA)
- How to write a decision procedure in a theorem prover based on HOL (in this talk: Isabelle)?

# Integrating Decision Procedures into a Theorem Prover

Oracle-based Use an external tool such as MONA, and simply trust the answer of the tool.

# Integrating Decision Procedures into 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.

# Integrating Decision Procedures into 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.

**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.

# Integrating Decision Procedures into 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.

**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.

**Reflection** Write and verify the decision procedure as a recursive function in HOL itself (this talk).

# Related Work

[Boutin, TACS 1997] Decision procedure for abelian rings in Coq (reflection)

# Related Work

[Boutin, TACS 1997] Decision procedure for abelian rings in Coq (reflection)

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

# Related Work

[Boutin, TACS 1997] Decision procedure for abelian rings in Coq (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)

# Related Work

[Boutin, TACS 1997] Decision procedure for abelian rings in Coq (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)

# Related Work

[Boutin, TACS 1997] Decision procedure for abelian rings in Coq (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)

[Basin and Friedrich, FroCoS 2000] Combining WS1S and HOL (oracle)

# 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]

# 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!

# Contributions

- Library for automata on bitstrings

# Contributions

- Library for automata on bitstrings
  - Product automaton

# Contributions

- Library for automata on bitstrings
  - Product automaton
  - Projection

# Contributions

- Library for automata on bitstrings
  - Product automaton
  - Projection
  - Determinization

# Contributions

- Library for automata on bitstrings
  - Product automaton
  - Projection
  - Determinization
- First formalization of an automata-based decision procedure for Presburger arithmetic in a theorem prover

# Contributions

- Library for automata on bitstrings
  - Product automaton
  - Projection
  - Determinization
- First formalization of an automata-based decision procedure for Presburger arithmetic in a theorem prover
- Can easily be changed to cover WS1S

# Contributions

- Library for automata on bitstrings
  - Product automaton
  - Projection
  - Determinization
- First formalization of an automata-based decision procedure for Presburger arithmetic in a theorem prover
- Can easily be changed to cover WS1S
- Efficiently executable thanks to Isabelle's ML code generator

# Outline

# Presburger Arithmetic

## Syntax (using de Bruijn indices)

**datatype** $pf = Eq\ (int\ list)\ int \mid Le\ (int\ list)\ int \mid And\ pf\ pf$
$\mid Or\ pf\ pf \mid Imp\ pf\ pf \mid Forall\ pf \mid Exist\ pf \mid Neg\ pf$

# Presburger Arithmetic

## Syntax (using de Bruijn indices)

**datatype** $pf = Eq\ (int\ list)\ int \mid Le\ (int\ list)\ int \mid And\ pf\ pf$
$\mid Or\ pf\ pf \mid Imp\ pf\ pf \mid Forall\ pf \mid Exist\ pf \mid Neg\ pf$

## Example: Stamp problem

$\forall x{\geq}8.\ \exists y\ z.\ 3 * y + 5 * z = x$

# Presburger Arithmetic

## Syntax (using de Bruijn indices)

**datatype** $pf = Eq\ (int\ list)\ int \mid Le\ (int\ list)\ int \mid And\ pf\ pf \mid Or\ pf\ pf \mid Imp\ pf\ pf \mid Forall\ pf \mid Exist\ pf \mid Neg\ pf$

## Example: Stamp problem

$\forall x{\geq}8.\ \exists y\ z.\ 3 * y + 5 * z = x$

## Encoding

$Forall\ (Imp\ (Le\ [-1]\ -8)\ (Exist\ (Exist\ (Eq\ [5, 3, -1]\ 0))))$

# Evaluation

## Diophantine (In)Equations

$$\textit{eval-dioph} :: \textit{int list} \Rightarrow \textit{nat list} \Rightarrow \textit{int}$$
$$\textit{eval-dioph}\ (k \cdot ks)\ (x \cdot xs) = k * \textit{int}\ x + \textit{eval-dioph}\ ks\ xs$$
$$\textit{eval-dioph}\ []\ xs = 0$$
$$\textit{eval-dioph}\ ks\ [] = 0$$

# Evaluation

## Diophantine (In)Equations

$$\textit{eval-dioph} :: \textit{int list} \Rightarrow \textit{nat list} \Rightarrow \textit{int}$$
$$\textit{eval-dioph}\ (k \cdot ks)\ (x \cdot xs) = k * \textit{int}\ x + \textit{eval-dioph}\ ks\ xs$$
$$\textit{eval-dioph}\ []\ xs = 0$$
$$\textit{eval-dioph}\ ks\ [] = 0$$

## Formulae

$$\textit{eval-pf} :: \textit{pf} \Rightarrow \textit{nat list} \Rightarrow \textit{bool}$$
$$\textit{eval-pf}\ (\textit{Eq}\ ks\ l)\ xs = (\textit{eval-dioph}\ ks\ xs = l)$$
$$\textit{eval-pf}\ (\textit{Le}\ ks\ l)\ xs = (\textit{eval-dioph}\ ks\ xs \leq l)$$
$$\textit{eval-pf}\ (\textit{Neg}\ p)\ xs = (\neg\ \textit{eval-pf}\ p\ xs)$$
$$\textit{eval-pf}\ (\textit{And}\ p\ q)\ xs = (\textit{eval-pf}\ p\ xs \wedge \textit{eval-pf}\ q\ xs)$$
$$\textit{eval-pf}\ (\textit{Or}\ p\ q)\ xs = (\textit{eval-pf}\ p\ xs \vee \textit{eval-pf}\ q\ xs)$$
$$\textit{eval-pf}\ (\textit{Forall}\ p)\ xs = (\forall x.\ \textit{eval-pf}\ p\ (x \cdot xs))$$
$$\textit{eval-pf}\ (\textit{Exist}\ p)\ xs = (\exists x.\ \textit{eval-pf}\ p\ (x \cdot xs))$$

# Abstract Automata Framework

**Purpose:** Factor out common properties of DFAs and NFAs

# Abstract Automata Framework

**Purpose:** Factor out common properties of DFAs and NFAs

## Ingredients

| States | Input symbols | Transition function |
|---|---|---|
| $\sigma$ | $\alpha$ | $tr :: \sigma \Rightarrow \alpha \Rightarrow \sigma$ |

# Abstract Automata Framework

**Purpose:** Factor out common properties of DFAs and NFAs

## Ingredients

| States | Input symbols | Transition function |
|---|---|---|
| $\sigma$ | $\alpha$ | $tr :: \sigma \Rightarrow \alpha \Rightarrow \sigma$ |

## Extending Transition Functions to Words

$steps :: (\sigma \Rightarrow \alpha \Rightarrow \sigma) \Rightarrow \sigma \Rightarrow \alpha\ list \Rightarrow \sigma$

$steps\ tr\ q\ [] = q$

$steps\ tr\ q\ (a \cdot as) = steps\ tr\ (tr\ q\ a)\ as$

# Abstract Automata Framework

**Purpose:** Factor out common properties of DFAs and NFAs

## Ingredients

| States | Input symbols | Transition function |
|---|---|---|
| $\sigma$ | $\alpha$ | $tr :: \sigma \Rightarrow \alpha \Rightarrow \sigma$ |

## Extending Transition Functions to Words

$steps :: (\sigma \Rightarrow \alpha \Rightarrow \sigma) \Rightarrow \sigma \Rightarrow \alpha\ list \Rightarrow \sigma$

$steps\ tr\ q\ [] = q$

$steps\ tr\ q\ (a \cdot as) = steps\ tr\ (tr\ q\ a)\ as$

## Accepted Words

$accepts :: (\sigma \Rightarrow \alpha \Rightarrow \sigma) \Rightarrow (\sigma \Rightarrow bool) \Rightarrow \sigma \Rightarrow \alpha\ list \Rightarrow bool$

$accepts\ tr\ P\ s\ as \equiv P\ (steps\ tr\ s\ as)$

# 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 $n$:

$$\begin{matrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{matrix} \left[\left[\begin{matrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{matrix}\right]\right]$$

# 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 $n$:

$$\begin{matrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{matrix} \left[\left[\begin{matrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{matrix}\right]\right]$$

- $i$-th row: value of $i$-th variable (natural number)

# 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 $n$:

$$\begin{matrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{matrix} \left[\left[\begin{matrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{matrix}\right]\right]$$

- $i$-th row: value of $i$-th variable (natural number)
- $j$-th column: $j$-th bit of variables

# 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 $n$:

$$\begin{matrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{matrix} \left[\left[\begin{matrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{matrix}\right]\right]$$

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

# 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 $n$:

$$\begin{matrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{matrix} \left[\left[\begin{matrix} b_{0,0} \\ \vdots \\ b_{i,0} \\ \vdots \\ b_{n-1,0} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,j} \\ \vdots \\ b_{i,j} \\ \vdots \\ b_{n-1,j} \end{matrix}\right] \cdots \left[\begin{matrix} b_{0,m-1} \\ \vdots \\ b_{i,m-1} \\ \vdots \\ b_{n-1,m-1} \end{matrix}\right]\right]$$

- $i$-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]

# Values of Variables

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

$$
\begin{aligned}
&\textit{nats-of-boolss} :: \textit{nat} \Rightarrow \textit{bool list list} \Rightarrow \textit{nat list} \\
&\textit{nats-of-boolss}\ n\ [] = \textit{replicate}\ n\ 0 \\
&\textit{nats-of-boolss}\ n\ (bs \cdot bss) = \\
&\quad \textit{map}\ (\lambda(b, x).\ \textit{nat-of-bool}\ b + 2 * x) \\
&\quad (\textit{zip}\ bs\ (\textit{nats-of-boolss}\ n\ bss))
\end{aligned}
$$

