The Extended Theory of Trees and Algebraic (Co)datatypes

Fabian Zaiser Luke Ong

Department of Computer Science

UNIVERSITY OF OXFORD

SMT@CAV 2021

Algebraic datatypes (inductive datatypes)

```
data   nat    = zero   | succ(pred: nat)
data   list   = nil    | cons(head: nat,  tail: list)
```

Algebraic datatypes (inductive datatypes)

```
data    nat     = zero   | succ(pred: nat)
data    list    = nil    | cons(head: nat,  tail: list)
```

Constructors: `zero`, `succ`, `nil`, `cons`

Selectors: `pred`, `head`, `tail`

Algebraic datatypes (inductive datatypes)

```
data    nat     = zero    | succ(pred: nat)
data    list    = nil     | cons(head: nat,  tail: list)
```

Constructors: `zero`, `succ`, `nil`, `cons`

Selectors: `pred`, `head`, `tail`

`head(cons(x, y))` $=$ `x`

SMT-LIB standard

SMT-LIB syntax:

```
(declare-datatypes
  ((nat 0)(list 0)) (
    ((zero)    (succ (pred nat)                 ))
    ((nil)     (cons (head nat) (tail list)))
  ))
```

Q: *What is* `head(nil)`*?*

SMT-LIB standard

SMT-LIB syntax:

```
(declare-datatypes
  ((nat 0)(list 0)) (
    ((zero)    (succ (pred nat)           ))
    ((nil)     (cons (head nat) (tail list)))
  ))
```

Q: *What is* `head(nil)`*?*

A: *result unspecified, but must be consistent*

SMT-LIB standard

SMT-LIB syntax:

```
(declare-datatypes
  ((nat 0)(list 0)) (
    ((zero)    (succ (pred nat)             ))
    ((nil)     (cons (head nat) (tail list)))
  ))
```

Q: *What is* `head(nil)`*?*

A: *result unspecified, but must be consistent*

$\leadsto$ quantified formulae become undecidable!

Algebraic **Co**datatypes (coinductive datatypes)

```
codata list = nil | cons(head: nat, tail: list)

let t = cons(zero, t)
```

→ allow infinite inhabitants: `t = cons(zero, cons(zero, ...))`

Algebraic **Co**datatypes (coinductive datatypes)

```
codata list = nil | cons(head: nat, tail: list)

let t = cons(zero, t)
```

→ allow infinite inhabitants: `t = cons(zero, cons(zero, ...))`

Why?

- dual of inductive datatypes
- lazily evaluated infinite objects (e.g. in Haskell)
- useful in theorem provers (e.g. Coq, Lean)

Algebraic **Co**datatypes (coinductive datatypes)

```
codata list = nil | cons(head: nat, tail: list)

let t = cons(zero, t)
```

→ allow infinite inhabitants: `t = cons(zero, cons(zero, ...))`

Why?

- dual of inductive datatypes
- lazily evaluated infinite objects (e.g. in Haskell)
- useful in theorem provers (e.g. Coq, Lean)

Supported for SMT?

✗ in SMT-LIB standard

✓ CVC4 (but not many more solvers)

(Co)datatypes: Pros/Cons

- ✓ datatypes part of SMT-LIB
- ✓ good support for datatypes

(Co)datatypes: Pros/Cons

✓ datatypes part of SMT-LIB

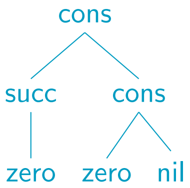
✓ good support for datatypes

✗ lacking support for codatatypes

✗ static finite/infinite separation

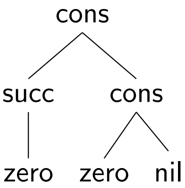
✗ first-order theory is undecidable

A different perspective: Trees

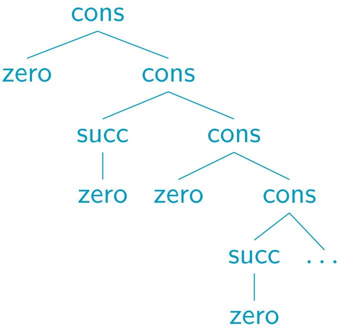


$x = \mathsf{cons}(\mathsf{succ}(\mathsf{zero}), \mathsf{cons}(\mathsf{zero}, \mathsf{nil}))$

