Decision Procedures for Algebraic Data Types with Abstractions

Philippe Suter, Mirco Dotta
and Viktor Kuncak

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Verification of functional programs

proof

counterexample

(input, trace)

```
sealed abstract class Tree
case class Node(left: Tree, value: Int, right: Tree) extends Tree
case class Leaf() extends Tree

object BST {
  def add(tree: Tree, element: Int): Tree = tree match {
    case Leaf() ⇒ Node(Leaf(), element, Leaf())
    case Node(l, v, r) if v > element ⇒ Node(add(l, element), v, r)
    case Node(l, v, r) if v < element ⇒ Node(l, v, add(r, element))
    case Node(l, v, r) if v == element ⇒ tree
  } ensuring (result ≠ Leaf())
}
```

$$(tree = Node(l, v, r) \land v > element \land result \neq Leaf()) \Rightarrow Node(result, v, r) \neq Leaf()$$

We know how to generate verification conditions for functional programs

Proving verification conditions

$$(tree = Node(l, v, r) \land v > element \land result \neq Leaf()) \Rightarrow Node(result, v, r) \neq Leaf()$$

D.C. Oppen, *Reasoning about Recursively Defined Data Structures*, POPL '78

G. Nelson, D.C. Oppen, *Simplification by Cooperating Decision Procedure*, TOPLAS '79

Previous work gives decision procedures that can handle certain verification conditions

```
sealed abstract class Tree
case class Node(left: Tree, value: Int, right: Tree) extends Tree
case class Leaf() extends Tree

object BST {
  def add(tree: Tree, element: Int): Tree = tree match {
    case Leaf() ⇒ Node(Leaf(), element, Leaf())
    case Node(l, v, r) if v > element ⇒ Node(add(l, element), v, r)
    case Node(l, v, r) if v < element ⇒ Node(l, v, add(r, element))
    case Node(l, v, r) if v == element ⇒ tree
  } ensuring (content(result) == content(tree) ∪ { element })

  def content(tree: Tree) : Set[Int] = tree match {
    case Leaf() ⇒ ∅
    case Node(l, v, r) ⇒ content(l) ∪ { v } ∪ content(r)
  }
}
```

Complex verification condition

Set Expressions

$t_1 = \text{Node}(t_2, e_1, t_3)$

$\land\ \text{content}(t_4) = \text{content}(t_2) \cup \{ e_2 \}$

$\land\ \text{content}(\text{Node}(t_4, e_1, t_3)) \neq \text{content}(t_1) \cup \{ e_2 \}$

?

where

```
def content(tree: Tree) : Set[Int] = tree match {
  case Leaf() ⇒ Ø
  case Node(l, v, r) ⇒ content(l) ∪ { v } ∪ content(r)
}
```

Algebraic Data Types

Recursive Function

Our contribution

Decision procedures for extensions of algebraic data types with certain recursive functions

Formulas we aim to prove

Quantifier-free Formula

$t_1 = \text{Node}(t_2, e_1, t_3)$
$\wedge\ \text{content}(t_4) = \text{content}(t_2) \cup \{ e_2 \}$
$\wedge\ \text{content}(\text{Node}(t_4, e_1, t_3)) \neq \text{content}(t_1) \cup \{ e_2 \}$

where

def content(tree: Tree) : Set[Int] = tree match {
 case Leaf() ⇒ ∅
 case Node(l, v, r) ⇒ content(l) ∪ { v } ∪ content(r)
}

Generalized Fold Function

Domain with a Decidable Theory

General form of our recursive functions

empty : *C*
combine : *(C, E, C)* → *C*

```
def content(tree: Tree) : Set[Int] = tree match {
  case Leaf() ⇒ ∅
  case Node(l, v, r) ⇒ content(l) ∪ { v } ∪ content(r)
}
```

Scope of our result - Examples

Tree content abstraction, as a:

- Set [Kuncak,Rinard'07]
- Multiset [Piskac,Kuncak'08]
- List [Plandowski'04]

Tree size, height, min [Papadimitriou'81]

Invariants (sortedness,…) [Nelson,Oppen'79]

How do we prove such formulas?

Quantifier-free Formula

$t_1 = \text{Node}(t_2, e_1, t_3)$
$\land\ \text{content}(t_4) = \text{content}(t_2) \cup \{ e_2 \}$
$\land\ \text{content}(\text{Node}(t_4, e_1, t_3)) \neq \text{content}(t_1) \cup \{ e_2 \}$

where

def content(tree: Tree) : Set[Int] = tree match {
  case Leaf() ⇒ ∅
  case Node(l, v, r) ⇒ content(l) ∪ { v } ∪ content(r)
}

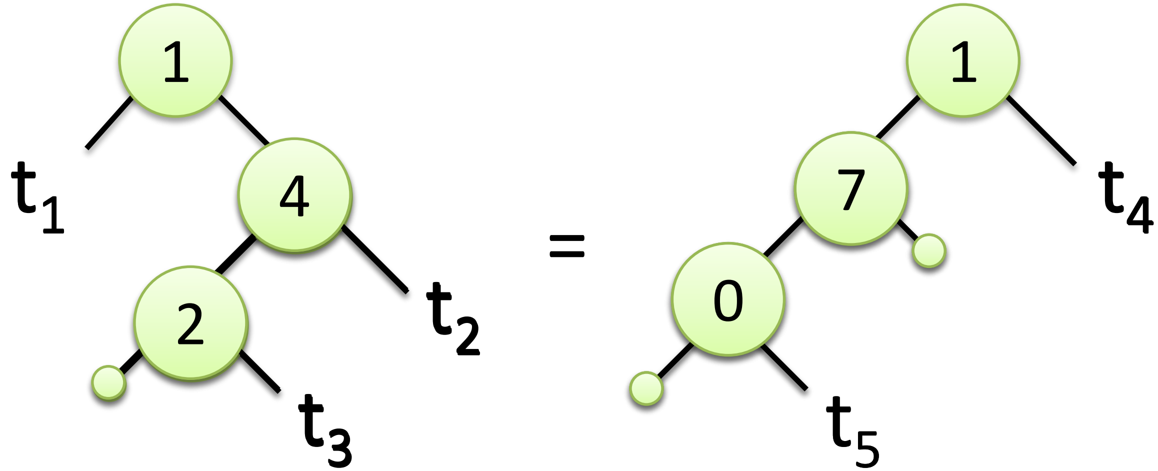
Generalized Fold Function

Domain with a Decidable Theory

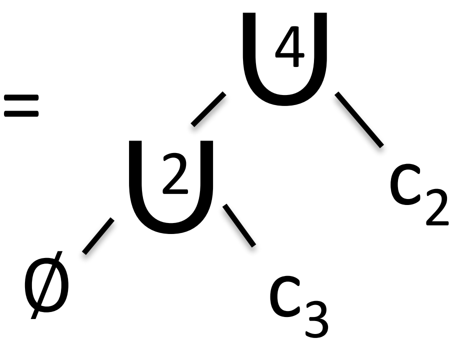
Separate the Conjuncts

$$t_1 = \mathrm{Node}(t_2, e_1, t_3)$$
$$\land\ \mathrm{content}(t_4) = \mathrm{content}(t_2) \cup \{ e_2 \}$$
$$\land\ \mathrm{content}(\mathrm{Node}(t_4, e_1, t_3)) \neq \mathrm{content}(t_1) \cup \{ e_2 \}$$