# Values of Variables

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

$$
\begin{aligned}
&\mathit{nats\text{-}of\text{-}boolss} :: \mathit{nat} \Rightarrow \mathit{bool\ list\ list} \Rightarrow \mathit{nat\ list} \\
&\mathit{nats\text{-}of\text{-}boolss}\ n\ [] = \mathit{replicate}\ n\ 0 \\
&\mathit{nats\text{-}of\text{-}boolss}\ n\ (bs \cdot bss) = \\
&\quad \mathit{map}\ (\lambda(b, x).\ \mathit{nat\text{-}of\text{-}bool}\ b + 2 * x) \\
&\quad (\mathit{zip}\ bs\ (\mathit{nats\text{-}of\text{-}boolss}\ n\ bss))
\end{aligned}
$$

## Single variable (encoded as row vector)

$$
\begin{aligned}
&\mathit{nat\text{-}of\text{-}bools} :: \mathit{bool\ list} \Rightarrow \mathit{nat} \\
&\mathit{nat\text{-}of\text{-}bools}\ [] = 0 \\
&\mathit{nat\text{-}of\text{-}bools}\ (b \cdot bs) = \mathit{nat\text{-}of\text{-}bool}\ b + 2 * \mathit{nat\text{-}of\text{-}bools}\ bs
\end{aligned}
$$

# Inserting Variables (for Modelling Quantifiers)

## Inserting a row vector into a list of columns

$$insertll :: nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list\ list \Rightarrow \alpha\ list\ list$$
$$insertll\ i\ []\ []\ =\ []$$
$$insertll\ i\ (a \cdot as)\ (bs \cdot bss)\ =\ insertl\ i\ a\ bs \cdot insertll\ i\ as\ bss$$

# Inserting Variables (for Modelling Quantifiers)

## Inserting a row vector into a list of columns

$$insertll :: nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list\ list \Rightarrow \alpha\ list\ list$$
$$insertll\ i\ []\ []\ =\ []$$
$$insertll\ i\ (a \cdot as)\ (bs \cdot bss)\ =\ insertl\ i\ a\ bs \cdot insertll\ i\ as\ bss$$

## Inserting an element into a column vector

$$insertl :: nat \Rightarrow \alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$$
$$insertl\ i\ a\ []\ =\ [a]$$
$$insertl\ 0\ a\ (b \cdot bs)\ =\ a \cdot b \cdot bs$$
$$insertl\ (Suc\ i)\ a\ (b \cdot bs)\ =\ b \cdot insertl\ i\ a\ bs$$

# Inserting Variables (for Modelling Quantifiers)

## Inserting a row vector into a list of columns

$$insertll :: nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list\ list \Rightarrow \alpha\ list\ list$$
$$insertll\ i\ []\ []\ =\ []$$
$$insertll\ i\ (a \cdot as)\ (bs \cdot bss)\ =\ insertl\ i\ a\ bs \cdot insertll\ i\ as\ bss$$

## Inserting an element into a column vector

$$insertl :: nat \Rightarrow \alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$$
$$insertl\ i\ a\ []\ =\ [a]$$
$$insertl\ 0\ a\ (b \cdot bs)\ =\ a \cdot b \cdot bs$$
$$insertl\ (Suc\ i)\ a\ (b \cdot bs)\ =\ b \cdot insertl\ i\ a\ bs$$

## Theorem

*If* $\forall\, bs \in bss.\ is\text{-}alph\ n\ bs$ *and* $|bs| = |bss|$ *and* $i \leq n$ *then*
$nats\text{-}of\text{-}boolss\ (Suc\ n)\ (insertll\ i\ bs\ bss) =$
$insertl\ i\ (nat\text{-}of\text{-}bools\ bs)\ (nats\text{-}of\text{-}boolss\ n\ bss)$.

# BDDs

- **Transition function:** maps state and bitstring to new state(s)

# BDDs

- **Transition function:** maps state and bitstring to new state(s)
- Naive encoding of transition function as a list: lookup operation exponential in the number of variables

# BDDs

- **Transition function:** maps state and bitstring to new state(s)
- Naive encoding of transition function as a list: lookup operation exponential in the number of variables
- Better solution: use BDDs [Klarlund, CSL 1997]

# BDDs

- **Transition function:** maps state and bitstring to new state(s)
- Naive encoding of transition function as a list: lookup operation exponential in the number of variables
- Better solution: use BDDs [Klarlund, CSL 1997]

**Purely functional BDDs (no sharing!)**

**datatype** $\alpha$ *bdd* $=$ *Leaf* $\alpha$ | *Branch* ($\alpha$ *bdd*) ($\alpha$ *bdd*)

# BDDs

- **Transition function:** maps state and bitstring to new state(s)
- Naive encoding of transition function as a list: lookup operation exponential in the number of variables
- Better solution: use BDDs [Klarlund, CSL 1997]

## Purely functional BDDs (no sharing!)

**datatype** $\alpha$ *bdd* $=$ *Leaf* $\alpha$ | *Branch* ($\alpha$ *bdd*) ($\alpha$ *bdd*)

## Lookup

*bdd-lookup* :: $\alpha$ *bdd* $\Rightarrow$ *bool list* $\Rightarrow$ $\alpha$
*bdd-lookup* (*Leaf x*) *bs* $=$ *x*
*bdd-lookup* (*Branch l r*) ($b \cdot$ *bs*) $=$
  *bdd-lookup* (*if b then r else l*) *bs*

# BDDs

- **Transition function:** maps state and bitstring to new state(s)
- Naive encoding of transition function as a list: lookup operation exponential in the number of variables
- Better solution: use BDDs [Klarlund, CSL 1997]

## Purely functional BDDs (no sharing!)

**datatype** $\alpha$ *bdd* $=$ *Leaf* $\alpha$ | *Branch* ($\alpha$ *bdd*) ($\alpha$ *bdd*)

## Lookup

*bdd-lookup* :: $\alpha$ *bdd* $\Rightarrow$ *bool list* $\Rightarrow$ $\alpha$
*bdd-lookup* (*Leaf x*) *bs* $=$ *x*
*bdd-lookup* (*Branch l r*) ($b \cdot bs$) $=$
  *bdd-lookup* (*if b then r else l*) *bs*

**Note:** Only returns meaningful results if height of *bdd* is less or equal to length of *bs* (denoted by *bddh* $|bs|$ *bdd*)

# More Functions on BDDs

- *bdd-map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd*

# More Functions on BDDs

- *bdd-map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd*
- *bdd-all* :: $(\alpha \Rightarrow$ *bool*$) \Rightarrow \alpha$ *bdd* $\Rightarrow$ *bool*

# More Functions on BDDs

- *bdd-map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd*
- *bdd-all* :: $(\alpha \Rightarrow$ *bool*$) \Rightarrow \alpha$ *bdd* $\Rightarrow$ *bool*
- *bdd-binop* :: $(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd* $\Rightarrow \gamma$ *bdd*

# More Functions on BDDs

- *bdd-map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd*
- *bdd-all* :: $(\alpha \Rightarrow$ *bool*$) \Rightarrow \alpha$ *bdd* $\Rightarrow$ *bool*
- *bdd-binop* :: $(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd* $\Rightarrow \gamma$ *bdd*
- *add-leaves* :: $\alpha$ *bdd* $\Rightarrow \alpha$ *list* $\Rightarrow \alpha$ *list*

# More Functions on BDDs

- *bdd-map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd*
- *bdd-all* :: $(\alpha \Rightarrow$ *bool*$) \Rightarrow \alpha$ *bdd* $\Rightarrow$ *bool*
- *bdd-binop* :: $(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha$ *bdd* $\Rightarrow \beta$ *bdd* $\Rightarrow \gamma$ *bdd*
- *add-leaves* :: $\alpha$ *bdd* $\Rightarrow \alpha$ *list* $\Rightarrow \alpha$ *list*

**Theorem (Correctness of *bdd-binop*)**

*If bddh* $|bs|$ *l and bddh* $|bs|$ *r then*
*bdd-lookup* (*bdd-binop f l r*) *bs* =
*f* (*bdd-lookup l bs*) (*bdd-lookup r bs*).

# Deterministic Automata

**Represented by type**

$$dfa = \underbrace{nat\ bdd\ list}_{\text{transition table}} \times \underbrace{bool\ list}_{\text{accepting states}}$$

# Deterministic Automata

**Represented by type**

$$dfa = \underbrace{nat\ bdd\ list}_{\text{transition table}} \times \underbrace{bool\ list}_{\text{accepting states}}$$

**Note:** start state = 0

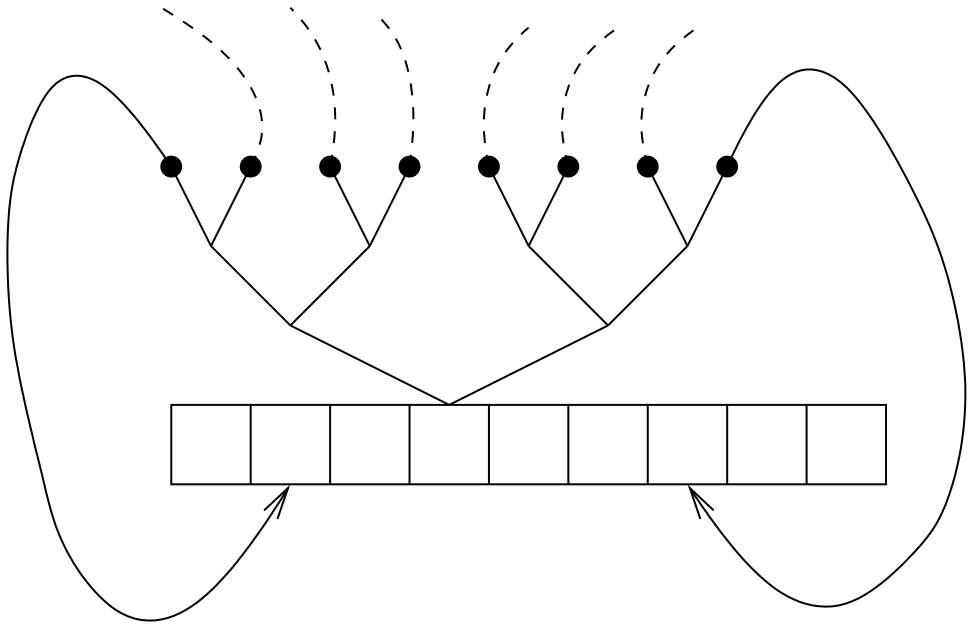
# Deterministic Automata

## Represented by type

$$dfa = \underbrace{nat\ bdd\ list}_{\text{transition table}} \times \underbrace{bool\ list}_{\text{accepting states}}$$

