Formale Spezifikation und Verifikation

Wintersemester 2024

Prof. Dr. Gidon Ernst
gidon.ernst@lmu.de

Software and Computational Systems Lab
Ludwig-Maximilians-Universität München, Germany

September 30, 2024

Evaluation der Vorlesung!

- Bitte nehmen Sie an der Umfrage teil
- Gerne konkretes Feedback im Freitext!

Inferenz von Invarianten sowie Vor-/Nachbedingungen

Vor-/Nachbedingungen!

```
method SchorrWaite(root: Node, ghost S: set<Node>)
  requires root ∈ S;
  // S is closed under 'children':
  requires (∀ n • n ∈ S ⟹ n ≠ null ∧ (∀ ch • ch ∈ n.children ⟹ ch = null ∨ ch ∈ S));
  // the graph starts off with nothing marked and nothing being indicated as currently being visited:
  requires (∀ n • n ∈ S ⟹ ¬n.marked ∧ n.childrenVisited = 0);
  modifies S;
  // nodes reachable from 'root' are marked:
  ensures root.marked;
  ensures (∀ n • n ∈ S ∧ n.marked ⟹ (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
  // every marked node was reachable from 'root' in the pre-state:
  ensures (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
  // the structure of the graph has not changed:
  ensures (∀ n • n ∈ S ⟹ n.childrenVisited = old(n.childrenVisited) ∧ n.children = old(n.children));
{
```

Leino: *Dafny: An automatic program verifier for functional correctness*, LPAR 2010

Invarianten!

```
while (true)
  invariant root.marked ∧ t ≠ null ∧ t ∈ S ∧ t ∉ stackNodes;
  invariant |stackNodes| = 0 ⟺ p = null;
  invariant 0 < |stackNodes| ⟹ p = stackNodes[|stackNodes|-1];
  // stackNodes has no duplicates:
  invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |stackNodes| ⟹ stackNodes[i] ≠ stackNodes[j]);
  invariant (∀ n • n ∈ stackNodes ⟹ n ∈ S);
  invariant (∀ n • n ∈ stackNodes ∨ n = t ⟹
    n.marked ∧ 0 ≤ n.childrenVisited ∧ n.childrenVisited ≤ |n.children| ∧
    (∀ j • 0 ≤ j ∧ j < n.childrenVisited ⟹ n.children[j] = null ∨ n.children[j].marked));
  invariant (∀ n • n ∈ stackNodes ⟹ n.childrenVisited < |n.children|);
  invariant (∀ n • n ∈ S ∧ n.marked ∧ n ∉ stackNodes ∧ n ≠ t ⟹
    (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
  invariant (∀ n • n ∈ S ∧ n ∉ stackNodes ∧ n ≠ t ⟹
    n.childrenVisited = old(n.childrenVisited));
  invariant (∀ n • n ∈ S ⟹ n ∈ stackNodes ∨ n.children = old(n.children));
  invariant (∀ n • n ∈ stackNodes ⟹
    |n.children| = old(|n.children|) ∧
    (∀ j • 0 ≤ j ∧ j < |n.children| ⟹ j = n.childrenVisited ∨ n.children[j] = old(n.children[j])));
  // every marked node is reachable:
  invariant old(ReachableVia(root, path, t, S));
  invariant (∀ n, pth • n ∈ S ∧ n.marked ∧ pth = n.pathFromRoot ⟹ old(ReachableVia(root, pth, n, S)));
  invariant (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
  // the current values of m.children[m.childrenVisited] for m's on the stack:
  invariant 0 < |stackNodes| ⟹ stackNodes[0].children[stackNodes[0].childrenVisited] = null;
  invariant (∀ k • 0 < k ∧ k < |stackNodes| ⟹
    stackNodes[k].children[stackNodes[k].childrenVisited] = stackNodes[k-1]);
  // the original values of m.children[m.childrenVisited] for m's on the stack:
  invariant (∀ k • 0 ≤ k ∧ k+1 < |stackNodes| ⟹
    old(stackNodes[k].children)[stackNodes[k].childrenVisited] = stackNodes[k+1]);
  invariant 0 < |stackNodes| ⟹
    old(stackNodes[|stackNodes|-1].children)[stackNodes[|stackNodes|-1].childrenVisited] = t;
  invariant (∀ n • n ∈ S ∧ ¬n.marked ⟹ n ∈ unmarkedNodes);
  decreases unmarkedNodes, stackNodes, |t.children| - t.childrenVisited;
{
```

Leino: *Dafny: An automatic program verifier for functional correctness*, LPAR 2010