A different perspective: Trees



$$x = \mathsf{cons}(\mathsf{succ}(\mathsf{zero}), \mathsf{cons}(\mathsf{zero}, \mathsf{nil}))$$



$$x = \mathsf{cons}(\mathsf{zero}, \mathsf{cons}(\mathsf{succ}(\mathsf{zero}), x))$$

First-order Theory of Trees

only constructors, no selectors

First-order Theory of Trees

only constructors, no selectors

- Decision procedures: MAHER 1988 and COMON & LESCANNE 1989 (independently)

First-order Theory of Trees

only constructors, no selectors

- Decision procedures: MAHER 1988 and COMON & LESCANNE 1989 (independently)
- applications:
  - matching and unification
  - semantics of logic & functional programs
  - recursion schemes
  - verification of programs
  - term rewriting systems
  - **in this talk:** algebraic (co)datatypes

An **Extended** Theory of Trees

DJELLOUL, DAO, FRÜHWIRTH (2008)

add a **finiteness predicate**: $\operatorname{fin}(x)$

$\rightsquigarrow$ can reason about finite & infinite trees *within the same theory*

An **Extended** Theory of Trees

DJELLOUL, DAO, FRÜHWIRTH (2008)

add a **finiteness predicate**: $\mathrm{fin}(x)$

$\rightsquigarrow$ can reason about finite & infinite trees *within the same theory*

- they gave a decision procedure
- outputs a simplified formula
- easy to read off all models

Good properties

The (Extended) First-Order Theory of Trees . . .

- ✓ has seen many applications
- ✓ is decidable
- ✓ even admits a simplification procedure
- ✓ can be used to solve formulae in the theory of (co)datatypes*
- ✓ is more expressive than (co)datatypes

Good properties

The (Extended) First-Order Theory of Trees . . .

- ✓ has seen many applications
- ✓ is decidable
- ✓ even admits a simplification procedure
- ✓ can be used to solve formulae in the theory of (co)datatypes*
- ✓ is more expressive than (co)datatypes

$\Longrightarrow$ **interesting theory for the SMT community**

Contributions

- explain relationship between (co)datatypes and trees

Contributions

- explain relationship between (co)datatypes and trees
- simplification procedure/constraint solver
  - allowing quantifiers
  - based on DJELLOUL, DAO, FRÜHWIRTH (2008) …
  - …but allowing finitely many constructors
  - …which is necessary for (co)datatypes!

Contributions

- explain relationship between (co)datatypes and trees
- simplification procedure/constraint solver
  - allowing quantifiers
  - based on DJELLOUL, DAO, FRÜHWIRTH (2008) ...
  - ...but allowing finitely many constructors
  - ...which is necessary for (co)datatypes!
- prototype implementation
- evaluation on SMT-LIB

Contributions

- explain relationship between (co)datatypes and trees
- simplification procedure/constraint solver
  - allowing quantifiers
  - based on DJELLOUL, DAO, FRÜHWIRTH (2008) ...
  - ... but allowing finitely many constructors
  - ... which is necessary for (co)datatypes!
- prototype implementation
- evaluation on SMT-LIB

$\Longrightarrow$ Extended Theory of Trees is useful and decidable

Part II: Relationship between (Co)datatypes and Trees

Expressivity

Concept	Trees	(Co)datatypes

Expressivity

Concept	Trees	(Co)datatypes
x is finite	$\mathrm{fin}(x)$	x is datatype
x is finite or infinite	(default)	x is codatatype

Expressivity

Concept	Trees	(Co)datatypes
x is finite	$\mathrm{fin}(x)$	x is datatype
x is finite or infinite	(default)	x is codatatype
x is infinite	$\neg\,\mathrm{fin}(x)$	✗
x is conditionally finite	$\phi \to \mathrm{fin}(x)$	✗

Expressivity

Concept	Trees	(Co)datatypes
x is finite	$\mathrm{fin}(x)$	x is datatype
x is finite or infinite	(default)	x is codatatype
x is infinite	$\neg\,\mathrm{fin}(x)$	✗
x is conditionally finite	$\phi \to \mathrm{fin}(x)$	✗

$\Longrightarrow$ Trees more expressive than (co)datatypes

Expressivity

Concept	Trees	(Co)datatypes
x is finite	$\mathrm{fin}(x)$	x is datatype
x is finite or infinite	(default)	x is codatatype
x is infinite	$\neg\,\mathrm{fin}(x)$	✗
x is conditionally finite	$\phi \to \mathrm{fin}(x)$	✗

$\Longrightarrow$ Trees more expressive than (co)datatypes

. . . but datatypes have selectors?