**Note:** start state = 0

## Transition table



# Deterministic Automata — Continued

## Well-formedness

$$wf\text{-}dfa :: dfa \Rightarrow nat \Rightarrow bool$$
$$wf\text{-}dfa\ A\ n \equiv$$
$$(\forall\, bdd{\in}fst\ A.\ bddh\ n\ bdd) \wedge$$
$$(\forall\, bdd{\in}fst\ A.\ bdd\text{-}all\ (dfa\text{-}is\text{-}node\ A)\ bdd) \wedge$$
$$|snd\ A| = |fst\ A| \wedge 0 < |fst\ A|$$

# Deterministic Automata — Continued

## Well-formedness

$$wf\text{-}dfa :: dfa \Rightarrow nat \Rightarrow bool$$
$$wf\text{-}dfa\ A\ n \equiv$$
$$(\forall\, bdd{\in}fst\ A.\ bddh\ n\ bdd) \wedge$$
$$(\forall\, bdd{\in}fst\ A.\ bdd\text{-}all\ (dfa\text{-}is\text{-}node\ A)\ bdd) \wedge$$
$$|snd\ A| = |fst\ A| \wedge 0 < |fst\ A|$$

## Transition function

$$dfa\text{-}trans :: dfa \Rightarrow nat \Rightarrow bool\ list \Rightarrow nat$$
$$dfa\text{-}trans\ A\ q\ bs \equiv bdd\text{-}lookup\ (fst\ A)[q]\ bs$$

# Deterministic Automata — Continued

## Well-formedness

$$wf\text{-}dfa :: dfa \Rightarrow nat \Rightarrow bool$$
$$wf\text{-}dfa\ A\ n \equiv$$
$$(\forall\, bdd{\in}fst\ A.\ bddh\ n\ bdd) \wedge$$
$$(\forall\, bdd{\in}fst\ A.\ bdd\text{-}all\ (dfa\text{-}is\text{-}node\ A)\ bdd) \wedge$$
$$|snd\ A| = |fst\ A| \wedge 0 < |fst\ A|$$

## Transition function

$$dfa\text{-}trans :: dfa \Rightarrow nat \Rightarrow bool\ list \Rightarrow nat$$
$$dfa\text{-}trans\ A\ q\ bs \equiv bdd\text{-}lookup\ (fst\ A)[q]\ bs$$

## Acceptance condition

$$dfa\text{-}accepting :: dfa \Rightarrow nat \Rightarrow bool$$
$$dfa\text{-}accepting\ A\ q \equiv (snd\ A)[q]$$

**Represented by type**

$$nfa = \underbrace{bool\ list\ bdd\ list}_{\text{transition table}} \times \underbrace{bool\ list}_{\text{accepting states}}$$

**Represented by type**

$$nfa = \underbrace{bool\ list\ bdd\ list}_{\text{transition table}} \times \underbrace{bool\ list}_{\text{accepting states}}$$

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

# Nondeterministic Automata

**Represented by type**

$$nfa = \underbrace{bool\ list\ bdd\ list}_{\text{transition table}} \times \underbrace{bool\ list}_{\text{accepting states}}$$

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

## Acceptance condition

$nfa\text{-}accepting' :: bool\ list \Rightarrow bool\ list \Rightarrow bool$
$nfa\text{-}accepting'\ []\ bs = False$
$nfa\text{-}accepting'\ (a \cdot as)\ [] = False$
$nfa\text{-}accepting'\ (a \cdot as)\ (b \cdot bs) = (a \wedge b \vee nfa\text{-}accepting'\ as\ bs)$

$nfa\text{-}accepting :: nfa \Rightarrow bool\ list \Rightarrow bool$
$nfa\text{-}accepting\ A \equiv nfa\text{-}accepting'\ (snd\ A)$

# Nondeterministic Automata — Continued

## Transition table for specific state

$$
\begin{aligned}
&\mathit{subsetbdd} :: \mathit{bool\ list\ bdd\ list} \\
&\qquad \Rightarrow \mathit{bool\ list} \Rightarrow \mathit{bool\ list\ bdd} \Rightarrow \mathit{bool\ list\ bdd} \\
&\mathit{subsetbdd}\ []\ []\ \mathit{bdd} = \mathit{bdd} \\
&\mathit{subsetbdd}\ (\mathit{bdd}' \cdot \mathit{bdds})\ (b \cdot \mathit{bs})\ \mathit{bdd} = \\
&\quad (\mathit{if}\ b\ \mathit{then}\ \mathit{subsetbdd}\ \mathit{bdds}\ \mathit{bs}\ (\mathit{bdd\text{-}binop}\ \mathit{bv\text{-}or}\ \mathit{bdd}\ \mathit{bdd}') \\
&\qquad \mathit{else}\ \mathit{subsetbdd}\ \mathit{bdds}\ \mathit{bs}\ \mathit{bdd}) \\
\\
&\mathit{nfa\text{-}emptybdd} :: \mathit{nat} \Rightarrow \mathit{bool\ list\ bdd} \\
&\mathit{nfa\text{-}emptybdd}\ n \equiv \mathit{Leaf}\ (\mathit{replicate}\ n\ \mathit{False})
\end{aligned}
$$

# Nondeterministic Automata — Continued

## Transition table for specific state

$$
\begin{aligned}
&\mathit{subsetbdd} :: \mathit{bool\ list\ bdd\ list} \\
&\qquad \Rightarrow \mathit{bool\ list} \Rightarrow \mathit{bool\ list\ bdd} \Rightarrow \mathit{bool\ list\ bdd} \\
&\mathit{subsetbdd}\ []\ []\ \mathit{bdd} = \mathit{bdd} \\
&\mathit{subsetbdd}\ (\mathit{bdd}' \cdot \mathit{bdds})\ (b \cdot \mathit{bs})\ \mathit{bdd} = \\
&\quad (\mathit{if}\ b\ \mathit{then}\ \mathit{subsetbdd}\ \mathit{bdds}\ \mathit{bs}\ (\mathit{bdd\text{-}binop}\ \mathit{bv\text{-}or}\ \mathit{bdd}\ \mathit{bdd}') \\
&\qquad \mathit{else}\ \mathit{subsetbdd}\ \mathit{bdds}\ \mathit{bs}\ \mathit{bdd}) \\
\\
&\mathit{nfa\text{-}emptybdd} :: \mathit{nat} \Rightarrow \mathit{bool\ list\ bdd} \\
&\mathit{nfa\text{-}emptybdd}\ n \equiv \mathit{Leaf}\ (\mathit{replicate}\ n\ \mathit{False})
\end{aligned}
$$

## Transition function

$$
\begin{aligned}
&\mathit{nfa\text{-}trans} :: \mathit{nfa} \Rightarrow \mathit{bool\ list} \Rightarrow \mathit{bool\ list} \Rightarrow \mathit{bool\ list} \\
&\mathit{nfa\text{-}trans}\ A\ \mathit{qs}\ \mathit{bs} \equiv \\
&\quad \mathit{bdd\text{-}lookup}\ (\mathit{subsetbdd}\ (\mathit{fst}\ A)\ \mathit{qs}\ (\mathit{nfa\text{-}emptybdd}\ |\mathit{qs}|))\ \mathit{bs}
\end{aligned}
$$

# Depth First Search

## Naive implementation

- Product automaton: $m \cdot n$ states

# Depth First Search

## Naive implementation

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

# Depth First Search

**Naive implementation**

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

**Better:** only generate reachable states

# Depth First Search

**Naive implementation**

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

**Better:** only generate reachable states

**Generic DFS function [Nishihara and Minamide, AFP 2004]**

*dfs* :: $\beta \Rightarrow \alpha$ *list* $\Rightarrow \beta$ ($\alpha$: node, $\beta$: node store)

*dfs* $S$ [] $= S$

*dfs* $S$ $(x \cdot xs) =$

(*if memb* $x$ $S$ *then dfs* $S$ $xs$ *else dfs* (*ins* $x$ $S$) (*succs* $x$ @ $xs$))

# Depth First Search

**Naive implementation**

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

**Better:** only generate reachable states

## Generic DFS function [Nishihara and Minamide, AFP 2004]

$dfs :: \beta \Rightarrow \alpha\ list \Rightarrow \beta$ ($\alpha$: node, $\beta$: node store)