Invarianten! — AAAARGH :|

```
  while (true)
    invariant root.marked ∧ t ≠ null ∧ t ∈ S ∧ t ∉ stackNodes;
    invariant |stackNodes| = 0 ⟺ p = null;
    invariant 0 < |stackNodes| ⟹ p = stackNodes[|stackNodes|-1];
    // stackNodes has no duplicates:
    invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |stackNodes| ⟹ stackNodes[i] ≠ stackNodes[j]);
    invariant (∀ n • n ∈ stackNodes ⟹ n ∈ S);
    invariant (∀ n • n ∈ stackNodes ∨ n = t ⟹
      n.marked ∧ 0 ≤ n.childrenVisited ∧ n.childrenVisited ≤ |n.children| ∧
      (∀ j • 0 ≤ j ∧ j < n.childrenVisited ⟹ n.children[j] = null ∨ n.children[j].marked));
    invariant (∀ n • n ∈ stackNodes ⟹ n.childrenVisited < |n.children|);
    invariant (∀ n • n ∈ S ∧ n.marked ∧ n ∉ stackNodes ∧ n ≠ t ⟹
      (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
    invariant (∀ n • n ∈ S ∧ n ∉ stackNodes ∧ n ≠ t ⟹
      n.childrenVisited = old(n.childrenVisited));
    invariant (∀ n • n ∈ S ⟹ n ∈ stackNodes ∨ n.children = old(n.children));
    invariant (∀ n • n ∈ stackNodes ⟹
      |n.children| = old(|n.children|) ∧
      (∀ j • 0 ≤ j ∧ j < |n.children| ⟹ j = n.childrenVisited ∨ n.children[j] = old(n.children[j])));
    // every marked node is reachable:
    invariant old(ReachableVia(root, path, t, S));
    invariant (∀ n, pth • n ∈ S ∧ n.marked ∧ pth = n.pathFromRoot ⟹ old(ReachableVia(root, pth, n, S)));
    invariant (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
    // the current values of m.children[m.childrenVisited] for m's on the stack:
    invariant 0 < |stackNodes| ⟹ stackNodes[0].children[stackNodes[0].childrenVisited] = null;
    invariant (∀ k • 0 < k ∧ k < |stackNodes| ⟹
      stackNodes[k].children[stackNodes[k].childrenVisited] = stackNodes[k-1]);
    // the original values of m.children[m.childrenVisited] for m's on the stack:
    invariant (∀ k • 0 ≤ k ∧ k+1 < |stackNodes| ⟹
      old(stackNodes[k].children)[stackNodes[k].childrenVisited] = stackNodes[k+1]);
    invariant 0 < |stackNodes| ⟹
      old(stackNodes[|stackNodes|-1].children)[stackNodes[|stackNodes|-1].childrenVisited] = t;
    invariant (∀ n • n ∈ S ∧ ¬n.marked ⟹ n ∈ unmarkedNodes);
    decreases unmarkedNodes, stackNodes, |t.children| - t.childrenVisited;
  {
```

Leino: *Dafny: An automatic program verifier for functional correctness*, LPAR 2010

Erinnerung: Schleifeninvarianten in Hoare-Logik

Beweisregel

$$\frac{P \Rightarrow I \qquad \{I \land \phi\}\ c\ \{I\} \qquad I \land \neg\phi \Rightarrow Q}{\{P\}\ \textbf{while}\ \phi\ \textbf{do}\ c\ \{Q\}}\ \textsc{While}$$

mit drei Prämissen

- Schleifeneintritt: I muss mit P bewiesen werden
- Schleifeniteration: I wird von c aufrecht erhalten
- Schleifenende: I mit $\neg\phi$ etabliert die Nachbedingung Q

Erinnerung: Schleifeninvarianten in Hoare-Logik

Beweisregel

$$\frac{\exists\, I. \qquad P \Rightarrow I \qquad \{I \land \phi\}\; c\; \{I\} \qquad I \land \neg\phi \Rightarrow Q}{\{P\}\ \textbf{while}\ \phi\ \textbf{do}\ c\ \{Q\}}\ \textsc{While}$$

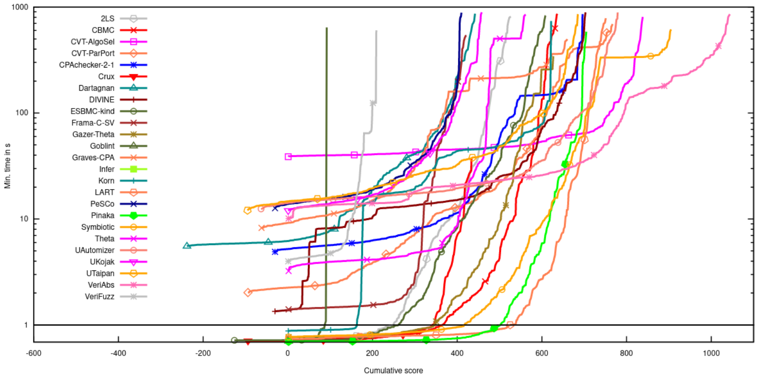
mit drei Prämissen

- Schleifeneintritt: I muss mit P bewiesen werden
- Schleifeniteration: I wird von c aufrecht erhalten
- Schleifenende: I mit $\neg\phi$ etabliert die Nachbedingung Q

Finden von Invarianten ist schwierig:

- I ist eine Prädikatvariable (analog: Kontrakte für rekursive Prozeduren)
- Formeln mit $\exists\, I. \ldots$ nur in Logik *höherer* Stufe ausdrückbar (im Allgemeinen sehr unentscheidbar)

Tools in SV-COMP können routinemäßig Invarianten inferieren!



https://sv-comp.sosy-lab.org/2022/results/results-verified/

Demo:

- KORN (https://github.com/gernst/korn)
- ELDARICA (https://github.com/uuverifiers/eldarica)

Heute: Inferenz von Invarianten und Kontrakten

1. Teil: Horn-Klauseln zur Programmverifikation
   - Geschickte Codierung von Inferenzproblemen:
     Schleifen, Kontrakte, Nebenfläufigkeit, OOP Spezifikationen, ...
   - SMT-LIB gestützte Tools und Solver
   - Literatur: Bjørner, Gurfinkel, McMillan, Rybalchenko:
     Horn clause solvers for program verification, 2015.