Expressivity

Concept	**Trees**	**(Co)datatypes**
x is finite	$\mathrm{fin}(x)$	x is datatype
x is finite or infinite	(default)	x is codatatype
x is infinite	$\neg\,\mathrm{fin}(x)$	✗
x is conditionally finite	$\phi \to \mathrm{fin}(x)$	✗

$\Longrightarrow$ Trees more expressive than (co)datatypes

. . . but datatypes have selectors? We can get rid of them . . .

Translating (quantifier-free) (co)datatypes to trees

Input: $tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations

Input: $tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$

Output: $tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations

Input: $tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$

Output: $\exists x, y.\quad x = \mathsf{cons}(\mathsf{zero}, y)$
$\wedge \quad x = tail(v)$
$\wedge \quad y = tail(w)$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence

$$\text{Input: } tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$$

$$\begin{aligned}\text{Output: } \exists x, y. \quad & x = \mathsf{cons}(\mathsf{zero}, \; y \;) \\ \wedge \quad & x = tail(v) \\ \wedge \quad & y = tail(w)\end{aligned}$$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence

Input: $tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$

$$\begin{aligned} \text{Output: } \exists x, y.\quad & x = \mathsf{cons}(\mathsf{zero}, y) \\ \wedge\quad & x = tail(v) \\ \wedge\quad & y = tail(w) \\ & \wedge\, (v = w \to x = y) \end{aligned}$$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence
3. eliminate selectors: $x = tail(t) \rightsquigarrow \forall y, z.\, t = \mathsf{cons}(y, z) \to x = z$

$$\text{Input: } tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$$

$$\begin{aligned}\text{Output: } \exists x, y.\quad & x = \mathsf{cons}(\mathsf{zero}, y) \\ \wedge\quad & x = tail(v) \\ \wedge\quad & y = tail(w) \\ \wedge\quad & (v = w \to x = y)\end{aligned}$$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence
3. eliminate selectors: $x = tail(t) \rightsquigarrow \forall y, z.\, t = \mathsf{cons}(y, z) \to x = z$

$$\text{Input: } tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$$

$$\begin{aligned}\text{Output: } \exists x, y.\quad & x \quad = \mathsf{cons}(\mathsf{zero}, \quad y \quad) \\ \wedge\ & (\forall a, b.\, v = \mathsf{cons}(a, b) \to x = b) \\ \wedge\ & (\forall c, d.\, w = \mathsf{cons}(c, d) \to y = d) \\ \wedge\ & (v = w \to x = y)\end{aligned}$$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence
3. eliminate selectors: $x = tail(t) \rightsquigarrow \forall y, z.\, t = \mathsf{cons}(y, z) \rightarrow x = z$
4. add fin for datatypes

$$\text{Input: } tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$$

$$\begin{aligned} \text{Output: } \exists x, y.\quad & x = \mathsf{cons}(\mathsf{zero}, \quad y \quad) \\ & \wedge (\forall a, b.\, v = \mathsf{cons}(a, b) \rightarrow x = b) \\ & \wedge (\forall c, d.\, w = \mathsf{cons}(c, d) \rightarrow y = d) \\ & \wedge (v = w \rightarrow x = y) \end{aligned}$$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence
3. eliminate selectors: $x = tail(t) \rightsquigarrow \forall y, z.\, t = \mathsf{cons}(y, z) \rightarrow x = z$
4. add fin for datatypes

$$\text{Input: } tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$$

$$\begin{aligned}\text{Output: } \exists x, y.\quad & x = \mathsf{cons}(\mathsf{zero}, y) \\ & \wedge (\forall a, b.\, v = \mathsf{cons}(a, b) \rightarrow x = b) \\ & \wedge (\forall c, d.\, w = \mathsf{cons}(c, d) \rightarrow y = d) \\ & \wedge (v = w \rightarrow x = y) \\ & \wedge \mathrm{fin}(v) \wedge \mathrm{fin}(w)\end{aligned}$$