$dfs\ S\ [] = S$

$dfs\ S\ (x \cdot xs) =$

$(if\ memb\ x\ S\ then\ dfs\ S\ xs\ else\ dfs\ (ins\ x\ S)\ (succs\ x\ @\ xs))$

## Parameters for DFS

$succs :: \alpha \Rightarrow \alpha\ list$

$empt :: \beta$

$is\text{-}node :: \alpha \Rightarrow bool$

$ins :: \alpha \Rightarrow \beta \Rightarrow \beta$

$memb :: \alpha \Rightarrow \beta \Rightarrow bool$

$invariant :: \beta \Rightarrow bool$

# Outline

# Automata Construction

- Complement: for negation

# Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\vee$, $\wedge$, and $\longrightarrow$

# Automata Construction

- Complement: for negation
- Product automaton: for binary operators, i.e. $\vee$, $\wedge$, and $\longrightarrow$
- Projection: for existential quantifiers

# Automata Construction

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

# Complement

$$\mathit{negate\text{-}dfa} :: \mathit{dfa} \Rightarrow \mathit{dfa}$$
$$\mathit{negate\text{-}dfa} \equiv \lambda(t,\ a).\ (t,\ \mathit{map}\ \mathit{Not}\ a)$$

# Complement

$$\textit{negate-dfa} :: \textit{dfa} \Rightarrow \textit{dfa}$$
$$\textit{negate-dfa} \equiv \lambda(t,\ a).\ (t,\ \textit{map Not}\ a)$$

**Theorem (Correctness)**

$$\textit{If wf-dfa } A\ n \textit{ and } \forall\, bs{\in}bss.\ \textit{is-alph } n\ bs \textit{ then}$$
$$\textit{dfa-accepts } (\textit{negate-dfa } A)\ bss = (\neg\ \textit{dfa-accepts } A\ bss).$$

# Complement

$$\textit{negate-dfa} :: \textit{dfa} \Rightarrow \textit{dfa}$$
$$\textit{negate-dfa} \equiv \lambda(t,\ a).\ (t,\ \textit{map Not}\ a)$$

**Theorem (Correctness)**

$$\textit{If wf-dfa } A\ n \textit{ and } \forall\, bs{\in}bss.\ \textit{is-alph } n\ bs \textit{ then}$$
$$\textit{dfa-accepts } (\textit{negate-dfa } A)\ bss = (\neg\ \textit{dfa-accepts } A\ bss).$$

**Theorem (Well-formedness)**

$$\textit{wf-dfa } (\textit{negate-dfa } A)\ n = \textit{wf-dfa } A\ n$$

# Product Automaton

- Given two automata $A$ and $B$ with states in *nat*, product automaton has states in $nat \times nat$

# Product Automaton

- Given two automata $A$ and $B$ with states in *nat*, product automaton has states in $nat \times nat$
- Type of automata must be closed under product construction

# Product Automaton

- Given two automata $A$ and $B$ with states in *nat*, product automaton has states in $nat \times nat$
- Type of automata must be closed under product construction
- Need mappings $nat \times nat \Rightarrow nat$ and $nat \Rightarrow nat \times nat$

# Product Automaton

- Given two automata $A$ and $B$ with states in $nat$, product automaton has states in $nat \times nat$
- Type of automata must be closed under product construction
- Need mappings $nat \times nat \Rightarrow nat$ and $nat \Rightarrow nat \times nat$

## Instantiation of DFS

node: $nat \times nat$

node store: $\underbrace{nat\ option\ list\ list}_{\text{mapping } nat \times nat \Rightarrow nat} \times \underbrace{(nat \times nat)\ list}_{\text{mapping } nat \Rightarrow nat \times nat}$

$$prod\text{-}dfs :: dfa \Rightarrow dfa \Rightarrow nat \times nat \Rightarrow nat\ option\ list\ list \times (nat \times nat)\ list$$

$$prod\text{-}dfs\ A\ B\ x \equiv gen\text{-}dfs\ (prod\text{-}succs\ A\ B)\ prod\text{-}ins\ prod\text{-}memb\ (prod\text{-}empt\ A\ B)\ [x]$$

# Product Automaton — Continued

## Building the automaton

$$
\begin{aligned}
&\mathit{binop\text{-}dfa} :: (\mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}) \Rightarrow \mathit{dfa} \Rightarrow \mathit{dfa} \Rightarrow \mathit{dfa} \\
&\mathit{binop\text{-}dfa}\ f\ A\ B \equiv \\
&\mathit{let}\ (\mathit{tab}, \mathit{ps}) = \mathit{prod\text{-}dfs}\ A\ B\ (0, 0) \\
&\mathit{in}\ (\mathit{map}\ (\lambda(i, j). \\
&\qquad \mathit{bdd\text{-}binop}\ (\lambda k\ l.\ \mathit{the}\ \mathit{tab}[k][l])\ (\mathit{fst}\ A)[i]\ (\mathit{fst}\ B)[j]) \\
&\quad \mathit{ps}, \\
&\quad \mathit{map}\ (\lambda(i, j).\ f\ (\mathit{snd}\ A)[i]\ (\mathit{snd}\ B)[j])\ \mathit{ps})
\end{aligned}
$$

# Product Automaton — Continued

## Building the automaton

$$
\begin{aligned}
&\mathit{binop\text{-}dfa} :: (\mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}) \Rightarrow \mathit{dfa} \Rightarrow \mathit{dfa} \Rightarrow \mathit{dfa} \\
&\mathit{binop\text{-}dfa}\ f\ A\ B \equiv \\
&\mathit{let}\ (\mathit{tab}, \mathit{ps}) = \mathit{prod\text{-}dfs}\ A\ B\ (0, 0) \\
&\mathit{in}\ (\mathit{map}\ (\lambda(i, j). \\
&\qquad \mathit{bdd\text{-}binop}\ (\lambda k\ l.\ \mathit{the}\ \mathit{tab}[k][l])\ (\mathit{fst}\ A)[i]\ (\mathit{fst}\ B)[j]) \\
&\quad \mathit{ps}, \\
&\quad \mathit{map}\ (\lambda(i, j).\ f\ (\mathit{snd}\ A)[i]\ (\mathit{snd}\ B)[j])\ \mathit{ps})
\end{aligned}
$$

## Theorem (Correctness)

$$
\begin{aligned}
&\mathit{If}\ \mathit{wf\text{-}dfa}\ A\ n\ \mathit{and}\ \mathit{wf\text{-}dfa}\ B\ n\ \mathit{and}\ \forall\, \mathit{bs} \in \mathit{bss}.\ \mathit{is\text{-}alph}\ n\ \mathit{bs}\ \mathit{then} \\
&\mathit{dfa\text{-}accepts}\ (\mathit{binop\text{-}dfa}\ f\ A\ B)\ \mathit{bss} = \\
&f\ (\mathit{dfa\text{-}accepts}\ A\ \mathit{bss})\ (\mathit{dfa\text{-}accepts}\ B\ \mathit{bss}).
\end{aligned}
$$

# Quantifiers

## Existential quantifiers

- Convert DFA to NFA (trivial)