2. Teil: Lösungsverfahren (hauptsächlich Invarianten in LIA)
   - numerische abstrakte Domänen
   - Prädikatenabstraktion, syntaxgetriebene Synthese
   - Machine Learning
   - Ausblick: Softwareverifikation (Mastervorlesung im WS, Prof. Beyer)

Hinweis: Diese Lösungsverfahren können natürlich auch verwendet werden, wenn die Programmverifikation nicht mit Horn-Klauseln realisiert ist.

SMT-Solver: Erfüllbarkeit (bisher)

Eine Formel φ ist erfüllbar, wenn es eine gültige Belegung s gibt, sodass $s \models \varphi$

SMT-Solver: Erfüllbarkeit (bisher)

Eine Formel φ ist erfüllbar, wenn es eine gültige Belegung s gibt, sodass $s \models \varphi$

Problemdefinition für den SMT-Solver, z.B. in SMT-LIB

- freie Variablen (= Konstanten) $x_1 : \tau_1, \dots, x_n : \tau_n$
- Bedingungen $\phi_1, \dots, \phi_m$ über $x_1, \dots, x_n$

SMT-Solver: Erfüllbarkeit (bisher)

Eine Formel φ ist erfüllbar, wenn es eine gültige Belegung s gibt, sodass $s \models \varphi$

Problemdefinition für den SMT-Solver, z.B. in SMT-LIB

- freie Variablen (= Konstanten) $x_1 : \tau_1, \dots, x_n : \tau_n$
- Bedingungen $\phi_1, \dots, \phi_m$ über $x_1, \dots, x_n$

Lösung haben folgendes Format, z.B. via `(check-sat)`

- konkrete Belegung $s = [x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$
- sodass $s \models \phi_1 \land \cdots \land \phi_m$
- hier sind v_i *Werte* der jeweiligen Sorten (Typen) τ_i

SMT-Solver: Erfüllbarkeit (bisher)

Eine Formel φ ist erfüllbar, wenn es eine gültige Belegung s gibt, sodass $s \models \varphi$

Problemdefinition für den SMT-Solver, z.B. in SMT-LIB

- freie Variablen (= Konstanten) $x_1 : \tau_1, \dots, x_n : \tau_n$
- Bedingungen $\phi_1, \dots, \phi_m$ über $x_1, \dots, x_n$

Lösung haben folgendes Format, z.B. via `(check-sat)`

- konkrete Belegung $s = [x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$
- sodass $s \models \phi_1 \wedge \cdots \wedge \phi_m$
- hier sind v_i *Werte* der jeweiligen Sorten (Typen) τ_i

Ausreichend für Verifikation

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Allgemeine Form der SMT-LIB Probleme

- Demo: $\forall i.\ f(i) \geq i$ $\quad \forall i.\ f(i) > i$ $\quad \forall i.\ f(i) \neq i$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Allgemeine Form der SMT-LIB Probleme

- Demo: $\forall i.\ f(i) \geq i \qquad \forall i.\ f(i) > i \qquad \forall i.\ f(i) \neq i$

- freie Variablen (= Konstanten) $\quad x_1 : \tau_1, \ldots, x_n : \tau_n$
- uninterpretierte Funktionen $\quad f : \tau_1 \times \cdots \times \tau_k \to \tau$
  $g : \sigma_1 \times \cdots \times \sigma_l \to \sigma$
  $\vdots$
- Bedingungen $\quad \phi_1, \ldots, \phi_m$ über $x_1, \ldots, x_n$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Allgemeine Form der SMT-LIB Probleme

- Demo: $\forall i.\ f(i) \geq i$ $\quad \forall i.\ f(i) > i$ $\quad \forall i.\ f(i) \neq i$
- freie Variablen (= Konstanten) $\quad x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$
- uninterpretierte Funktionen $\quad f \colon \tau_1 \times \cdots \times \tau_k \to \tau$