$$t_1 = \mathrm{Node}(t_2, e_1, t_3) \land t_5 = \mathrm{Node}(t_4, e_1, t_3) \land$$
$$c_4 = c_2 \cup \{ e_2 \} \land c_5 \neq c_1 \cup \{ e_2 \} \land$$
$$c_1 = \mathrm{content}(t_1) \land \ldots \land c_5 = \mathrm{content}(t_5)$$



$$= \qquad \xrightarrow{\text{content}} c_4 =$$



$$= $$

$$c_4 = \{\, 4 \,\} \cup \{\, 2 \,\} \cup \emptyset \cup c_3 \cup c_2$$

Overview of the decision procedure

tree constraints from the input formula

$$t_1 = \mathrm{Node}(t_2, e_1, t_3) \wedge t_5 = \mathrm{Node}(t_4, e_1, t_3)$$

set constraints from the input formula

$$c_4 = c_2 \cup \{ e_2 \} \wedge c_5 \neq c_1 \cup \{ e_2 \}$$

mappings from the input formula

$$c_i = \mathrm{content}(t_i),\ i \in \{ 1, \ldots, 5 \}$$

$$c_1 = c_2 \cup \{ e_1 \} \cup c_3 \wedge c_5 = c_4 \cup \{ e_1 \} \cup c_3$$

additional derived constraints

$$c_4 = c_2 \cup \{ e_2 \} \wedge c_5 \neq c_1 \cup \{ e_2 \} \wedge c_1 = c_2 \cup \{ e_1 \} \cup c_3 \wedge c_5 = c_4 \cup \{ e_1 \} \cup c_3$$

resulting formula

Decision Procedure for Sets

The resulting formula is in the decidable theory of sets

What we have seen is a simple correct algorithm

But is it complete?

A verifier based on such procedure

```
val c1 = content(t1)
val c2 = content(t2)
if (t1 ≠ t2) {
 if (c1 == Ø) {
   assert(c2 ≠ Ø)
   x = c2.chooseElement
 }
}
```

Warning: possible assertion violation

$$c_1 = \mathrm{content}(t_1) \land c_2 = \mathrm{content}(t_2) \land t_1 \neq t_2 \land c_1 = \emptyset \land c_2 = \emptyset$$

Source of incompleteness

$$c_1 = \mathrm{content}(t_1) \wedge c_2 = \mathrm{content}(t_2) \wedge t_1 \neq t_2 \wedge c_1 = \emptyset \wedge c_2 = \emptyset$$

Models for the formula in the logic of sets must not contradict the disequalities over trees

How to make the algorithm complete

- Case analysis for each tree variable:
  - is it Leaf ?
  - Is it not Leaf ?

$c_1 = \text{content}(t_1) \wedge c_2 = \text{content}(t_2) \wedge t_1 \neq t_2 \wedge c_1 = \emptyset \wedge c_2 = \emptyset$

~~$\wedge\ t_1 = \text{Leaf} \wedge t_2 = \text{Node}(t_3, e, t_4)$~~

~~$\wedge\ t_1 = \text{Leaf} \wedge t_2 = \text{Leaf}$~~

~~$\wedge\ t_1 = \text{Node}(t_3, e_1, t_4) \wedge t_2 = \text{Node}(t_5, e_2, t_6)$~~

~~$\wedge\ t_1\ \text{Node}(t_3, e, t_4) \wedge t_2 = \text{Leaf}$~~

This gives a complete decision procedure for the content function that maps to sets

What about other content functions?

Tree content abstraction, as a:

- Set
- Multiset
- List

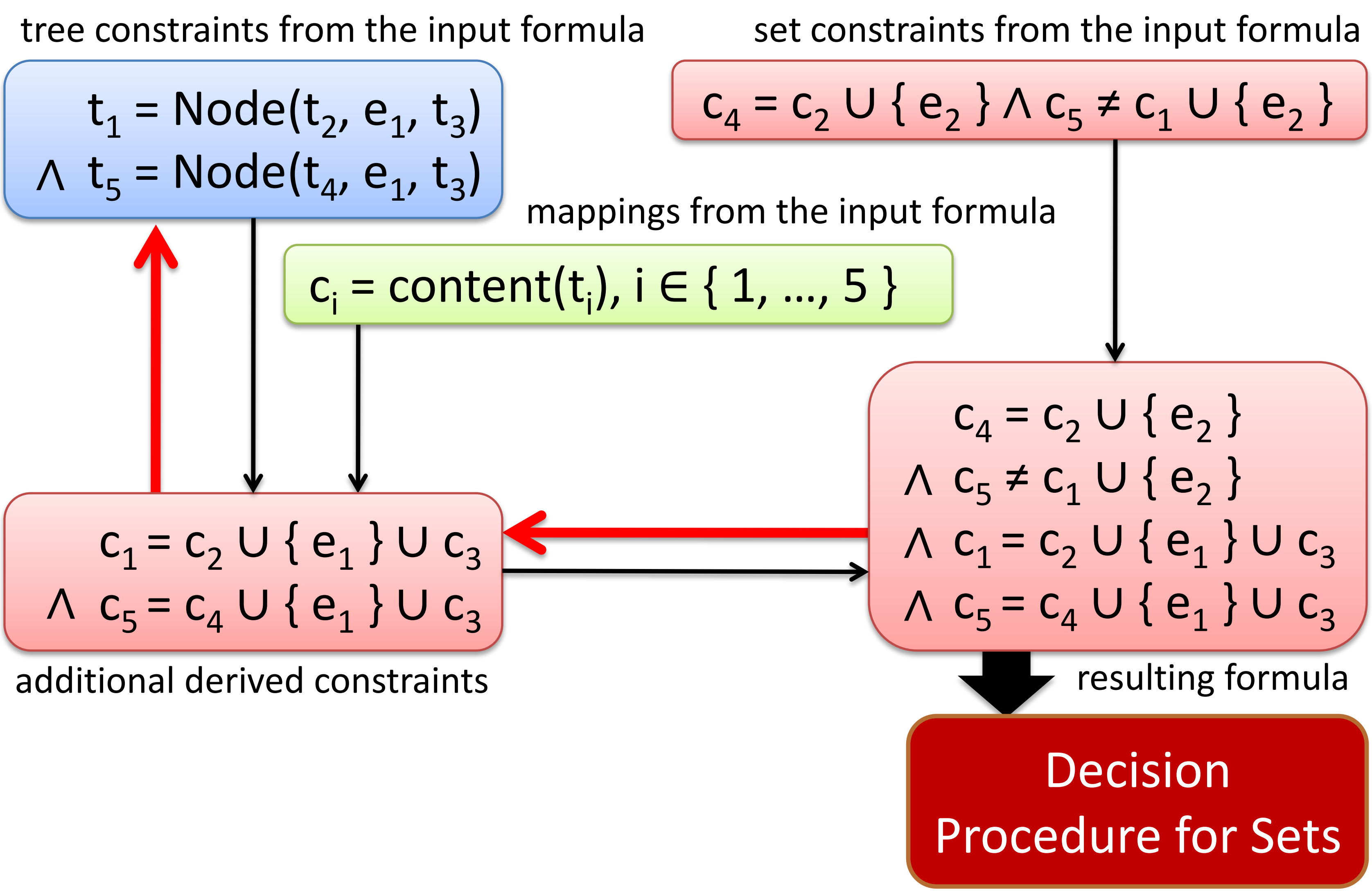
Tree size, height, min

Invariants (sortedness,…)