Translating (quantifier-free) (co)datatypes to trees

1. extract selectors to simple equations
2. add equations for congruence
3. eliminate selectors: $x = tail(t) \leadsto \forall y, z.\, t = \mathsf{cons}(y, z) \to x = z$
4. add fin for datatypes

$$\text{Input: } tail(v) = \mathsf{cons}(\mathsf{zero}, tail(w))$$

$$\begin{aligned}\text{Output: } \exists x, y.\quad x \quad &= \mathsf{cons}(\mathsf{zero}, \quad y \quad) \\ \wedge\, (\forall a, b.\, v &= \mathsf{cons}(a, b) \to x = b) \\ \wedge\, (\forall c, d.\, w &= \mathsf{cons}(c, d) \to y = d) \\ \wedge\, (v = w &\to x = y) \\ \wedge\, \mathrm{fin}(v) &\wedge \mathrm{fin}(w)\end{aligned}$$

Theorem (for quantifier-free formulae)

The result is equisatisfiable in the Extended Theory of Trees.

Translating quantified formulae

Q: What about quantifiers?

Translating quantified formulae

Q: What about quantifiers?

A: impossible (because of undecidability)

Translating quantified formulae

Q: What about quantifiers?

A: impossible (because of undecidability)

Q: What if we change the semantics of selectors?

Translating quantified formulae

Q: What about quantifiers?

A: impossible (because of undecidability)

Q: What if we change the semantics of selectors?

A: With default values, it's possible

- selectors must return a fixed default value when applied to the wrong constructor
- e.g. require $tail(\mathsf{nil}) = \mathsf{nil}$.

Translating quantified formulae

Q: What about quantifiers?

A: impossible (because of undecidability)

Q: What if we change the semantics of selectors?

A: With default values, it's possible

- selectors must return a fixed default value when applied to the wrong constructor
- e.g. require $tail(\mathsf{nil}) = \mathsf{nil}$.

Theorem (for selectors with default values)

We can translate any quantified formula in (Co)datatypes to an equisatisfiable one in Trees.

Trees vs (co)datatypes: relationship

✗ due to $\neg\,\mathrm{fin}(x)\ \ldots$

Extended
Theory of Trees

Theory of
(Co)datatypes

✓ in decidable situations

Part III: Decision procedures

Deciding the Extended Theory of Trees

Original algorithm (DJELLOUL, ET AL 2008)

- works on normal forms ϕ:

$$\phi \equiv \exists \bar{x}.\ \alpha \ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction"

nested normal form

Deciding the Extended Theory of Trees

Original algorithm (DJELLOUL, ET AL 2008)

- works on normal forms ϕ:

$$\phi \equiv \exists \bar{x}.\ \alpha \ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction"

nested normal form

- 16 rewrite rules

Deciding the Extended Theory of Trees

Original algorithm (DJELLOUL, ET AL 2008)

- works on normal forms ϕ:

$$\phi \equiv \exists \bar{x}.\ \alpha \ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction" nested normal form

- 16 rewrite rules
- simplified formula: nested only once + more conditions
- Example: $x = \mathsf{succ}(y) \wedge \mathrm{fin}(y) \wedge \neg(\exists w.\, y = \mathsf{succ}(w) \wedge \mathrm{fin}(z))$

Deciding the Extended Theory of Trees

Original algorithm (Djelloul, et al 2008)

- works on normal forms ϕ:

$$\phi \equiv \exists \bar{x}.\ \alpha \ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction"

nested normal form

- 16 rewrite rules
- simplified formula: nested only once + more conditions
- Example: $x = \mathsf{succ}(y) \wedge \mathrm{fin}(y) \wedge \neg(\exists w.\, y = \mathsf{succ}(w) \wedge \mathrm{fin}(z))$
- **Restriction:** infinitely many constructors required

$\rightsquigarrow$ useless for (co)datatypes

- **Our contribution:** lift this restriction

Challenges in the unrestricted setting

Challenges in the unrestricted setting

- case analysis on constructors:
  - $\forall x : nat.\, x = \mathsf{zero} \lor \exists y : nat.\, x = \mathsf{succ}(y) \leadsto$ ✓
  - $\forall y : nat.\, x \neq \mathsf{succ}(y) \leadsto x = \mathsf{zero}$

Challenges in the unrestricted setting

- case analysis on constructors:
  - $\forall x : nat.\, x = \mathsf{zero} \lor \exists y : nat.\, x = \mathsf{succ}(y) \rightsquigarrow$ ✓
  - $\forall y : nat.\, x \neq \mathsf{succ}(y) \rightsquigarrow x = \mathsf{zero}$