  $g \colon \sigma_1 \times \cdots \times \sigma_l \to \sigma$

  $\vdots$
- Bedingungen $\quad \phi_1, \ldots, \phi_m$ über $x_1, \ldots, x_n$

Lösung haben folgendes Format, z.B. via `(check-sat)`,
mit konkreten Belegung der Variablen und Definitionen der Funktionen

$$\begin{aligned} s = [&x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \\ &f(z_1, \ldots, z_k) := e_f \\ &g(z_1, \ldots, z_l) := e_g, \quad \ldots] \end{aligned}$$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\, f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele: $\exists\, f.\ \forall\, x.\ f(x) \cdot f(x) = x$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\, f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele: $\exists\, f.\ \forall\, x.\ f(x) \cdot f(x) = x \qquad f(x) \coloneqq \sqrt{x}$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\, f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele: $\exists\, f.\ \forall\, x.\ f(x) \cdot f(x) = x \qquad f(x) := \sqrt{x}$
  $\exists\, f.\ \forall\, i.\ f(i) > i$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\ f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele: $\exists\ f.\ \forall\ x.\ f(x) \cdot f(x) = x$ $\quad f(x) := \sqrt{x}$
  $\exists\ f.\ \forall\ i.\ f(i) > i$ $\quad f(x) := x + 1$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\ f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele:

$$\exists\ f.\ \forall\ x.\ f(x) \cdot f(x) = x \qquad f(x) := \sqrt{x}$$
$$\exists\ f.\ \forall\ i.\ f(i) > i \qquad f(x) := x + 1$$
$$\exists\ f.\ \forall\ i.\ f(i) \neq i$$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\ f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele:

$$\exists\ f.\ \forall\ x.\ f(x) \cdot f(x) = x \qquad f(x) := \sqrt{x}$$

$$\exists\ f.\ \forall\ i.\ f(i) > i \qquad f(x) := x + 1$$

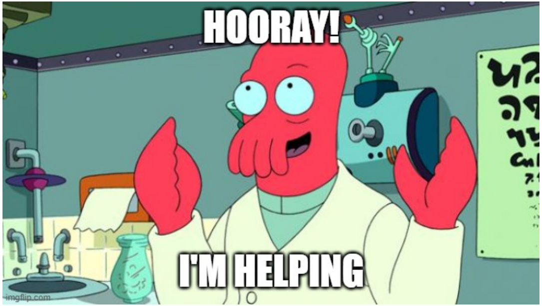
$$\exists\ f.\ \forall\ i.\ f(i) \neq i \qquad f(x) := \mathtt{ite}(x = 0, 2, 0)$$

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\ f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele:

$\exists\ f.\ \forall\ x.\ f(x) \cdot f(x) = x$	$f(x) := \sqrt{x}$
$\exists\ f.\ \forall\ i.\ f(i) > i$	$f(x) := x + 1$
$\exists\ f.\ \forall\ i.\ f(i) \neq i$	$f(x) := \texttt{ite}(x = 0, 2, 0)$



Stand der Technik: “Standard” SMT-Solver sind oft überfordert

SMT-Solver: Erfüllbarkeit mit uninterpretierten Funktionen (UF)

Synthese von Funktionen: $\exists\ f.\ \varphi(f)$

- Deklarative Programmierung!
- f wird repräsentiert durch ein funktionales Programm (z.B. Haskell)
- Beispiele:

$$\begin{array}{ll} \exists\ f.\ \forall\ x.\ f(x) \cdot f(x) = x & f(x) := \sqrt{x} \\ \exists\ f.\ \forall\ i.\ f(i) > i & f(x) := x + 1 \\ \exists\ f.\ \forall\ i.\ f(i) \neq i & f(x) := \texttt{ite}(x = 0, 2, 0) \end{array}$$

Stand der Technik: "Standard" SMT-Solver sind oft überfordert

Spezialfälle, Einschränkungen der Problemstellungen

- Funktionen über algebraischen Datentypen, z.B. funktionale Listen, Bäume
- Cyclic Horn Clauses (CHC): Invarianten, Vor-/Nachbedingungen
- Abstrakte Domänen: (typischerweise) arithmetische Constraints
- Syntaxgetriebene Synthese (SyGuS): Lösungsraum = Grammatiken

Programmsynthese (Folie bereits gezeigt)

```
data BST a where
    Empty :: BST a
    Node  :: x: a -> l: BST {a | _v < x} -> r: BST {a | x < _v} -> BST a

measure keys :: BST a -> Set a where
    Empty -> []
    Node x l r -> keys l + keys r + [x]
```

Synthese von Implementierungen:

```
insert :: x: a -> t: BST a -> {BST a | keys _v == keys t + [x]}
insert = ??
```

Tools

- Liquid Haskell `https://ucsd-progsys.github.io/liquidhaskell-blog/`
- Synquid `http://comcom.csail.mit.edu/comcom/#Synquid`
- Leon `https://leon.epfl.ch/`

Heute: Inferenz von Invarianten und Kontrakten

1. Teil: Horn-Klauseln zur Programmverifikation
   - Geschickte Codierung von Inferenzproblemen:
     Schleifen, Kontrakte, Nebenfläufigkeit, OOP Spezifikationen, ...
   - SMT-LIB gestützte Tools und Solver
   - Literatur: Bjørner, Gurfinkel, McMillan, Rybalchenko:
     Horn clause solvers for program verification, 2015.