Sufficient Surjectivity

How and when we can have a complete algorithm

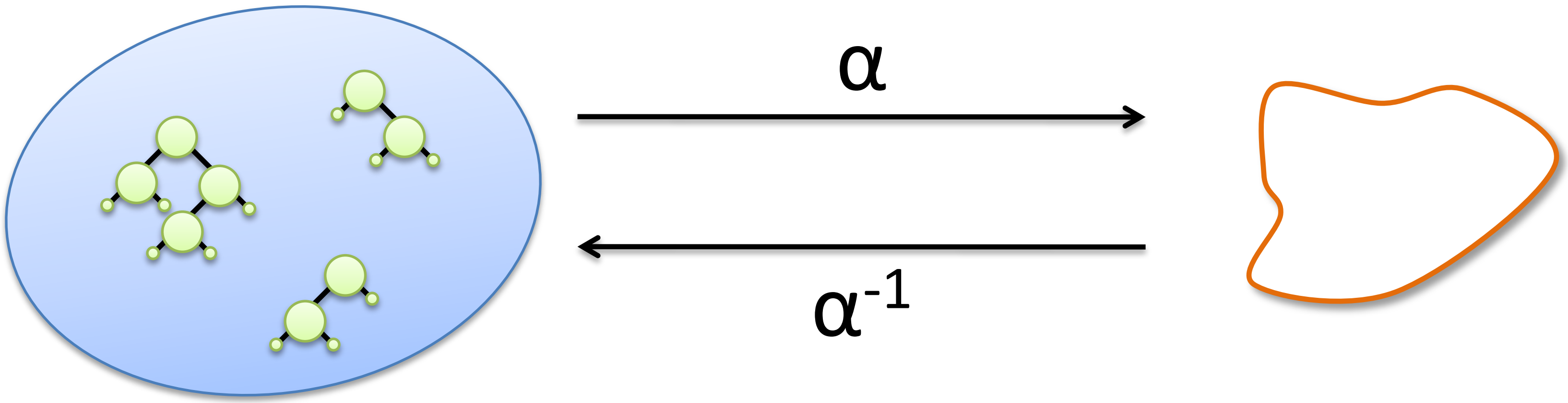
Choice of trees is constrained by sets



Inverse images

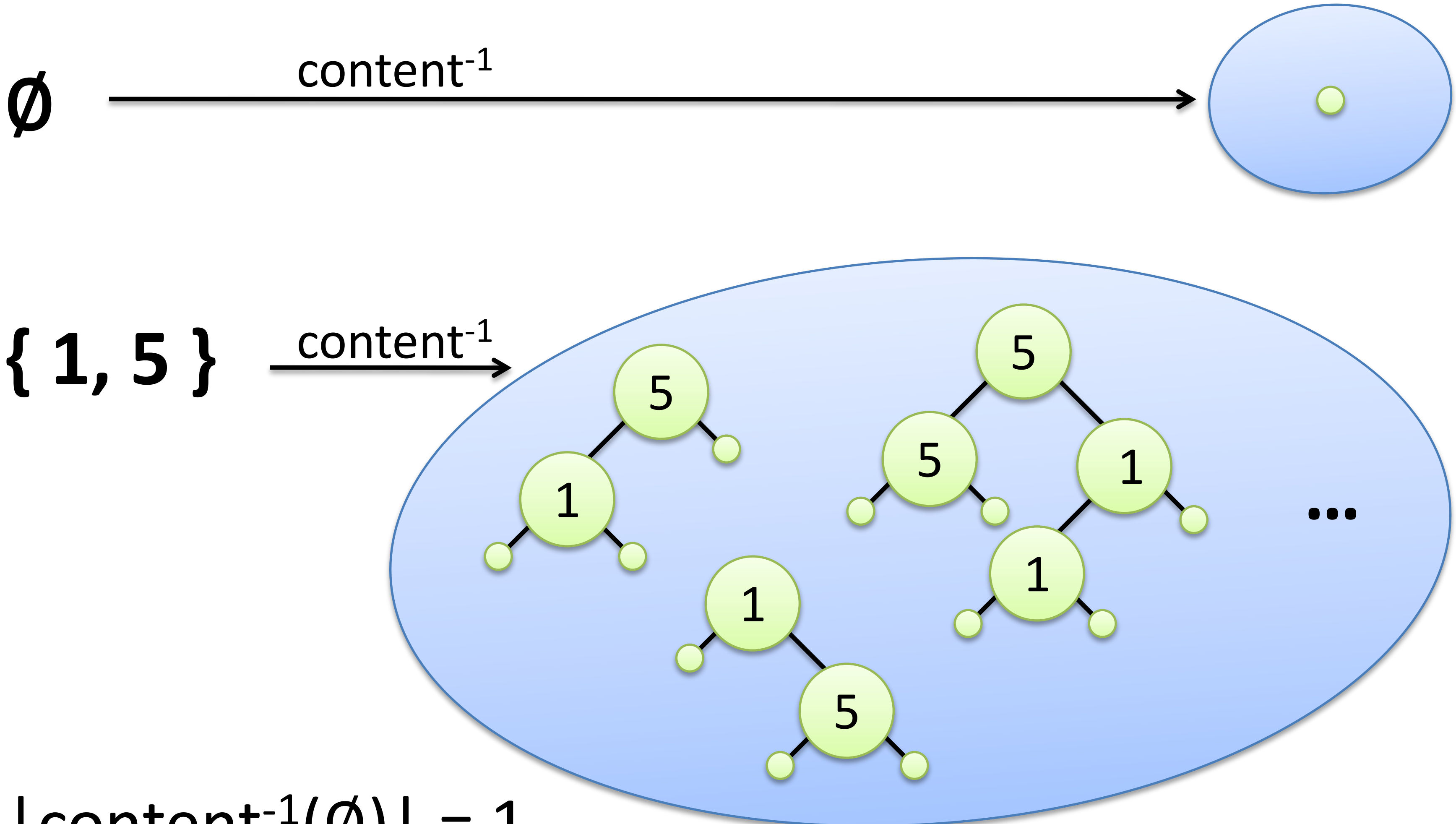
- When we have a model for c_1, c_2, … how can we pick distinct values for t_1, t_2,… ?

$$t_i \in \text{content}^{-1}(c_i) \iff c_i = \text{content}(t_i)$$



The cardinality of $\alpha^{-1}(c_i)$ is what matters.

