Decision Procedures for Algebraic Data Types with Abstractions

Philippe Suter, Mirco Dotta
and Viktor Kuncak
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) ∧ v > element ∧ result ≠ Leaf())
  ⇒ Node(result, v, r) ≠ Leaf()
Proving verification conditions

\[(\text{tree} = \text{Node}(l, v, r) \land v > \text{element} \land \text{result} \neq \text{Leaf}()) \Rightarrow \text{Node}(\text{result}, v, r) \neq \text{Leaf}())\]

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


*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) U { element })
}

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

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

where

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

Decision procedures for extensions of algebraic data types with certain recursive functions
Formulas we aim to prove

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

where

\[
\text{def} \; \text{content}(\text{tree}: \text{Tree}) : \text{Set}[\text{Int}] = \text{tree match \{ 
\text{case} \; \text{Leaf()} \Rightarrow \emptyset \\
\text{case} \; \text{Node}(l, v, r) \Rightarrow \text{content}(l) \cup \{ v \} \cup \text{content}(r) 
\}}
\]
def α(tree: Tree):
    C = tree match {
        case Leaf() ⇒ empty
        case Node(l, v, r) ⇒ combine(α(l), v, α(r))
    }

General form of our recursive functions

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

Generalized Fold Function

Domain with a Decidable Theory
Separate the Conjuncts

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

\[ t_1 = \text{Node}(t_2, e_1, t_3) \land t_5 = \text{Node}(t_4, e_1, t_3) \land \]

\[ c_4 = c_2 \cup \{ e_2 \} \land c_5 \neq c_1 \cup \{ e_2 \} \]

\[ c_1 = \text{content}(t_1) \land ... \land c_5 = \text{content}(t_5) \]
\[
\begin{align*}
\text{content} & \quad \rightarrow \quad c_4 = \bigcup & \quad \text{content} & \quad \rightarrow \quad c_4 = \bigcup \\
& = {4} \cup {2} \cup \emptyset \cup c_3 \cup c_2 & = {4} \cup {2} \cup \emptyset \cup c_3 \cup c_2 \\
\end{align*}
\]
Overview of the decision procedure

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 == ∅) {
        assert(c2 != ∅)
        x = c2.chooseElement
    }
}

Warning: possible assertion violation

\[ c_1 = \text{content}(t_1) \land c_2 = \text{content}(t_2) \land t_1 \neq t_2 \land c_1 = \emptyset \land c_2 = \emptyset \]
Source of incompleteness

\[ c_1 = \text{content}(t_1) \land c_2 = \text{content}(t_2) \land t_1 \neq t_2 \land c_1 = \emptyset \land c_2 = \emptyset \]

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?

\[ c_1 = \text{content}(t_1) \land c_2 = \text{content}(t_2) \land t_1 \neq t_2 \land c_1 = \emptyset \land c_2 = \emptyset \]

\[ \land t_1 \text{ Leaf} \land t_2 \text{ Node}(t_3, e, t_4) \]

\[ \land t_1 \text{ Leaf} \land t_2 \text{ Leaf} \]

\[ \land t_1 \text{ Node}(t_3, e_1, t_4) \land t_2 = \text{Node}(t_5, e_2, t_6) \]

\[ \land t_1 \text{ Node}(t_3, e, t_4) \land t_2 = \text{Leaf} \]

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

t_1 = \text{Node}(t_2, e_1, t_3)
\land t_5 = \text{Node}(t_4, e_1, t_3)

\begin{align*}
c_1 &= c_2 \cup \{e_1\} \cup c_3 \\
\land c_5 &= c_4 \cup \{e_1\} \cup c_3
\end{align*}

\begin{align*}
c_4 &= c_2 \cup \{e_2\} \land c_5 \neq c_1 \cup \{e_2\}
\end{align*}

\begin{align*}
c_4 &= c_2 \cup \{e_2\} \\
\land c_5 &= c_4 \cup \{e_2\}
\end{align*}

\begin{align*}
c_1 &= c_2 \cup \{e_1\} \cup c_3 \\
\land c_5 &= c_4 \cup \{e_1\} \cup c_3
\end{align*}

\begin{align*}
c_4 &= c_2 \cup \{e_2\} \\
\land c_5 &= c_4 \cup \{e_1\} \cup c_3
\end{align*}

Decision Procedure for Sets
Inverse images

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

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

The cardinality of $\alpha^{-1}(c_i)$ is what matters.
‘Surjectivity’ of set abstraction

\[ \emptyset \xrightarrow{\text{content}^{-1}} \ldots \]

\[ \{1, 5\} \xrightarrow{\text{content}^{-1}} \]

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

\[ |\text{content}^{-1}({1, 5})| = \infty \]
In-order traversal

[ 1, 2, 4, 7 ]
‘Surjectivity’ of in-order traversal

\[ |\text{inorder}^{-1}(\text{list})| = \frac{(2n)!}{(n + 1)!n!} \]

(number of trees of size \(n = \text{length}(\text{list})\))
More trees map to longer lists

$$|\text{inorder}^{-1}(\text{list})|$$

length(\text{list})
An abstraction function $\alpha$ (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 $\hat{s}(t)$ 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


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

Some implemented systems:
ACL2, Isabelle, Coq, Verifun, Liquid Types
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 $\alpha^{-1}$ 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 $\alpha$ 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 $\dot{s}(t)$ in $S_p$. 
An abstraction function $\alpha$ 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:

$$\lim_{p \to \infty} \inf \ |\alpha^{-1}(\alpha(t))| = \infty$$

$$p \to \infty \ \check{s}(t) \notin S_p$$
\[
\lim_{p \to \infty} \inf_{\hat{s}(t) \notin S_p} |\alpha^{-1}(\alpha(t))| = \infty
\]
To copy-paste

\[ Wc_1W \land \lor \lor \neq \vdash \in \notin \Rightarrow \rightarrow \alpha \ W\alpha^{-1}W \]
\[ \dot{s} \iff \emptyset \alpha \]
$t_1 = 7$

$\text{content}$

$c_1 = \emptyset \cup \{ 0, 7 \} \cup c_5$

$\emptyset \cup \emptyset = \emptyset$

$t_4 = 4$

$\text{content}$

$c_4 = \emptyset \cup 2 \cup c_2 \cup c_3$

$\emptyset \cup \emptyset = \emptyset$
Trees Trees Trees Trees
Overview of the Decision Procedure

\[
t_1 = \text{Node}(t_2, e_1, t_3) \land t_5 = \text{Node}(t_4, e_1, t_3) \\
\land t_1 \neq t_2 \land t_1 \neq t_3 \land \ldots \land e_1 = e_2
\]

unification

\[
t_1 = \text{Node}(t_2, e_1, t_3) \\
\land t_5 = \text{Node}(t_4, e_1, t_3)
\]

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

c_i = content(t_i), i ∈ { 1, ..., 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:

∀ (l : Leaf) .
    l.content = ∅
∀ (n : Node) .
    n.content = n.left.content ∪ { n.element } ∪ 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 = Ø }

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:

\[ t_i \neq t_j \land t_k \neq t_l \land \ldots \]
\[ \land c_i = \alpha(t_i) \land c_j = \alpha(t_j) \land \ldots \]

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

\[ \text{shape} \]