2. Teil: Lösungsverfahren (hauptsächlich Invarianten in LIA)
   - numerische abstrakte Domänen
   - Prädikatenabstraktion, syntaxgetriebene Synthese
   - Machine Learning
   - Ausblick: Softwareverifikation (Mastervorlesung im WS, Prof. Beyer)

Hinweis: Diese Lösungsverfahren können natürlich auch verwendet werden, wenn die Programmverifikation nicht mit Horn-Klauseln realisiert ist.

Systeme von Horn-Klauseln

Gegeben:

- Variablen $\vec{x} = x_1, \ldots, x_n$ (z.B. Programmvariablen)
- unbekannte Prädikate $P_1, \ldots, P_k$ (z.B. Invarianten, Vor-/Nachbedingungen)

Systeme von Horn-Klauseln

Gegeben:

- Variablen $\vec{x} = x_1, \ldots, x_n$ (z.B. Programmvariablen)
- unbekannte Prädikate $P_1, \ldots, P_k$ (z.B. Invarianten, Vor-/Nachbedingungen)

Definition: Horn-Klausel $\varphi_1 \land \cdots \land \varphi_m \implies P_i(\vec{x})$ ist eine Implikation mit

- einem unbekannten Prädikat P_i als Konklusion
- jede Annahme φ_j ist entweder
  - ein *positives* vorkommen eines Prädikats $P_j(\vec{e})$
  - oder eine Formel über $\vec{x}$ *ohne* unbekannte Prädikate

Systeme von Horn-Klauseln

Gegeben:

- Variablen $\vec{x} = x_1, \dots, x_n$ (z.B. Programmvariablen)
- unbekannte Prädikate $P_1, \dots, P_k$ (z.B. Invarianten, Vor-/Nachbedingungen)

Definition: Horn-Klausel $\varphi_1 \land \cdots \land \varphi_m \Longrightarrow P_i(\vec{x})$ ist eine Implikation mit

- einem unbekannten Prädikat P_i als Konklusion
- jede Annahme φ_j ist entweder
  - ein *positives* vorkommen eines Prädikats $P_j(\vec{e})$
  - oder eine Formel über $\vec{x}$ *ohne* unbekannte Prädikate

Definition: eine Zielklausel $\varphi_1 \land \cdots \land \varphi_m \Longrightarrow \psi$

- hat eine Formel ψ ohne unbekannte Prädikate als Konklusion
- Äquivalent: $\varphi_1 \land \cdots \land \varphi_m \land \neg\psi \Longrightarrow \texttt{false}$

Beispiel: Horn-Klauseln in der Programmierung (Prolog)

Beschreibung von Zusammenhängen

```
parent_child(X, Y) :- mother_child(X, Y) or  father_child(X, Y).
sibling(X, Y)      :- parent_child(Z, X) and parent_child(Z, Y).
```

Beispiel: Horn-Klauseln in der Programmierung (Prolog)

Beschreibung von Zusammenhängen

```
parent_child(X, Y) :- mother_child(X, Y) or  father_child(X, Y).
sibling(X, Y)      :- parent_child(Z, X) and parent_child(Z, Y).
```

Beachte

- Prolog-Notation: Konklusion steht links
- Disjunktion lässt sich aufspalten:

$$(\phi_1 \vee \phi_2) \Longrightarrow P(\vec{x}) \quad \text{entspricht} \quad \phi_1 \Longrightarrow P(\vec{x}) \text{ and } \phi_2 \Longrightarrow P(\vec{x})$$

Beispiel: Horn-Klauseln in der Programmierung (Prolog)

Beschreibung von Zusammenhängen

```
parent_child(X, Y) :- mother_child(X, Y) or  father_child(X, Y).
sibling(X, Y)      :- parent_child(Z, X) and parent_child(Z, Y).
```

Beschreibung von Fakten über <u>endlich</u> vielen Konstanten

```
mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
```

Beispiel: Horn-Klauseln in der Programmierung (Prolog)

Beschreibung von Zusammenhängen

```
parent_child(X, Y) :- mother_child(X, Y) or  father_child(X, Y).
sibling(X, Y)      :- parent_child(Z, X) and parent_child(Z, Y).
```

Beschreibung von Fakten über endlich vielen Konstanten

```
mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
```

Anfragen: für welche Argumente sind Prädikate wahr?

```
?- sibling(sally, erica).
  Yes
```

Beispiel: Horn-Klauseln in der Programmierung (Prolog)

Beschreibung von Zusammenhängen

```
parent_child(X, Y) :- mother_child(X, Y) or  father_child(X, Y).
sibling(X, Y)      :- parent_child(Z, X) and parent_child(Z, Y).
```

Beschreibung von Fakten über endlich vielen Konstanten

```
mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
```

Anfragen: für welche Argumente sind Prädikate wahr?

```
?- sibling(sally, erica).
  Yes
```

Suchstrategie muss nicht angegeben werden!