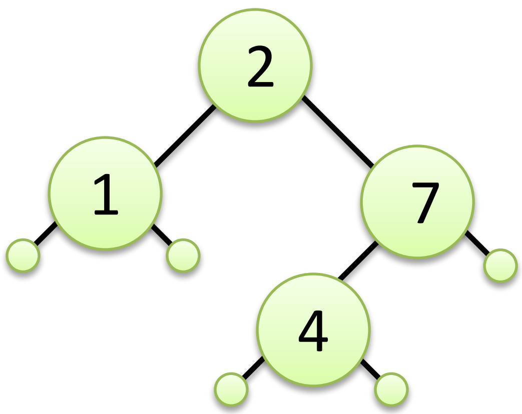
'Surjectivity' of set abstraction



$|\text{content}^{-1}(\emptyset)| = 1$

$|\text{content}^{-1}(\{1, 5\})| = \infty$

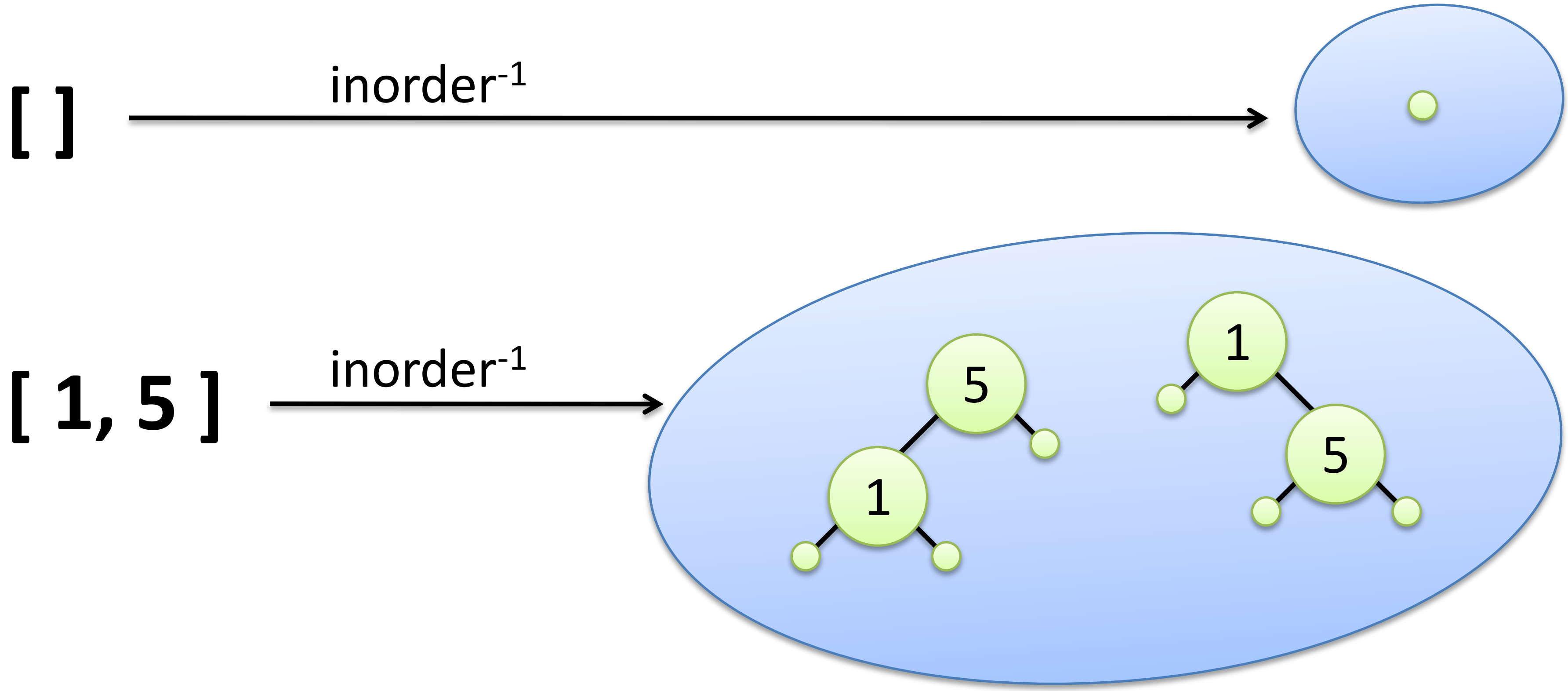
In-order traversal



inorder^{-}

[1, 2, 4, 7]