# Quantifiers

**Existential quantifiers**

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

# Quantifiers

**Existential quantifiers**

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

# Quantifiers

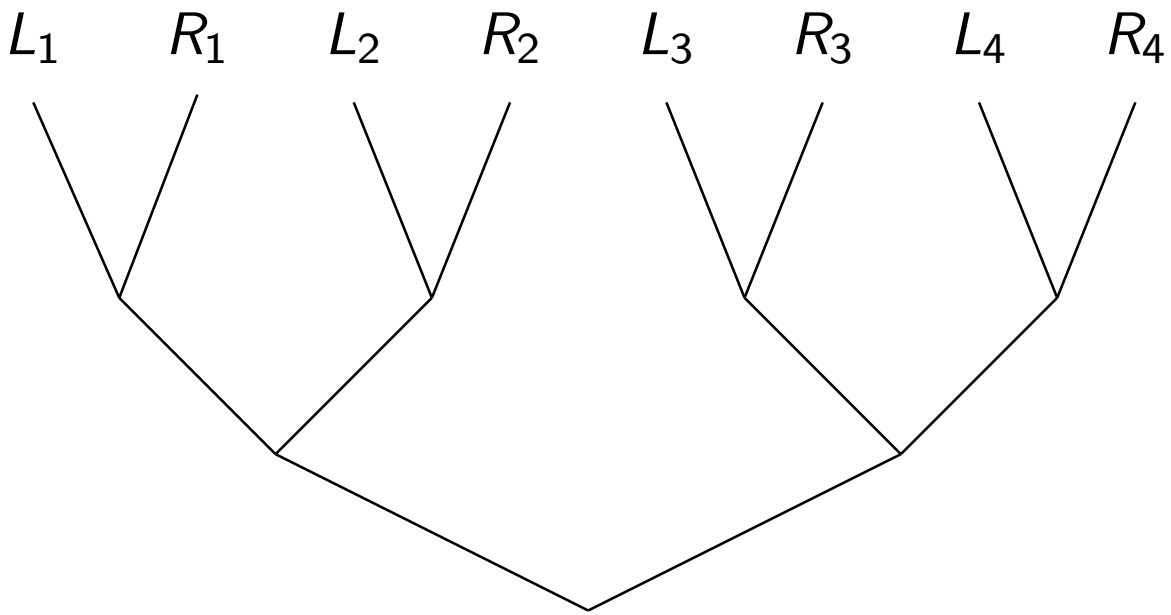
## Existential quantifiers

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

## Universal quantifiers

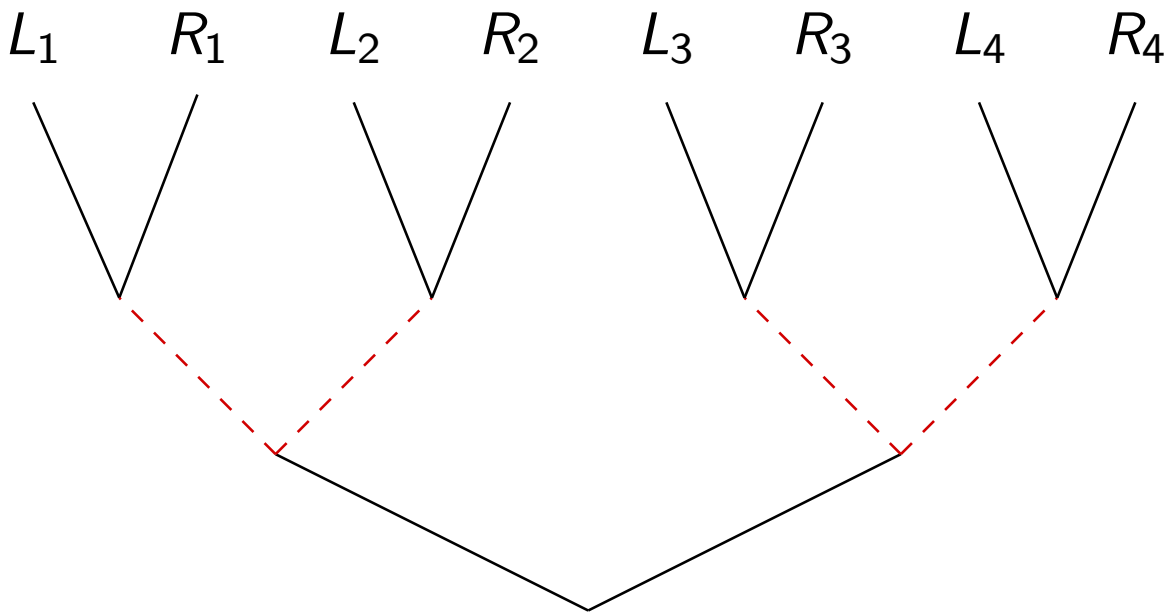
**Note:** $(\forall x.\ P\ x) = (\neg\ (\exists x.\ \neg\ P\ x))$

# Projection



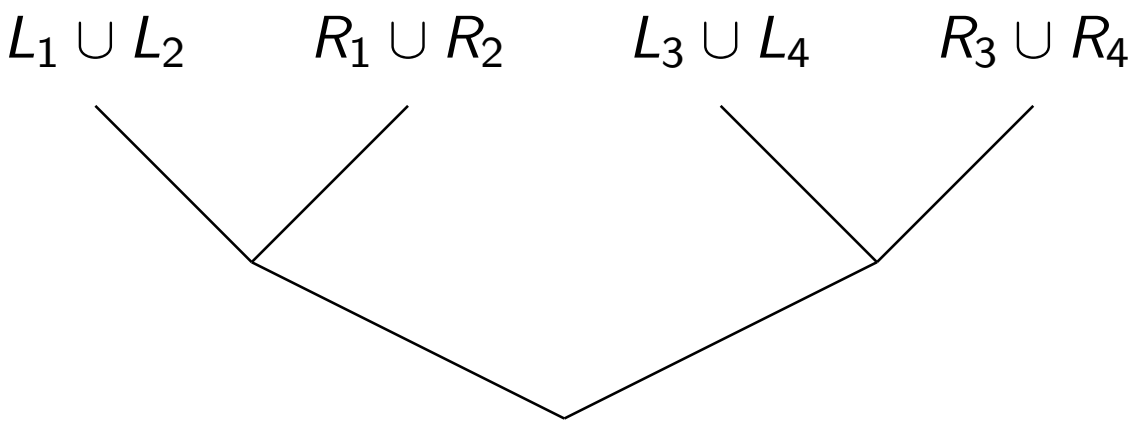

*quantify-bdd* :: *nat* ⇒ *bool list bdd* ⇒ *bool list bdd*
*quantify-bdd i* (*Leaf q*) = *Leaf q*
*quantify-bdd 0* (*Branch l r*) = *bdd-binop bv-or l r*
*quantify-bdd* (*Suc i*) (*Branch l r*) =
  *Branch* (*quantify-bdd i l*) (*quantify-bdd i r*)

# Projection




*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* $\Rightarrow$ *bool list bdd* $\Rightarrow$ *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 — Continued

## Application to all states of NFA

$quantify\text{-}nfa :: nat \Rightarrow nfa \Rightarrow nfa$

$quantify\text{-}nfa\ i \equiv \lambda(bdds, as).\ (map\ (quantify\text{-}bdd\ i)\ bdds, as)$

# Projection — Continued

## Application to all states of NFA

*quantify-nfa* :: *nat* $\Rightarrow$ *nfa* $\Rightarrow$ *nfa*

*quantify-nfa i* $\equiv \lambda$(*bdds*, *as*). (*map* (*quantify-bdd i*) *bdds*, *as*)

## Theorem (Correctness)

*If wf-nfa A* (*Suc n*) *and* $i \leq n$ *and* $\forall$ *bs*$\in$*bss*. *is-alph n bs then*
*nfa-accepts* (*quantify-nfa i A*) *bss* =
($\exists$ *bs*. *nfa-accepts A* (*insertll i bs bss*) $\wedge$ $|bs| = |bss|$).

# Projection — Continued

## Application to all states of NFA

*quantify-nfa* :: *nat* $\Rightarrow$ *nfa* $\Rightarrow$ *nfa*

*quantify-nfa i* $\equiv \lambda$(*bdds*, *as*). (*map* (*quantify-bdd i*) *bdds*, *as*)

## Theorem (Correctness)

*If wf-nfa A* (*Suc n*) *and* $i \leq n$ *and* $\forall$ *bs*$\in$*bss*. *is-alph n bs then*
*nfa-accepts* (*quantify-nfa i A*) *bss* =
($\exists$ *bs*. *nfa-accepts A* (*insertll i bs bss*) $\wedge$ $|bs| = |bss|$).

**Note**

- Word accepted by resulting NFA must have same length as witness *bs*

# Projection — Continued

## Application to all states of NFA

*quantify-nfa* :: *nat* $\Rightarrow$ *nfa* $\Rightarrow$ *nfa*

*quantify-nfa i* $\equiv \lambda$(*bdds*, *as*). (*map* (*quantify-bdd i*) *bdds*, *as*)

## Theorem (Correctness)

*If wf-nfa A* (*Suc n*) *and* $i \leq n$ *and* $\forall$ *bs*$\in$*bss*. *is-alph n bs then*
*nfa-accepts* (*quantify-nfa i A*) *bss* =
($\exists$ *bs*. *nfa-accepts A* (*insertll i bs bss*) $\wedge$ $|bs| = |bss|$).

**Note**

- Word accepted by resulting NFA must have same length as witness *bs*
- Requirement can be satisfied by appending zero vectors to end of *bss*

# Projection — Continued

## Application to all states of NFA

*quantify-nfa* :: *nat* $\Rightarrow$ *nfa* $\Rightarrow$ *nfa*

*quantify-nfa i* $\equiv \lambda$(*bdds*, *as*). (*map* (*quantify-bdd i*) *bdds*, *as*)

## Theorem (Correctness)

*If wf-nfa A* (*Suc n*) *and* $i \leq n$ *and* $\forall$ *bs*$\in$*bss*. *is-alph n bs then*
*nfa-accepts* (*quantify-nfa i A*) *bss* =
($\exists$ *bs*. *nfa-accepts A* (*insertll i bs bss*) $\wedge$ $|bs| = |bss|$).

**Note**

- Word accepted by resulting NFA must have same length as witness *bs*
- Requirement can be satisfied by appending zero vectors to end of *bss*
- Use right quotient construction to get automaton that accepts words without trailing zeros

# DFA of NFA

## Instantiation of DFS

node: *bool list*

node store: $\underbrace{\textit{nat option bdd}}_{\text{mapping } \textit{fin. set} \Rightarrow \textit{nat}} \times \underbrace{\textit{bool list list}}_{\text{mapping } \textit{nat} \Rightarrow \textit{fin. set}}$

*subset-dfs* :: *nfa* $\Rightarrow$ *bool list* $\Rightarrow$ *nat option bdd* $\times$ *bool list list*

*subset-dfs A x* $\equiv$

*gen-dfs* (*subset-succs A*) *subset-ins subset-memb subset-empt* [*x*]

# DFA of NFA

## Instantiation of DFS

node: *bool list*

node store: $\underbrace{\textit{nat option bdd}}_{\text{mapping } \textit{fin. set} \Rightarrow \textit{nat}} \times \underbrace{\textit{bool list list}}_{\text{mapping } \textit{nat} \Rightarrow \textit{fin. set}}$

*subset-dfs* :: *nfa* $\Rightarrow$ *bool list* $\Rightarrow$ *nat option bdd* $\times$ *bool list list*
*subset-dfs A x* $\equiv$
*gen-dfs* (*subset-succs A*) *subset-ins subset-memb subset-empt* [*x*]

*det-nfa* :: *nfa* $\Rightarrow$ *dfa*
*det-nfa A* $\equiv$
*let* (*bdd*, *qss*) = *subset-dfs A* (*nfa-startnode A*)
*in* (*map* ($\lambda$*qs*. *bdd-map* ($\lambda$*qs*. *the* (*bdd-lookup bdd qs*))
(*subsetbdd* (*fst A*) *qs* (*nfa-emptybdd* |*qs*|)))
*qss*,
*map* (*nfa-accepting A*) *qss*)

# Diophantine Equations

## Method by Boudet and Comon

$$
\begin{aligned}
&(\mathit{eval\text{-}dioph}\ \mathit{ks}\ \mathit{xs} = \mathit{l}) = \\
&(\mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ \mathit{xs})\ \mathit{mod}\ 2 = \mathit{l}\ \mathit{mod}\ 2 \wedge \\
&\ \mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{div}\ 2)\ \mathit{xs}) = \\
&\ (\mathit{l} - \mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ \mathit{xs}))\ \mathit{div}\ 2)
\end{aligned}
$$

# Diophantine Equations

## Method by Boudet and Comon

$$(\mathit{eval\text{-}dioph}\ \mathit{ks}\ \mathit{xs} = \mathit{l}) =$$
$$(\mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ \mathit{xs})\ \mathit{mod}\ 2 = \mathit{l}\ \mathit{mod}\ 2 \wedge$$
$$\mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{div}\ 2)\ \mathit{xs}) =$$
$$(\mathit{l} - \mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ \mathit{xs}))\ \mathit{div}\ 2)$$

*xs* is a solution iff...