AI Hype der 70er, zahlreiche Anwendungen in Forschung/Industrie

Ausblick: Horn-Klauseln in der Praxis (Datalog)

Reachability Analysis for AWS-Based Networks

John Backes[1], Sam Bayless[1,4], Byron Cook[1,2], Catherine Dodge[1], Andrew Gacek[1(✉)], Alan J. Hu[4], Temesghen Kahsai[1], Bill Kocik[1], Evgenii Kotelnikov[1,3], Jure Kukovec[1,5], Sean McLaughlin[1], Jason Reed[6], Neha Rungta[1], John Sizemore[1], Mark Stalzer[1], Preethi Srinivasan[1], Pavle Subotić[1,2], Carsten Varming[1], and Blake Whaley[1]

Abstract. Cloud services provide the ability to provision virtual networked infrastructure on demand over the Internet. The rapid growth of these virtually provisioned cloud networks has increased the demand for automated reasoning tools capable of identifying misconfigurations or security vulnerabilities. This type of automation gives customers the assurance they need to deploy sensitive workloads. It can also reduce the cost and time-to-market for regulated customers looking to establish compliance certification for cloud-based applications. In this industrial case-study, we describe a new network reachability reasoning tool, called TIROS, that uses off-the-shelf automated theorem proving tools to fill this need. TIROS is the foundation of a recently introduced network security analysis feature in the *Amazon Inspector* service now available to millions of customers building applications in the cloud. TIROS is also used within Amazon Web Services (AWS) to automate the checking of compliance certification and adherence to security invariants for many AWS services that build on existing AWS networking features.

Computer Aided Verification (CAV) 2019

Übersetzung: Programme → Horn-Klauseln

- *Hoare-Tripel* $\{P\}\ code\ \{Q\}$:
  Wenn vor einer Ausführung von Programmstück $code$ die Formel P gilt, dann gilt danach garantiert Q

Übersetzung: Programme $\to$ Horn-Klauseln

- *Hoare-Tripel* $\{P\}\ code\ \{Q\}$:
  Wenn vor einer Ausführung von Programmstück $code$ die Formel P gilt, dann gilt danach garantiert Q

- Regeln zur Konstruktion von *gültigen* Hoare-Tripeln per Syntax, z.B.
  - Annahmen: $\{P\}\ \mathbf{assume}\ \psi\ \{Q\}$ gültig falls $P \wedge \psi \Rightarrow Q$
  - Zusicherungen: $\{P\}\ \mathbf{assert}\ \psi\ \{Q\}$ gültig falls $P \Rightarrow \psi$ und $P \Rightarrow Q$
  - Zuweisungen: $\{P\}\ \mathsf{x} = e\ \{Q\}$ gültig falls $P \Rightarrow Q[\mathsf{x} \mapsto e]$
  - Sequenz: $\{P\}\ c_1; c_2\ \{Q\}$ gültig falls
    $\exists\ R.\ \{P\}\ c_1\ \{R\}$ und $\{R\}\ c_2\ \{Q\}$

Übersetzung: Programme $\to$ Horn-Klauseln

- *Hoare-Tripel* $\{P\}$ $code$ $\{Q\}$:
  Wenn vor einer Ausführung von Programmstück $code$ die Formel P gilt, dann gilt danach garantiert Q
- Regeln zur Konstruktion von *gültigen* Hoare-Tripeln per Syntax, z.B.
  - Annahmen: $\{P\}$ $\textbf{assume}\ \psi$ $\{Q\}$ gültig falls $P \wedge \psi \Rightarrow Q$
  - Zusicherungen: $\{P\}$ $\textbf{assert}\ \psi$ $\{Q\}$ gültig falls $P \Rightarrow \psi$ und $P \Rightarrow Q$
  - Zuweisungen: $\{P\}$ $\mathsf{x} = e$ $\{Q\}$ gültig falls $P \Rightarrow Q[\mathsf{x} \mapsto e]$
  - Sequenz: $\{P\}$ $c_1; c_2$ $\{Q\}$ gültig falls
    $\exists\ R.\ \{P\}\ c_1\ \{R\}$ und $\{R\}\ c_2\ \{Q\}$
- **Idee**: Führe *neue Prädikate* und *Seitenbedingungen* ($=$ Klauseln) durch Anwenden der Regeln des Hoare-Kalküls ein
- Suche nach Lösungen mit CHC Solvern, z.B. `(set-logic HORN)`

Beispiel: Horn-Klauseln in der Verifikations (Schleifen)

```
assume n >= 0;
for(int i=0; i++; i<n);
assert i == n;
```

Beispiel: Horn-Klauseln in der Verifikations (Schleifen)

```
assume n >= 0;
for(int i=0; i++; i<n);
assert i == n;
```

Demo: `max.smt2`

Diskussion

- Bedingungen für I können *direkt* ausgedrückt werden
- *zyklische* Klausel bei Schleifen:
  I sowohl Annahme als auch Konklusion
- Zielklausel hat I als Annahme aber nicht als Konklusion
- Initialisierungsklausel *erzwingt* sinnvolle Lösung

Beispiel: Horn-Klauseln in der Verifikation (Kontrakte)

```
int max(int x, int y) { return x > y ? x : y; }
assert max(x, y) >= x;
```

Beispiel: Horn-Klauseln in der Verifikation (Kontrakte)

```
int max(int x, int y) { return x > y ? x : y; }
assert max(x, y) >= x;
```

Demo: `max.smt2`

Diskussion

- mehrere kontextuelle vs. ein allgemeiner Kontrakt(e)
- Zielklauseln: Einschränkung *gültiger* Lösungen
  nicht alle Lösungen sind erlaubt, Modelle nicht nur `true`
- “initiale” Klauseln: *sinnvolle* Lösungen erzwingen
  es muss Lösungen geben, Modelle nicht nur `false`

Warum gerade Horn-Klauseln?