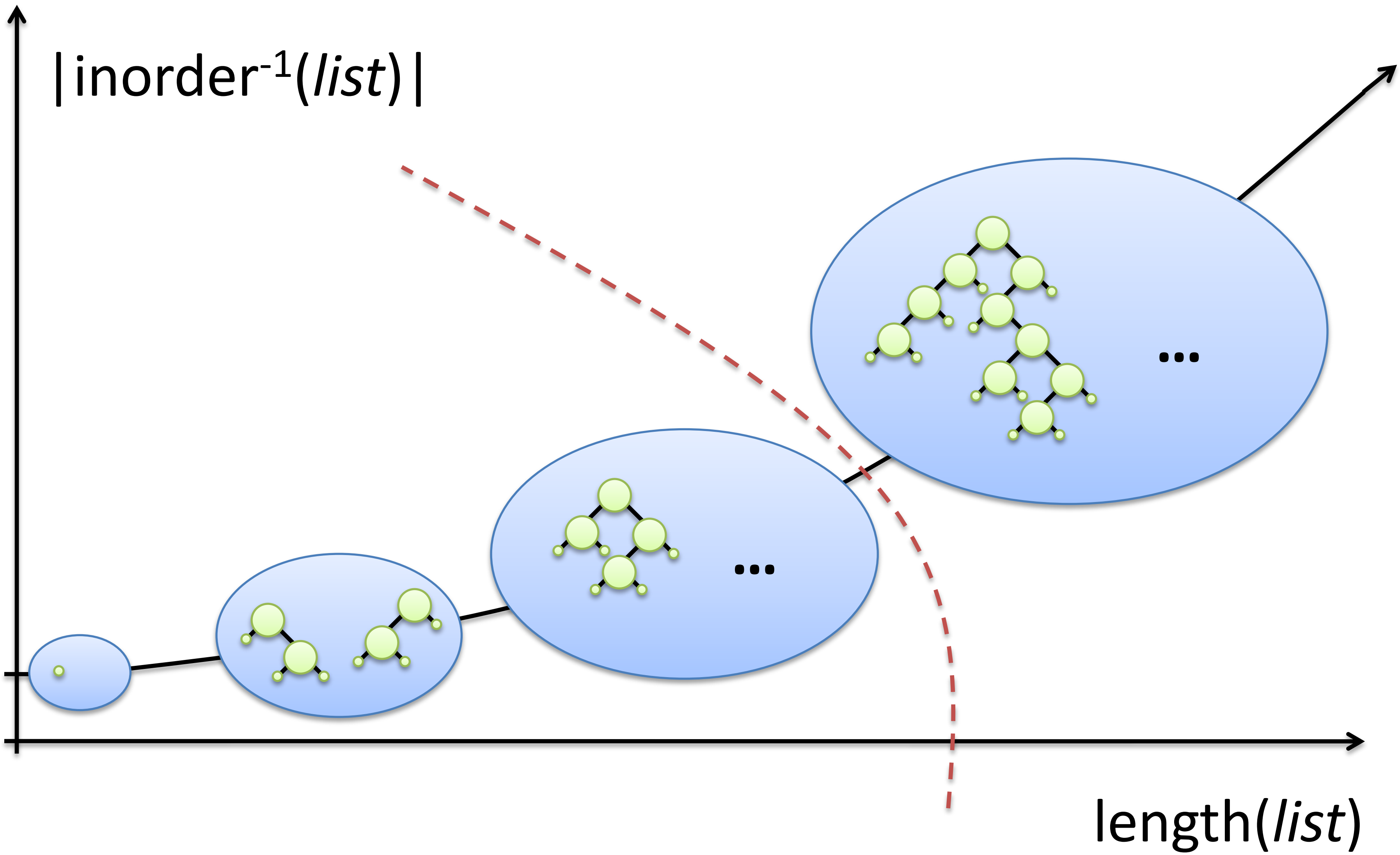
‘Surjectivity’ of in-order traversal



$$|\text{inorder}^{-1}(\mathit{list})| = \frac{(2n)!}{(n+1)!\,n!}$$

(number of trees of size n = length(*list*))

More trees map to longer lists



An abstraction function α (e.g. content, inorder) is *sufficiently surjective* if and only if, for each number $p > 0$, there exist, computable as a function of p:

- a finite set of shapes S_p
- a closed formula M_p in the collection theory such that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$

such that, for every term t, $M_p(\alpha(t))$ or $\check{s}(t)$ in S_p.

Pick p sufficiently large.

Guess which trees have a problematic shape.

Guess their shape and their elements.

By construction values for all other trees can be found.

Generalization of the Independence of Disequations Lemma

For a conjunction of *n* disequalities over tree terms, if for each term we can pick a value from a set of trees of size at least *n+1*, then we can pick values that satisfy all disequalities.

We can make sure there will be sufficiently many trees to choose from.

Sufficiently surjectivity holds in practice

Theorem:
For every sufficiently surjective abstraction our procedure is complete.

Theorem:
The following abstractions are sufficiently surjective:
set content, multiset content, list (any-order), tree height, tree size, minimum, sortedness

A complete decision procedure for all these cases!

Related Work

G. Nelson, D.C. Oppen, *Simplification by Cooperating Decision Procedure*, TOPLAS ’79

V. Sofronie-Stokkermans, *Locality Results for Certain Extensions of Theories with Bridging Functions*, CADE ’09

Some implemented systems:
ACL2, Isabelle, Coq, Verifun, Liquid Types

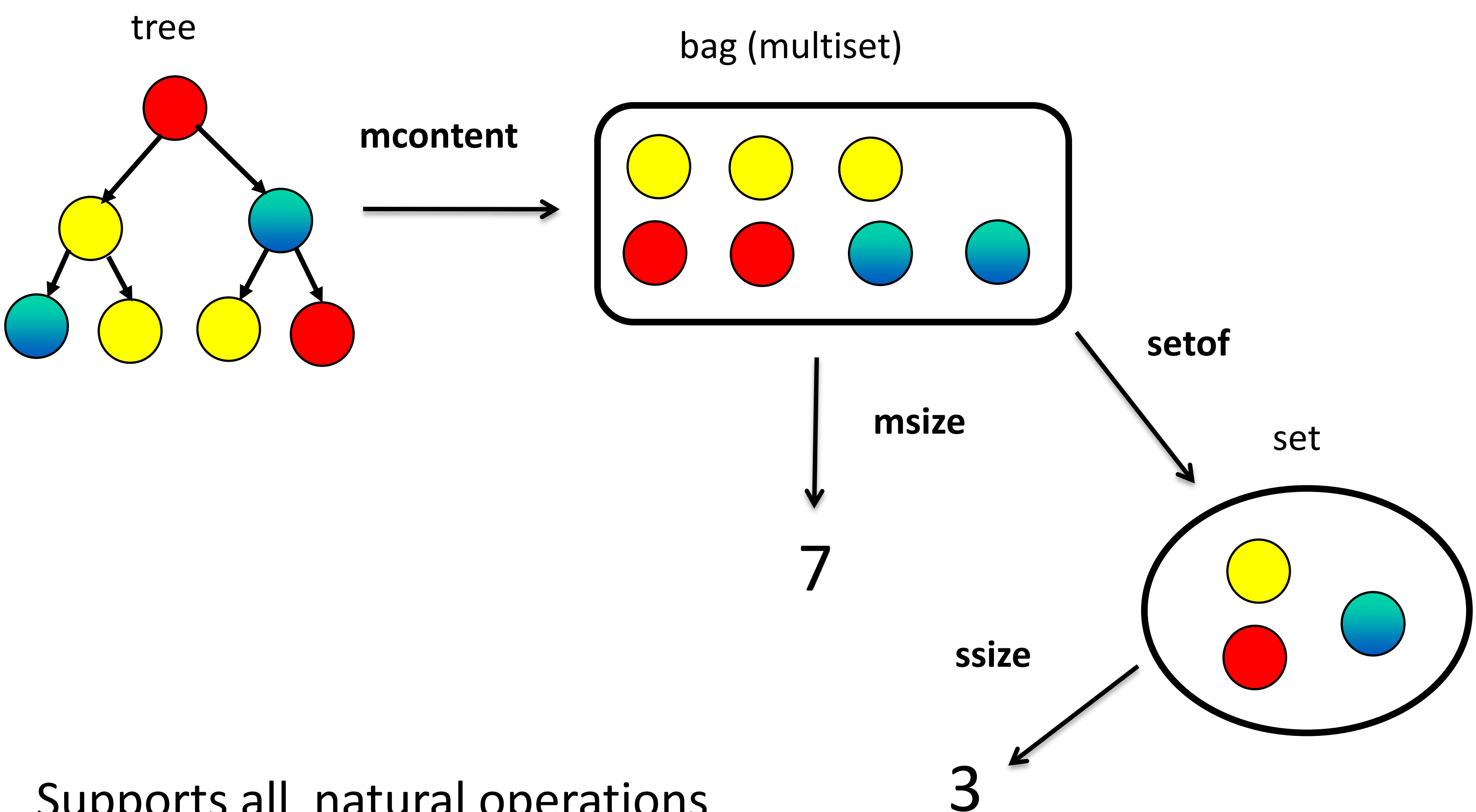
Decision Procedures for Algebraic Data Types with Abstractions

- Reasoning about functional programs reduces to proving formulas
- Decision procedures always find a proof or a counterexample
- Previous decision procedures handle recursion-free formulas
- We introduced decision procedures for formulas with recursive fold functions

Thank you !