- sorts with only (in)finite inhabitants

  $$bool = \mathsf{false} \mid \mathsf{true}$$
  $$inftree = \mathsf{c1}(inftree) \mid \mathsf{c2}(inftree)$$

  - $\forall x : bool.\, \mathrm{fin}(x) \rightsquigarrow$ ✓
  - $\forall x : inftree.\, \mathrm{fin}(x) \rightsquigarrow$ ✗

Challenges in the unrestricted setting

- case analysis on constructors:
  - $\forall x : nat.\, x = \mathsf{zero} \vee \exists y : nat.\, x = \mathsf{succ}(y) \rightsquigarrow$ ✓
  - $\forall y : nat.\, x \neq \mathsf{succ}(y) \rightsquigarrow x = \mathsf{zero}$

- sorts with only (in)finite inhabitants

  $bool = \mathsf{false} \mid \mathsf{true}$
  $inftree = \mathsf{c1}(inftree) \mid \mathsf{c2}(inftree)$

  - $\forall x : bool.\, \mathrm{fin}(x) \rightsquigarrow$ ✓
  - $\forall x : inftree.\, \mathrm{fin}(x) \rightsquigarrow$ ✗

- unique infinite inhabitants
  - $\neg\, \mathrm{fin}(x : nat) \rightsquigarrow x = \mathsf{succ}(x)$

Deciding the Extended Theory of Trees

Our extended algorithm for the general setting

Idea: **case splits**

Deciding the Extended Theory of Trees

Our extended algorithm for the general setting

Idea: **case splits**

- for sorts with finitely many constructors:
  - if $x : nat$ then $x = \mathsf{zero} \vee \exists y. x = \mathsf{succ}(y)$
  - Example: input $\exists x : nat.\, \alpha \wedge \cdots$ is transformed into $(\exists x.\, x = \mathsf{zero} \wedge \alpha \wedge \cdots) \vee (\exists x, y.\, x = \mathsf{succ}(y) \wedge \alpha \wedge \cdots)$

Deciding the Extended Theory of Trees

Our extended algorithm for the general setting

Idea: **case splits**

- for sorts with finitely many constructors:
  - if $x : nat$ then $x = \mathsf{zero} \vee \exists y.x = \mathsf{succ}(y)$
  - Example: input $\exists x : nat.\, \alpha \wedge \cdots$ is transformed into $(\exists x.\, x = \mathsf{zero} \wedge \alpha \wedge \cdots) \vee (\exists x, y.\, x = \mathsf{succ}(y) \wedge \alpha \wedge \cdots)$
- for sorts with finitely many (in)finite inhabitants:
  - if $x : nat$ then $\mathrm{fin}(x) \vee x = \mathsf{succ}(x)$

Deciding the Extended Theory of Trees

Our extended algorithm for the general setting

Idea: **case splits**

- for sorts with finitely many constructors:
  - if $x : nat$ then $x = \mathsf{zero} \vee \exists y. x = \mathsf{succ}(y)$
  - Example: input $\exists x : nat.\, \alpha \wedge \cdots$ is transformed into $(\exists x.\, x = \mathsf{zero} \wedge \alpha \wedge \cdots) \vee (\exists x, y.\, x = \mathsf{succ}(y) \wedge \alpha \wedge \cdots)$
- for sorts with finitely many (in)finite inhabitants:
  - if $x : nat$ then $\mathrm{fin}(x) \vee x = \mathsf{succ}(x)$
- but be clever about when to case split
  - avoid unnecessary work
  - avoid infinite loops

Results

Theorem

- *Our simplification procedure returns a simplified formula that is equivalent in the Extended Theory of Trees.*
- *Simplified formula allows reading off all satisfying assignments of free variables.*
- *If input formula is closed, the result is* true *or* false*.*

$\rightarrow$ proof in the paper

Results

Theorem

- *Our simplification procedure returns a simplified formula that is equivalent in the Extended Theory of Trees.*
- *Simplified formula allows reading off all satisfying assignments of free variables.*
- *If input formula is closed, the result is* true *or* false*.*

$\rightarrow$ proof in the paper

Examples

- $x = \mathsf{succ}(x) \vee \mathrm{fin}(x) \quad \rightsquigarrow \quad \mathrm{true}$
- $x \neq \mathsf{nil} \wedge \mathrm{fin}(x) \quad \rightsquigarrow \quad \exists y, z.\, x = \mathsf{cons}(y, z) \wedge \mathrm{fin}(y) \wedge \mathrm{fin}(z)$