Einfache Übersetzung!

- sehr direkt:
  - ein Prädikat pro Programmstelle (Knoten im CFA)
  - eine Klausel pro Transition
- keine Kreativität notwendig $\rightarrow$ Problem wird an den Solver delegiert
  - Übersetzung konzentriert sich auf Programmsemantik
  - CHC Solver von konkreter Programmiersprache unabhängig

Effiziente Lösungsansätze existieren

- Resultion: "Einsetzen" von Klauseln ineinander
- Abkürzungen analog zu DPLL Unit-Clause/Pure Literal existieren
- Entscheidbare Teilprobleme (z.B. ohne Schleifen)
- Gegenbeispiele bei inkorrekten Programmen

Für Interessierte: Mini Dafny Verifier mit Horn-Klauseln

Übersetzung von Dafny in Horn-Klauseln
`https://github.com/gernst/minihorn`

Achtung: Evtl Anpassung auf neue Scala-Version notwendig.

Weiterführend:

- Implementierungsdatentypen: `real`, `bool`, Arrays
- Spezifikation: Quantoren, Sequenzen, Mengen, ...
- Klassen und Inferenz von Klasseninvarianten
- Anschluss von CHC Solvern (z3, Eldarica)
- Rückübersetzung von Lösungen in Dafny-Syntax
- ...

Heute: Inferenz von Invarianten und Kontrakten

1. Teil: Horn-Klauseln zur Programmverifikation
   - Geschickte Codierung von Inferenzproblemen:
     Schleifen, Kontrakte, Nebenfläufigkeit, OOP Spezifikationen, ...
   - SMT-LIB gestützte Tools und Solver
   - Literatur: Bjørner, Gurfinkel, McMillan, Rybalchenko:
     Horn clause solvers for program verification, 2015.

2. Teil: Lösungsverfahren (hauptsächlich Invarianten in LIA)
   - numerische abstrakte Domänen
   - Prädikatenabstraktion, syntaxgetriebene Synthese
   - Machine Learning
   - Ausblick: Softwareverifikation (Mastervorlesung im WS, Prof. Beyer)

Hinweis: Diese Lösungsverfahren können natürlich auch verwendet werden, wenn die Programmverifikation nicht mit Horn-Klauseln realisiert ist.

Theorie: Lineare Arithmetik der *ganzen* Zahlen (LIA)

Erweiterung der Aussagenlogik um

- Variablen $x : \mathbb{Z}$ und Konstanten $k \in \mathbb{Z}$
- Terme $t ::= x \mid k \mid -t \mid t_1 + t_2 \mid k \cdot t$
- Propositionen $A ::= \cdots \mid t_1 = t_2 \mid t_1 < t_2 \mid t_1 \leq t_2$

Grafische Interpretation:

Heute: Inferenz von Invarianten und Kontrakten

1. Teil: Horn-Klauseln zur Programmverifikation
   - Geschickte Codierung von Inferenzproblemen: Schleifen, Kontrakte, Nebenfläufigkeit, OOP Spezifikationen, ...
   - SMT-LIB gestützte Tools und Solver
   - Literatur: Bjørner, Gurfinkel, McMillan, Rybalchenko: *Horn clause solvers for program verification*, 2015.

2. Teil: Lösungsverfahren (hauptsächlich Invarianten in LIA)
   - numerische abstrakte Domänen
   - Prädikatenabstraktion, syntaxgetriebene Synthese
   - Machine Learning
   - Ausblick: Softwareverifikation (Mastervorlesung im WS, Prof. Beyer)

Hinweis: Diese Lösungsverfahren können natürlich auch verwendet werden, wenn die Programmverifikation nicht mit Horn-Klauseln realisiert ist.

Abstrakte Domänen

Idee: Führe Programm mit *abstrakten* Werten aus

- $\mathbb{Z} \quad \rightsquigarrow \quad \{-, 0, +\}$
- Pointer/Referenzen $\quad \rightsquigarrow \quad \{\texttt{null}, \texttt{non-null}\}$
- Ähnlich wie Typcheck!

Abstrakte Domänen

Idee: Führe Programm mit *abstrakten* Werten aus

- $\mathbb{Z}$ $\leadsto$ $\{-, 0, +\}$
- Pointer/Referenzen $\leadsto$ $\{\texttt{null}, \texttt{non-null}\}$
- Ähnlich wie Typcheck!