Extra Slides

Decision procedure for data structure hierarchy



Supports all natural operations
on trees, multisets, sets, and homomorphisms between them

When we are not complete

- When α^{-1} does not grow
- The only natural example we found so far: when there is no abstraction!
  - Map trees into trees by mirroring them or
  - Reversing the list

Sortedness

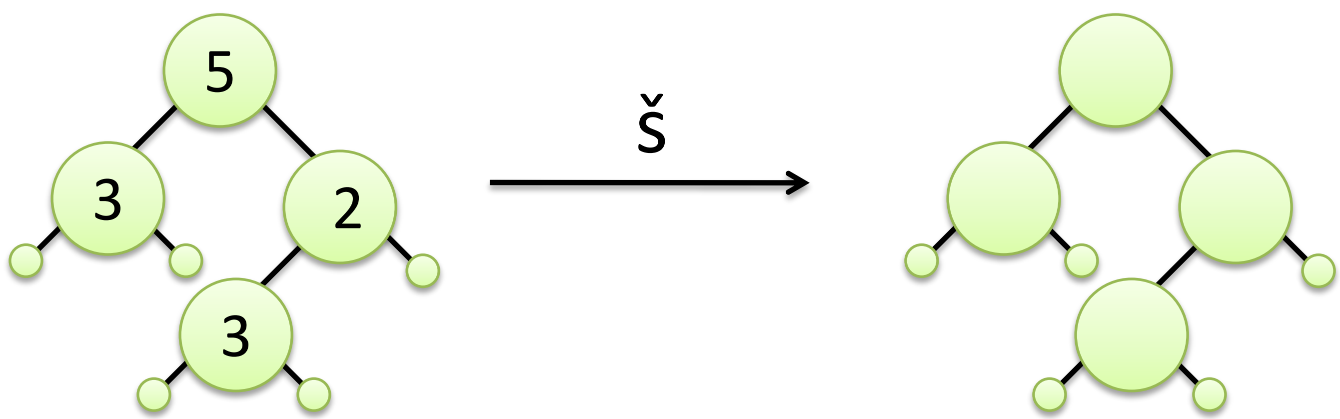
End of extra slides

Stop clicking

An abstraction function α is *sufficiently surjective* if and only if, for each number $p > 0$, there exist, computable as a function of p:

- a finite set of shapes S_p
- a closed formula M_p in the collection theory such that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$

such that, for every term t, $M_p(\alpha(t))$ or $\check{s}(t)$ in S_p.



An abstraction function α is *sufficiently surjective* if and only if, for each number $p > 0$, there exist, computable as a function of p:

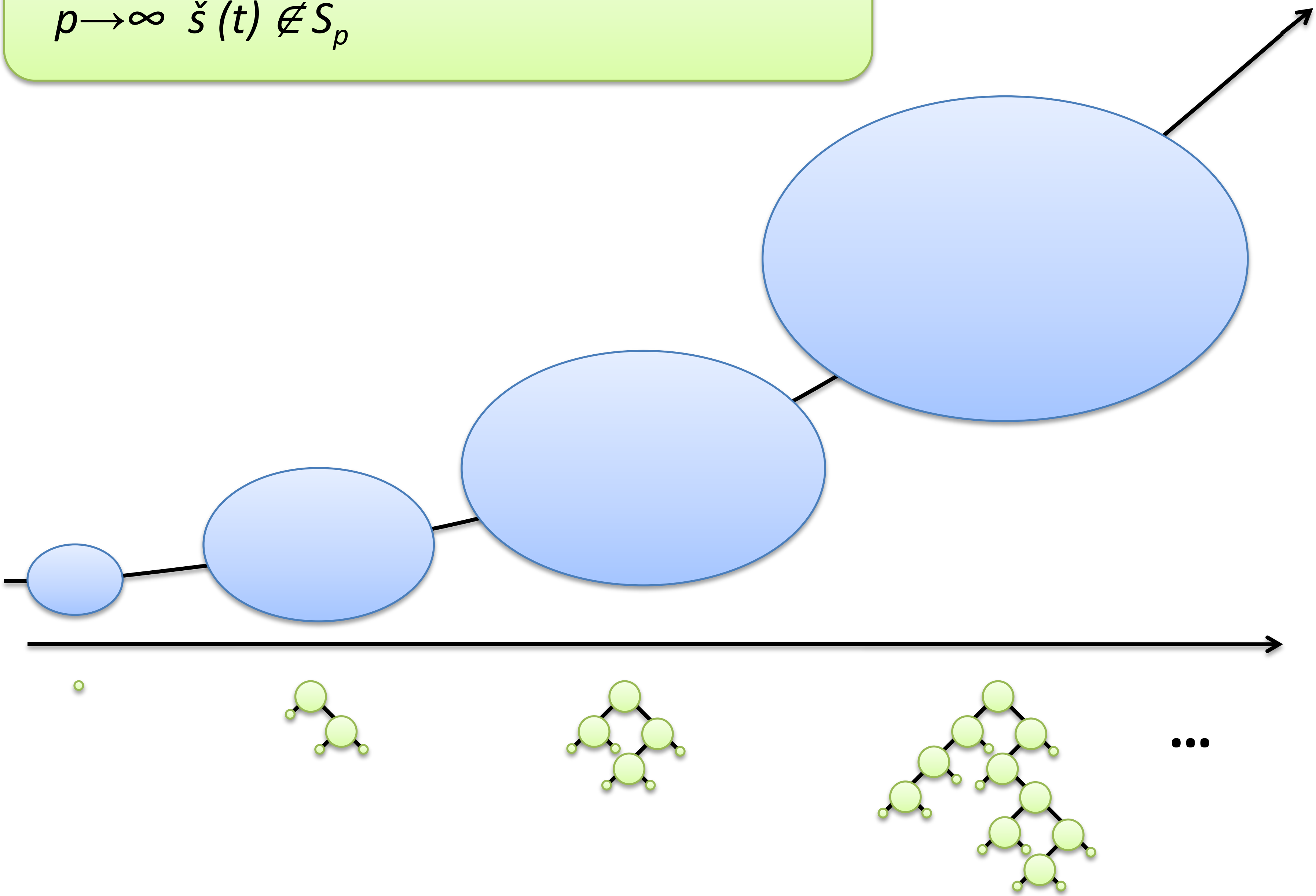
- a finite set of shapes S_p
- a closed formula M_p in the collection theory such that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$

such that, for every term t, $M_p(\alpha(t))$ or $\check{s}(t)$ in S_p.