- ...it is a solution modulo *2*, and...

# Diophantine Equations

## Method by Boudet and Comon

$$(\mathit{eval\text{-}dioph}\ \mathit{ks}\ \mathit{xs} = \mathit{l}) =$$
$$(\mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ \mathit{xs})\ \mathit{mod}\ 2 = \mathit{l}\ \mathit{mod}\ 2 \wedge$$
$$\mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{div}\ 2)\ \mathit{xs}) =$$
$$(\mathit{l} - \mathit{eval\text{-}dioph}\ \mathit{ks}\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ \mathit{xs}))\ \mathit{div}\ 2)$$

*xs* is a solution iff...

- ...it is a solution modulo *2*, and...
- ...quotient of *xs* and *2* is a solution of another equation with same coefficients, but different right-hand side.

# Diophantine Equations

## Method by Boudet and Comon

$$(\mathit{eval\text{-}dioph}\ ks\ xs = l) =$$
$$(\mathit{eval\text{-}dioph}\ ks\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ xs)\ \mathit{mod}\ 2 = l\ \mathit{mod}\ 2 \wedge$$
$$\mathit{eval\text{-}dioph}\ ks\ (\mathit{map}\ (\lambda x.\ x\ \mathit{div}\ 2)\ xs) =$$
$$(l - \mathit{eval\text{-}dioph}\ ks\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ xs))\ \mathit{div}\ 2)$$

$xs$ is a solution iff...

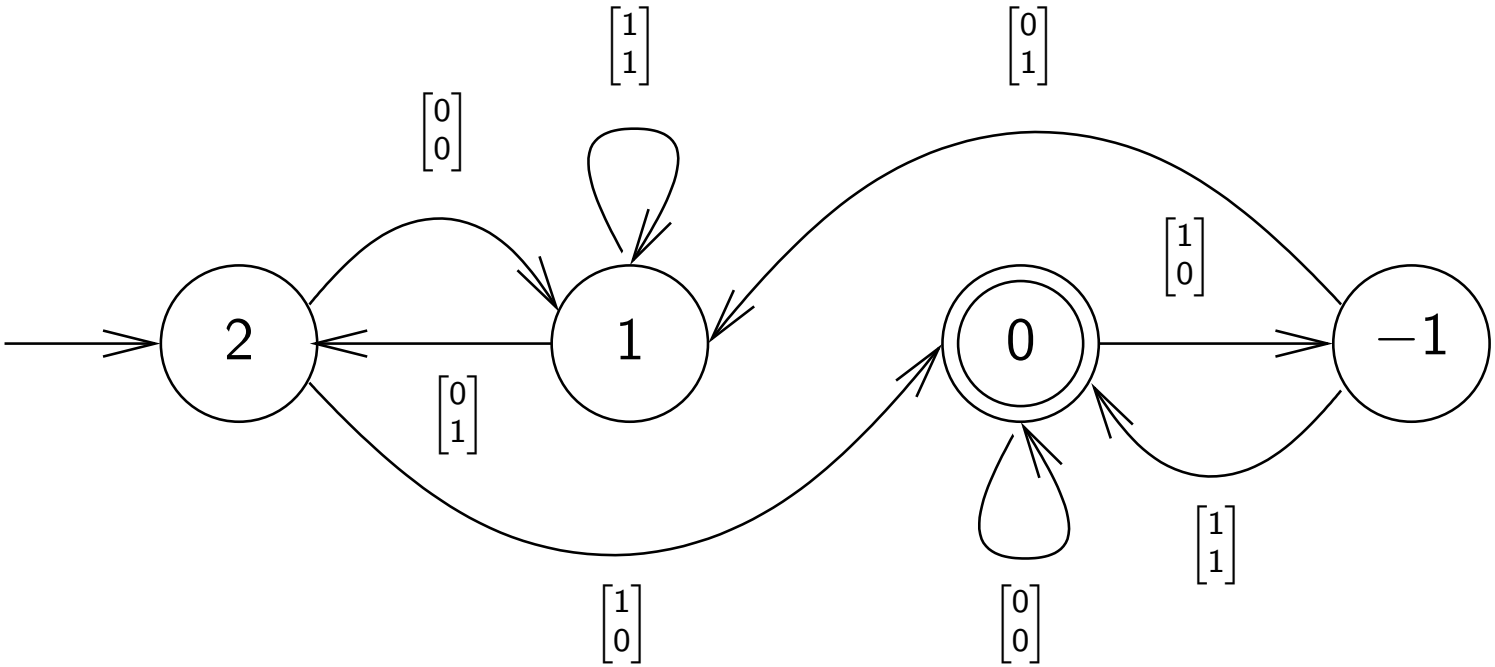
- ...it is a solution modulo $2$, and...
- ...quotient of $xs$ and $2$ is a solution of another equation with same coefficients, but different right-hand side.

## Reachable right-hand sides are bounded

$$\mathit{If}\ |m| \leq \mathit{max}\ |l|\ (\sum k{\leftarrow}ks.\ |k|)\ \ \mathit{then}$$
$$|(m - \mathit{eval\text{-}dioph}\ ks\ (\mathit{map}\ (\lambda x.\ x\ \mathit{mod}\ 2)\ xs))\ \mathit{div}\ 2|$$
$$\leq \mathit{max}\ |l|\ (\sum k{\leftarrow}ks.\ |k|).$$