Implementation

- prototype implementation in Scala
- translates (co)datatypes $\rightarrow$ trees
  - standard semantics (SMT-LIB) or
  - selector semantics with default values
- implements the simplification procedure

Try it online! $\rightarrow$ `tinyurl.com/trees-codata`

Evaluation

In theory:

- worst case: non-elementary time complexity
- cannot do better (VOROBYOV 1996)

Evaluation

In theory:

- worst case: non-elementary time complexity
- cannot do better (VOROBYOV 1996)

In practice:

- evaluated on 4000 tests of `QF_DT` suite of the SMT-LIB
- translate from datatypes to trees, then solve
- 90% took < 1 second
- 5% timed out after 10 seconds
- lots of “low-hanging fruit” for improvements

Conclusion

The Extended Theory of Trees is . . .

- useful: for (co)datatypes, logic programming, term rewriting, verification, . . .
- powerful: more expressive than (co)datatypes
- decidable: even admits a simplification procedure

Conclusion

The Extended Theory of Trees is ...

- useful: for (co)datatypes, logic programming, term rewriting, verification, ...
- powerful: more expressive than (co)datatypes
- decidable: even admits a simplification procedure

For details ...

- **Fabian Zaiser**, Luke Ong. *The Extended Theory of Trees and Algebraic (Co)datatypes.* HCVS@ETAPS2020
- Implementation: `tinyurl.com/trees-codata`

Backup slides

The basic idea

Manipulate normal forms ϕ (DJELLOUL, ET AL 2008):

$$\phi \equiv \exists \bar{x}.\ \alpha\ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction"

nested normal form

The basic idea

Manipulate normal forms ϕ (DJELLOUL, ET AL 2008):

$$\phi \equiv \exists \bar{x}.\ \alpha \ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction"

nested normal form

Perform case analysis:

- for sorts with finitely many constructors:
  - if $x : nat$ then $x = \mathsf{zero} \vee \exists y. x = \mathsf{succ}(y)$
  - $\exists x : nat.\, \alpha \wedge \ldots$
    $\rightsquigarrow (\exists x.\, x = \mathsf{zero} \wedge \alpha \wedge \cdots) \vee (\exists x, y.\, x = \mathsf{succ}(y) \wedge \alpha \wedge \cdots)$

The basic idea

Manipulate normal forms ϕ (DJELLOUL, ET AL 2008):

$$\phi \equiv \exists \bar{x}.\ \alpha\ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction"

nested normal form

Perform case analysis:

- for sorts with finitely many constructors:
  - if $x : nat$ then $x = \mathsf{zero} \vee \exists y. x = \mathsf{succ}(y)$
  - $\exists x : nat.\, \alpha \wedge \ldots$
    $\rightsquigarrow (\exists x.\, x = \mathsf{zero} \wedge \alpha \wedge \cdots) \vee (\exists x, y.\, x = \mathsf{succ}(y) \wedge \alpha \wedge \cdots)$
- for sorts with finitely many (in)finite inhabitants:
  - if $x : nat$ then $\mathrm{fin}(x) \vee x = \mathsf{succ}(x)$

The basic idea

Manipulate normal forms ϕ (DJELLOUL, ET AL 2008):

$$\phi \equiv \exists \bar{x}.\ \alpha\ \wedge \bigwedge_{i=1}^{n} \neg\ \phi_i$$

"simple conjunction" nested normal form

Perform case analysis:

- for sorts with finitely many constructors:
  - if $x : nat$ then $x =$ zero $\vee\ \exists y.x =$ succ(y)
  - $\exists x : nat.\, \alpha \wedge \ldots$
    $\rightsquigarrow (\exists x.\, x = \text{zero} \wedge \alpha \wedge \cdots) \vee (\exists x, y.\, x = \text{succ}(y) \wedge \alpha \wedge \cdots)$
- for sorts with finitely many (in)finite inhabitants:
  - if $x : nat$ then $\mathrm{fin}(x) \vee x = \text{succ}(x)$
- but avoid infinite loops:
  $x = \text{succ}(x) \quad \rightsquigarrow \quad \exists y.x = \text{succ}(y) \wedge y = \text{succ}(y) \quad \rightsquigarrow \quad \ldots$