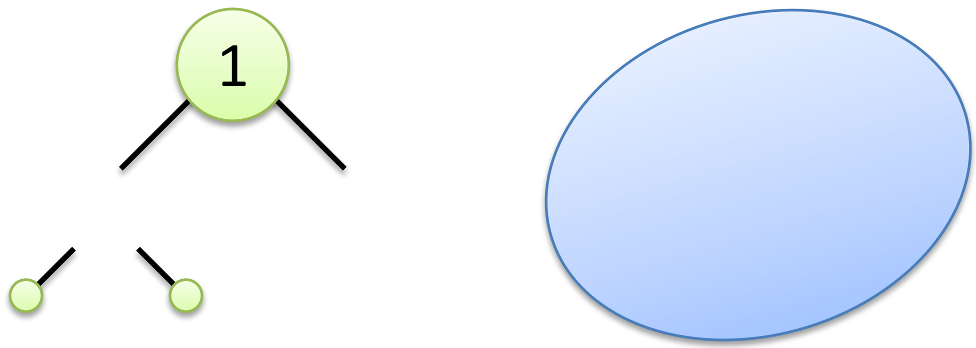
This definition implies:

$$\lim_{p \to \infty} \inf_{\check{s}(t) \notin S_p} |\alpha^{-1}(\alpha(t))| = \infty$$

$$\lim_{p \to \infty} \inf_{\check{s}(t) \notin S_p} |\alpha^{-1}(\alpha(t))| = \infty$$