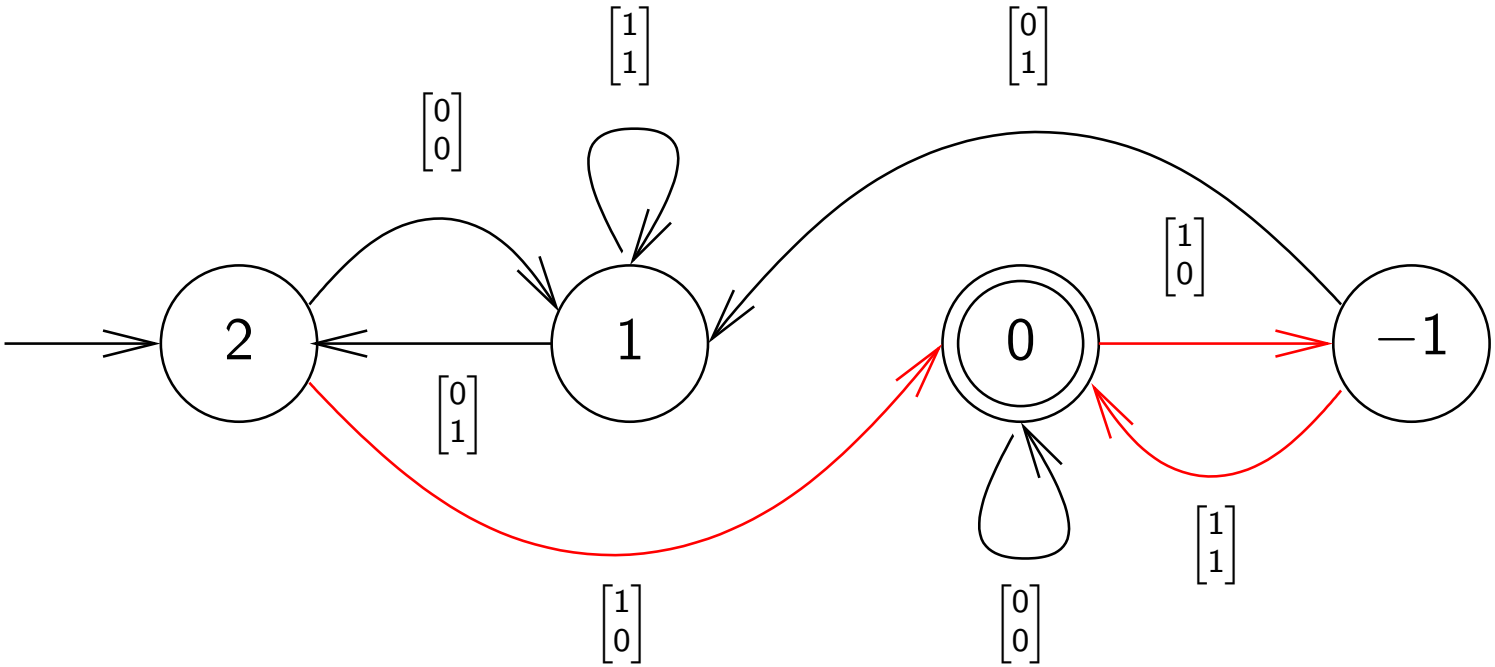
# Diophantine Equations — Example

**Formula:** $2x - 3y = 2$



# Diophantine Equations — Example

**Formula:** $2x - 3y = 2$

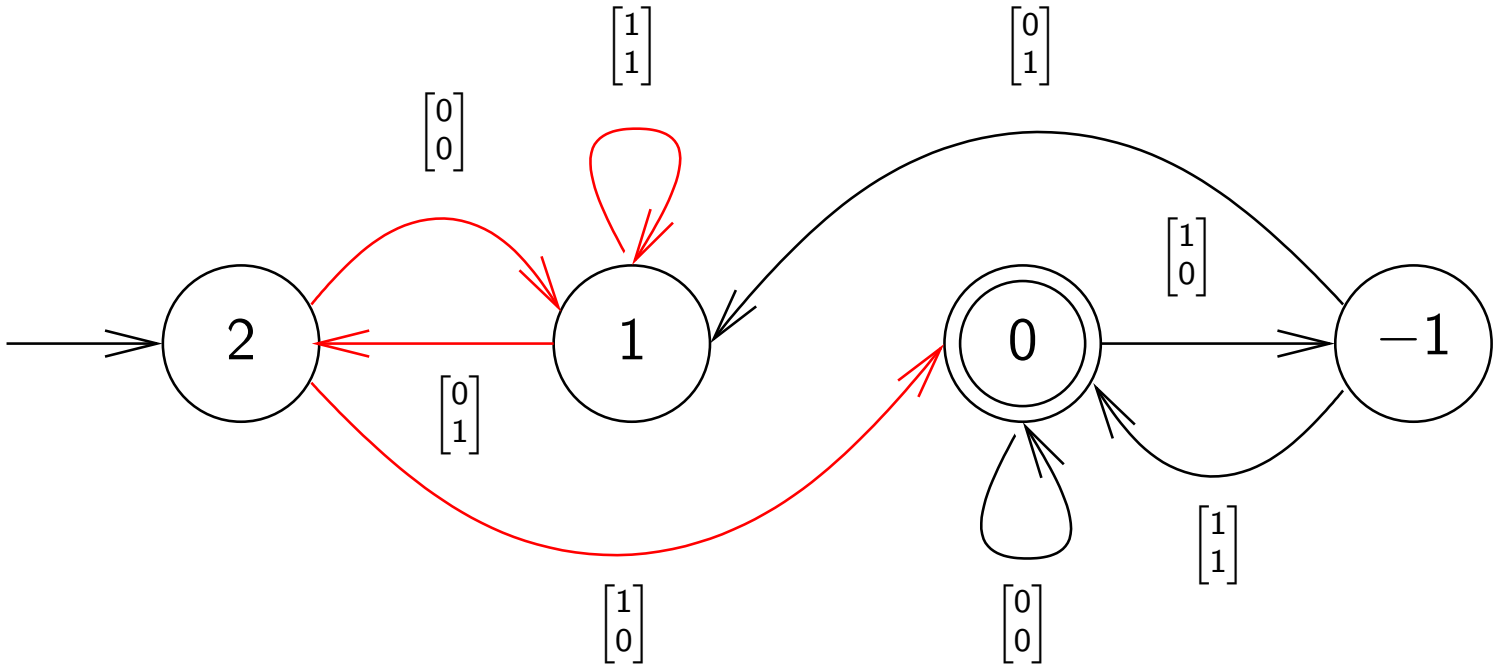



## Some solutions

| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | | | | | | $7 \cdot 2 = 14$ |
| 0 | 0 | 1 | | | | | | $4 \cdot 3 = 12$ |

# Diophantine Equations — Example

**Formula:** $2x - 3y = 2$

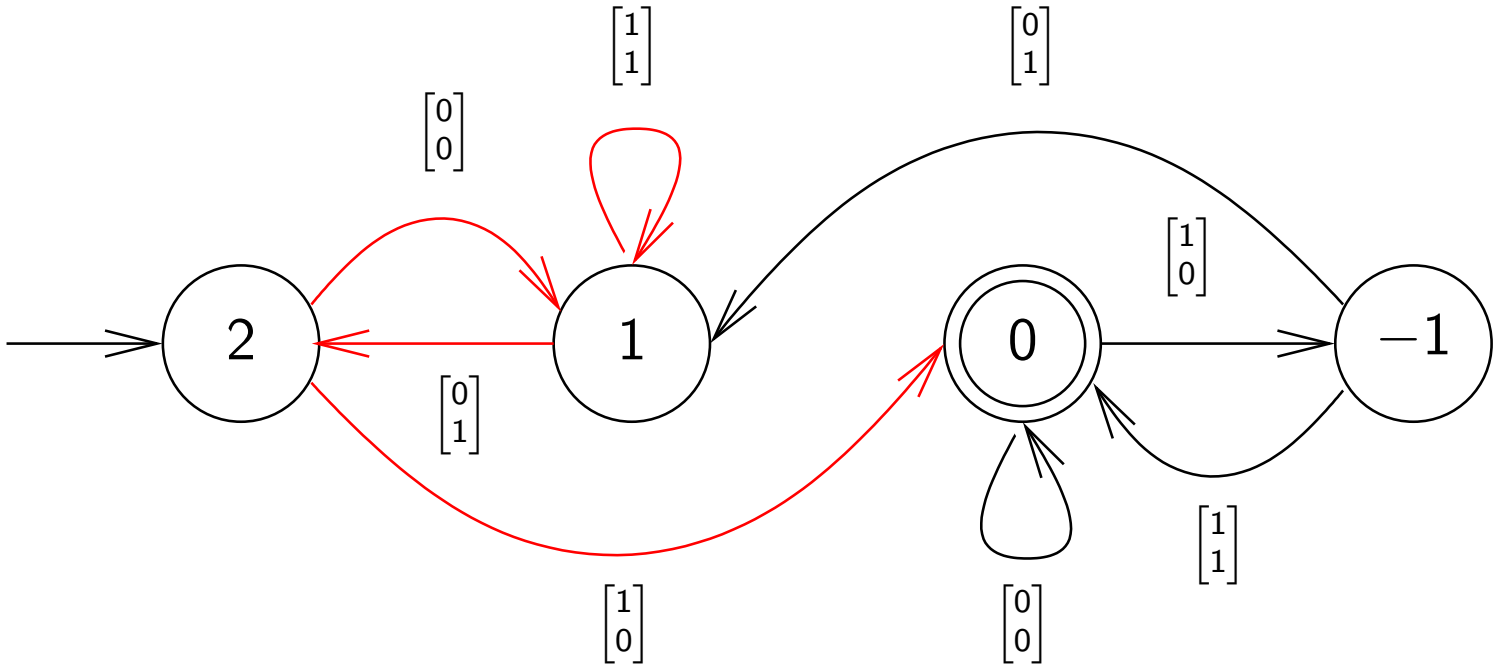



## Some solutions

| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | | | | $22 \cdot 2 = 44$ |
| 0 | 1 | 1 | 1 | 0 | | | | $14 \cdot 3 = 42$ |

# Diophantine Equations — Example

**Formula:** $2x - 3y = 2$

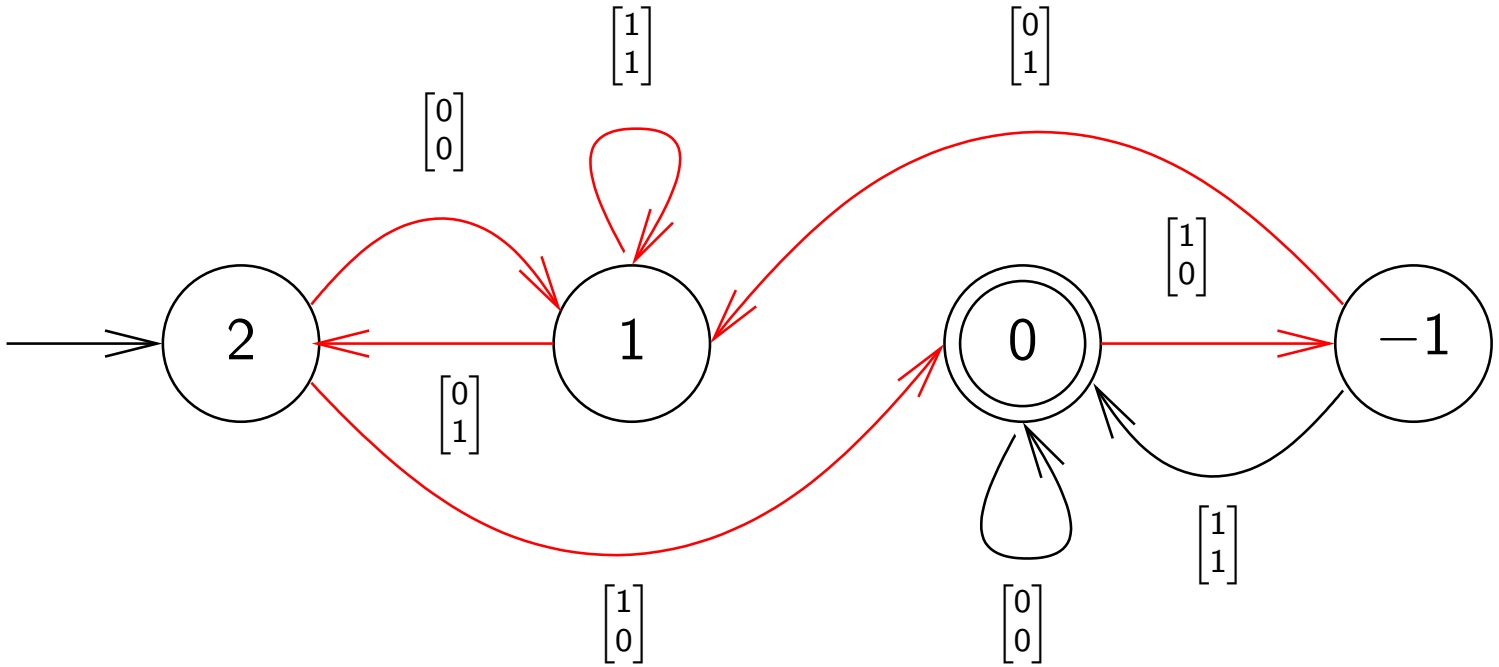



## Some solutions

| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | | | | | $10 \cdot 2 = 20$ |
| 0 | 1 | 1 | 0 | | | | | $6 \cdot 3 = 18$ |

# Diophantine Equations — Example

**Formula:** $2x - 3y = 2$




## Some solutions

| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | $154 \cdot 2 = 308$ |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | $102 \cdot 3 = 306$ |

# Diophantine Equations — The Code

```
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])
```

# Diophantine Equations — Correctness

## Strengthened statement

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

# Diophantine Equations — Correctness

## Strengthened statement

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

## Corollary $(l = m)$

*If* $\forall\, bs \in bss.\ is\text{-}alph\ n\ bs$ *then*
*dfa-accepts* $(eq\text{-}dfa\ n\ ks\ l)\ bss =$
$(eval\text{-}dioph\ ks\ (nats\text{-}of\text{-}boolss\ n\ bss) = l)$.

# Outline

# 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

$$
\begin{aligned}
&\textit{dfa-of-pf} :: \textit{nat} \Rightarrow \textit{pf} \Rightarrow \textit{dfa}\\
&\textit{dfa-of-pf}\ n\ (\textit{Eq}\ \textit{ks}\ l) = \textit{eq-dfa}\ n\ \textit{ks}\ l\\
&\textit{dfa-of-pf}\ n\ (\textit{Le}\ \textit{ks}\ l) = \textit{ineq-dfa}\ n\ \textit{ks}\ l\\
&\textit{dfa-of-pf}\ n\ (\textit{Neg}\ p) = \textit{negate-dfa}\ (\textit{dfa-of-pf}\ n\ p)\\
&\textit{dfa-of-pf}\ n\ (\textit{And}\ p\ q) =\\
&\quad \textit{binop-dfa}\ (\wedge)\ (\textit{dfa-of-pf}\ n\ p)\ (\textit{dfa-of-pf}\ n\ q)\\
&\textit{dfa-of-pf}\ n\ (\textit{Or}\ p\ q) =\\
&\quad \textit{binop-dfa}\ (\vee)\ (\textit{dfa-of-pf}\ n\ p)\ (\textit{dfa-of-pf}\ n\ q)\\
&\textit{dfa-of-pf}\ n\ (\textit{Exist}\ p) =\\
&\quad \textit{rquot}\ (\textit{det-nfa}\ (\textit{quantify-nfa}\ 0\ (\textit{nfa-of-dfa}\ (\textit{dfa-of-pf}\ (\textit{Suc}\ n)\ p))))\\
&\quad\ n\\
&\textit{dfa-of-pf}\ n\ (\textit{Forall}\ p) = \textit{dfa-of-pf}\ n\ (\textit{Neg}\ (\textit{Exist}\ (\textit{Neg}\ p)))
\end{aligned}
$$

**Theorem (Correctness)**

$$
\begin{aligned}
&\textit{If}\ \forall\, \textit{bs} \in \textit{bss}.\ \textit{is-alph}\ n\ \textit{bs}\ \ \textit{then}\\
&\textit{dfa-accepts}\ (\textit{dfa-of-pf}\ n\ p)\ \textit{bss} = \textit{eval-pf}\ p\ (\textit{nats-of-boolss}\ n\ \textit{bss}).
\end{aligned}
$$

# Correctness Proof — Existential Quantifier

*dfa-accepts* (*rquot* (*det-nfa* (*quantify-nfa 0* (*nfa-of-dfa* (*dfa-of-pf* (*Suc n*) *p*)))) *n*) *bss*

# Correctness Proof — Existential Quantifier

$$\mathit{dfa\text{-}accepts}\ (\mathit{rquot}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ n)\ \mathit{bss}$$

$\longleftrightarrow$ {Correctness statement for *rquot*}

$$\exists\, m.\ \mathit{dfa\text{-}accepts}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)$$

# Correctness Proof — Existential Quantifier

$$\mathit{dfa\text{-}accepts}\ (\mathit{rquot}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ n)\ \mathit{bss}$$

$\longleftrightarrow$ {Correctness statement for *rquot*}

$$\exists\, m.\ \mathit{dfa\text{-}accepts}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)$$

$\longleftrightarrow$ {Correctness of *det-nfa*, *quantify-nfa* and *nfa-of-dfa*}

$$\exists\, m\ \mathit{bs}.\ \mathit{dfa\text{-}accepts}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

# Correctness Proof — Existential Quantifier

$$\mathit{dfa\text{-}accepts}\ (\mathit{rquot}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ n)\ \mathit{bss}$$

$\longleftrightarrow$ {Correctness statement for *rquot*}

$$\exists\, m.\ \mathit{dfa\text{-}accepts}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)$$

$\longleftrightarrow$ {Correctness of *det-nfa*, *quantify-nfa* and *nfa-of-dfa*}

$$\exists\, m\ \mathit{bs}.\ \mathit{dfa\text{-}accepts}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

$\longleftrightarrow$ {Induction hypothesis}

$$\exists\, m\ \mathit{bs}.\ \mathit{eval\text{-}pf}\ p\ (\mathit{nats\text{-}of\text{-}boolss}\ (\mathit{Suc}\ n)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n))) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

# Correctness Proof — Existential Quantifier

$$\mathit{dfa\text{-}accepts}\ (\mathit{rquot}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ n)\ \mathit{bss}$$

$\longleftrightarrow$ {Correctness statement for *rquot*}

$$\exists\, m.\ \mathit{dfa\text{-}accepts}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)$$

$\longleftrightarrow$ {Correctness of *det-nfa*, *quantify-nfa* and *nfa-of-dfa*}

$$\exists\, m\ \mathit{bs}.\ \mathit{dfa\text{-}accepts}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n)) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

$\longleftrightarrow$ {Induction hypothesis}

$$\exists\, m\ \mathit{bs}.\ \mathit{eval\text{-}pf}\ p\ (\mathit{nats\text{-}of\text{-}boolss}\ (\mathit{Suc}\ n)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} \mathbin{@} \mathit{zeros}\ m\ n))) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

$\longleftrightarrow$ {Properties of *nats-of-boolss*}

$$\exists\, m\ \mathit{bs}.\ \mathit{eval\text{-}pf}\ p\ (\mathit{nat\text{-}of\text{-}bools}\ \mathit{bs} \cdot \mathit{nats\text{-}of\text{-}boolss}\ n\ \mathit{bss}) \wedge |\mathit{bs}| = |\mathit{bss}| + m$$

# Correctness Proof — Existential Quantifier

$$\mathit{dfa\text{-}accepts}\ (\mathit{rquot}\ (\mathit{det\text{-}nfa}\ (\mathit{quantify\text{-}nfa}\ 0\ (\mathit{nfa\text{-}of\text{-}dfa}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p))))\ n)\ \mathit{bss}$$

$\longleftrightarrow$ {Correctness of *det-nfa*, *quantify-nfa* and *nfa-of-dfa*}

$$\exists\, m\ \mathit{bs}.\ \mathit{dfa\text{-}accepts}\ (\mathit{dfa\text{-}of\text{-}pf}\ (\mathit{Suc}\ n)\ p)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} @ \mathit{zeros}\ m\ n)) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

$\longleftrightarrow$ {Induction hypothesis}

$$\exists\, m\ \mathit{bs}.\ \mathit{eval\text{-}pf}\ p\ (\mathit{nats\text{-}of\text{-}boolss}\ (\mathit{Suc}\ n)\ (\mathit{insertll}\ 0\ \mathit{bs}\ (\mathit{bss} @ \mathit{zeros}\ m\ n))) \wedge |\mathit{bs}| = |\mathit{bss}| + |\mathit{zeros}\ m\ n|$$

$\longleftrightarrow$ {Properties of *nats-of-boolss*}

$$\exists\, m\ \mathit{bs}.\ \mathit{eval\text{-}pf}\ p\ (\mathit{nat\text{-}of\text{-}bools}\ \mathit{bs} \cdot \mathit{nats\text{-}of\text{-}boolss}\ n\ \mathit{bss}) \wedge |\mathit{bs}| = |\mathit{bss}| + m$$

$\longleftrightarrow$ {Produce suitable instantiations for $m$ and $bs$ from $x$}

$$\exists\, x.\ \mathit{eval\text{-}pf}\ p\ (x \cdot \mathit{nats\text{-}of\text{-}boolss}\ n\ \mathit{bss})$$

# Outline

# Performance

- Algorithm can compete quite well with standard decision procedure for Presburger arithmetic available in Isabelle.

# 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

# 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>
<tr><td rowspan="2">Forall<br>6</td><td rowspan="2">Imp<br>15</td><td>Exist<br>13</td><td>Exist<br>9</td><td>Eq [5, 3, −1] 0<br>9</td></tr>
<tr><td colspan="3">Le [−1] −8<br>5</td></tr>
</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>
<tr><td rowspan="2">$Forall$<br>**6**</td><td rowspan="2">$Imp$<br>**15**</td><td>$Exist$<br>**13**</td><td>$Exist$<br>**9**</td><td colspan="2">$Eq\ [5, 3, -1]\ 0$<br>**9**</td></tr>
<tr><td colspan="4">$Le\ [-1]\ -8$<br>**5**</td></tr>
</table>

- Use of DFS pays off!

# Future Work

- Minimization algorithm

# Future Work

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

# Future Work

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

# Future Work

- Minimization algorithm
- 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

# Future Work

- Minimization algorithm
- 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

- Minimization algorithm
- 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

- Minimization algorithm
- 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

- Minimization algorithm
- 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

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