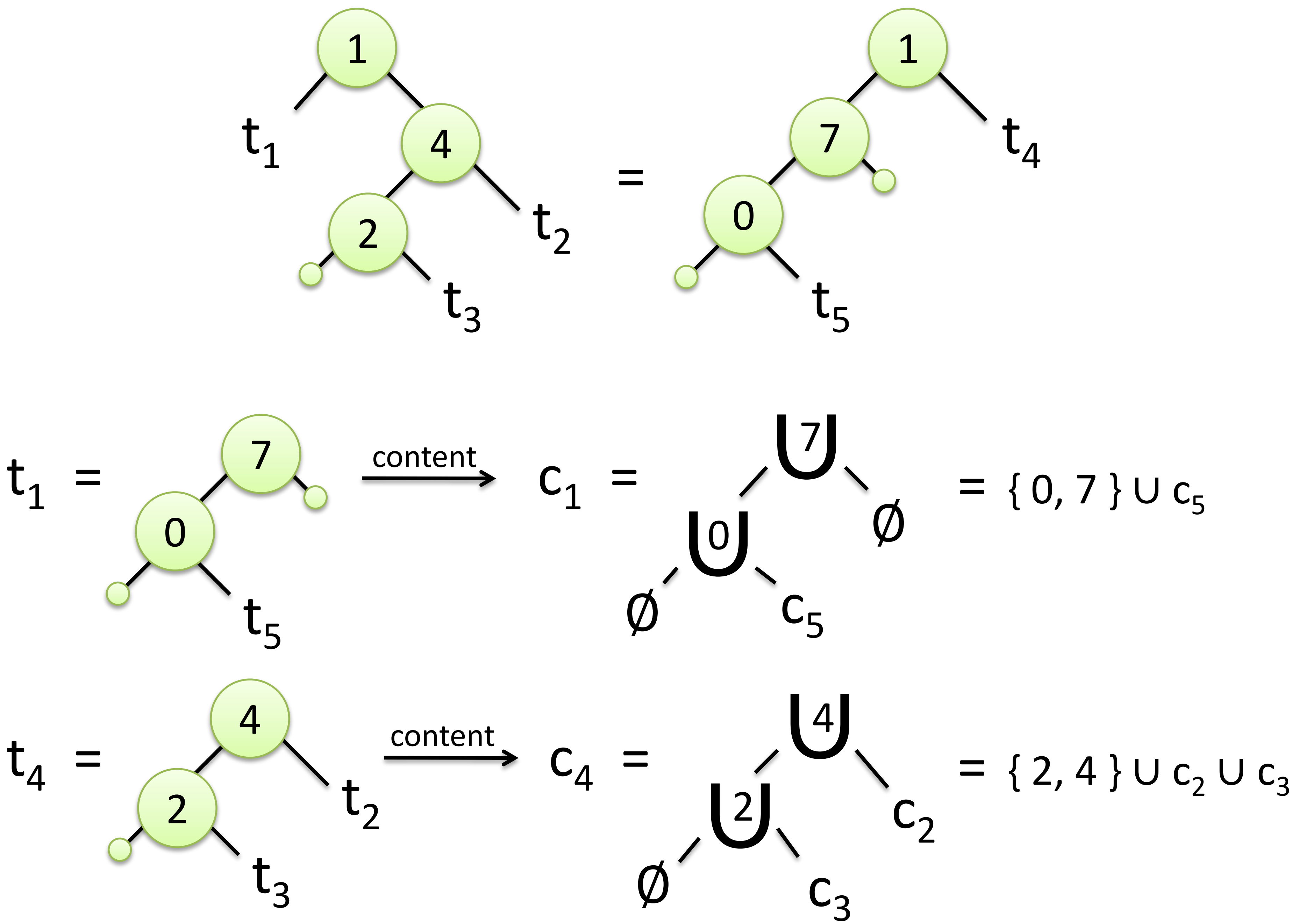


To copy-paste

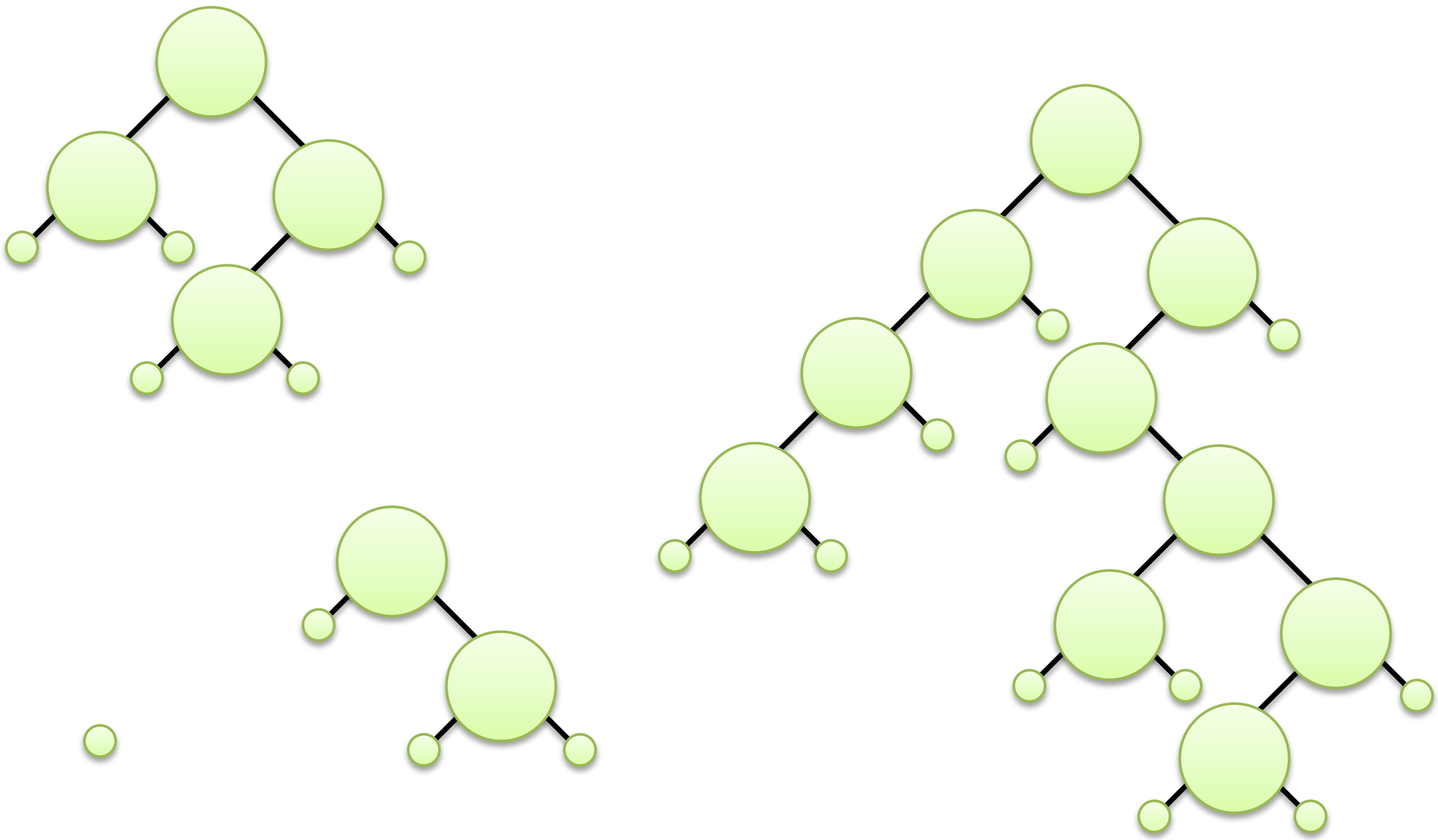


Wc_1W ∧ ∨ ∪ ≠ ⊢ ∈ ∉ ⇒ → α Wα^{-1}W

š ⇔ Ø α


1
4
2
t1
t2
t3
=
1
7
0
t4
t5
t1 =
7
0
t5
content
c1 =
7
0
∅
∅
c5
= { 0, 7 } ∪ c5
t4 =
4
2
t2
t3
content
c4 =
4
2
∅
c2
c3
= { 2, 4 } ∪ c2 ∪ c3

Trees Trees Trees



Overview of the Decision Procedure

$t_1 = \mathrm{Node}(t_2, e_1, t_3) \wedge t_5 = \mathrm{Node}(t_4, e_1, t_3)$

$\wedge\ t_1 \neq t_2 \wedge t_1 \neq t_3 \wedge \ldots \wedge e_1 = e_2$

unification

$t_1 = \mathrm{Node}(t_2, e_1, t_3)$

$\wedge\ t_5 = \mathrm{Node}(t_4, e_1, t_3)$

$c_1 = \mathrm{content}(t_1)$

$= \mathrm{content}(\mathrm{Node}(t_2, e_1, t_3))$

$= \mathrm{content}(t_2) \cup \{ e_1 \} \cup \mathrm{content}(t_3)$

$= c_2 \cup \{ e_1 \} \cup c_3$

```
def content(tree: Tree) : Set[Int] = tree match {
  case Leaf() ⇒ Ø
  case Node(l, v, r) ⇒ content(l) ∪ { v } ∪ content(r)
}
```

$c_i = \mathrm{content}(t_i),\ i \in \{ 1, \ldots, 5 \}$

Ghost Variables?

```
object BST {
  def contains(tree: Tree, element: Int): Tree = tree match {
    case Leaf() => false
    case Node(l, v, r) if v > element => contains(l, element)
    case Node(l, v, r) if v < element => contains(r, element)
    case Node(l, v, r) if v == element => true
  } ensuring (result <=> element ∈ tree.content)
}
```

Requires stating and proving an invariant such as:

$\forall$ (l : Leaf) .
  l.content = $\emptyset$
$\forall$ (n : Node) .
  n.content = n.left.content $\cup$ { n.element } $\cup$ n.right.content

```
sealed abstract class Tree { val content: Set[Int] }
case class Node(content: Set[Int], left: Tree, value: Int, right:
Tree) extends Tree
case class Leaf() extends Tree { val content = ∅ }

object BST {
  def add(tree: Tree, element: Int): Tree = tree match {
    case Leaf() => Node({ element }, Leaf(), element, Leaf())
    case Node(l, v, r) if v > element =>
      Node(tree.content ∪ { element }, add(l, element), v, r)
    case Node(l, v, r) if v < element =>
      Node(tree.content ∪ { element }, l, v, add(r, element))
    case Node(l, v, r) if v == element => tree
  } ensuring (result.content == tree.content ∪ { element })
}
```

- Essentially duplicates the code

Our Approach: No Ghosts!

- In a functional setting, specification variables are just another view on the same data
- Idea: provide the view explicitly, in the PL

Completeness

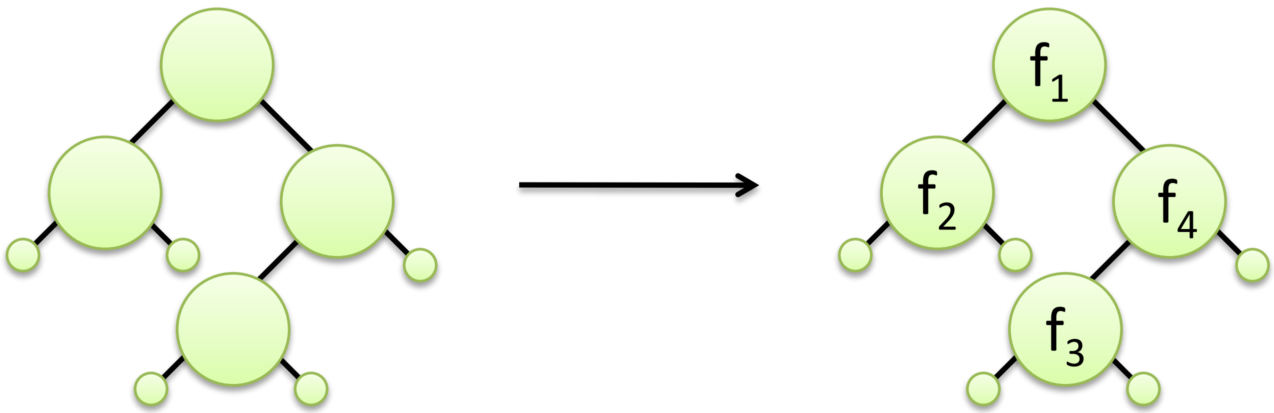
In general, we need a way to encode:

$$t_i \neq t_j \wedge t_k \neq t_l \wedge \ldots$$
$$\wedge\ c_i = \alpha(t_i) \wedge c_j = \alpha(t_j) \wedge \ldots$$

in the domain theory.

Sufficient Surjectivity

- For each tree *t* in the formula, guess its shape in S_p, or write $M_p(t)$
- Populate the shapes with fresh variables



- Trees with different shapes are different by construction.
- For the other ones, create a disjunction of disequalities over their elements

Sufficient Surjectivity

- All the trees such that $M_p(t)$ can be made distinct and still map to the same collection

Independence of Disequations Lemma:

For a conjunction of *n* disequalities of tree terms, if for each term we can pick a value from a set of trees of size at least *n*, then we can pick values that satisfy all disequalities.

Sufficient Surjectivity