Eigenschaften

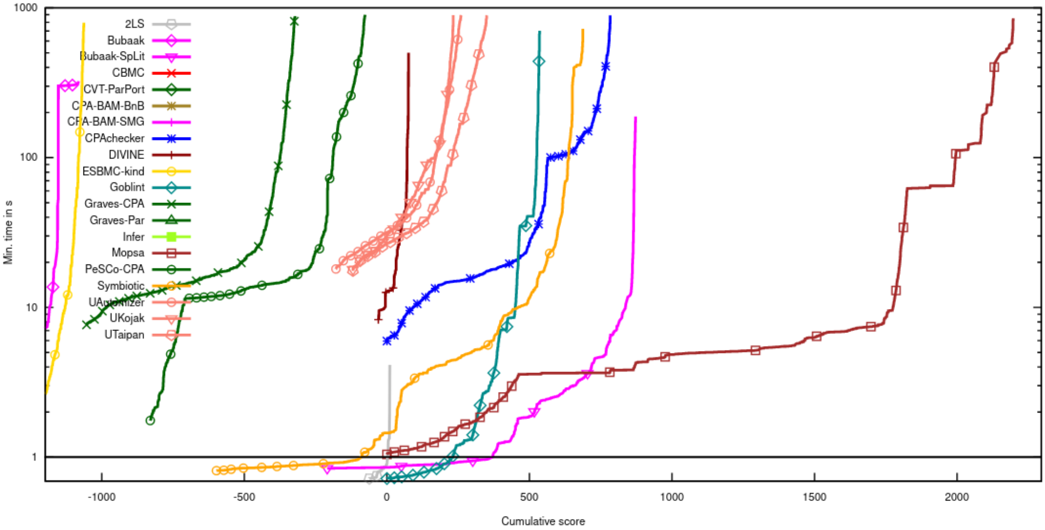
✓ Nur endlich viele Möglichkeiten: Analyse terminiert garantiert

✗ Möglicherweise unpräzises Ergebnis:

  - false positive: ein Fehler wird angezeigt der keiner ist (nervig)
  - false negative: ein Fehler wird übersehen (ungünstig)

Statische Analyse: gut geeignet für Embedded Systems

Skalierbarkeit abstrakter Domänen



SV-COMP 2024: Newcomer *Mopsa* gewinnt Kategorie “Software Systems”

Industrie: Abstrakte Domänen

Who uses Astrée?

Since 2003, Airbus France has been using Astrée in the development of safety-critical software for various aircraft series, including the A380.

In 2018, Bosch Automotive Steering replaced their legacy tools with Astrée and RuleChecker, resulting in significant savings thanks to faster analyses, higher accuracy, and optimized licensing and support costs.

Framatome employs Astrée for verification of their safety-critical TELEPERM XS platform that is used for engineering, testing, commissioning, operating and troubleshooting nuclear reactors.

The global automotive supplier Helbako in Germany is using Astrée to guarantee that no runtime errors can occur in their electronic control software and to demonstrate MISRA compliance of the code.

In 2008, Astrée proved the absence of any runtime errors in a C version of the automatic docking software of the Jules Verne Automated Transfer Vehicle, enabling ESA to transport payloads to the International Space Station.

https://www.absint.com/astree/index.htm

Abstrakte Domänen

Gegeben ein konkreter Datentyp

- Trägermenge C (z.B. $\mathbb{Z}$)
- Operationen: $+, \times, \ldots$

Eine dazu abstrakte Domäne besteht aus:

- Endliche Menge A von abstrakten Werten
- Abstraktionsfunktion $\alpha : C \to A$ (z.B. $4 \mapsto +$)
- Repräsentationsfunktion $\gamma : A \to \mathbb{P}(C)$ (z.B. $+ \mapsto \{x \in C | x > 0\}$
- Abstrakte Operationen $\widehat{+}, \widehat{\times}, \ldots$, "widening" $\nabla : A \times A \to A$

Bedingungen

- abstrakte Operationen "passen zu" konkreten, z.B. $x + y \in \gamma(\alpha(x) \widehat{+} \alpha(y))$
- widening konvergiert nach endlicher Zeit $\nabla_{i=1,\ldots,n+1} a_i = \nabla_{i=1,\ldots,n} a_i$

Beispiel: Vorzeichen

```
int i = 0;
while(*) { i++; };
assert i >= 0;
```

Beispiel: Intervalle

```
int i = -7;
while(*) { i++; };
assert i >= -7;
```

Beispiel: Polyeder

```
assume n >= 0;
for(int i=0; i++; i<n);
assert i == n;
```

Heute: Inferenz von Invarianten und Kontrakten

1. Teil: Horn-Klauseln zur Programmverifikation
   - Geschickte Codierung von Inferenzproblemen:
     Schleifen, Kontrakte, Nebenfläufigkeit, OOP Spezifikationen, ...
   - SMT-LIB gestützte Tools und Solver
   - Literatur: Bjørner, Gurfinkel, McMillan, Rybalchenko:
     Horn clause solvers for program verification, 2015.

2. Teil: Lösungsverfahren (hauptsächlich Invarianten in LIA)
   - numerische abstrakte Domänen
   - Prädikatenabstraktion, syntaxgetriebene Synthese
   - Machine Learning
   - Ausblick: Softwareverifikation (Mastervorlesung im WS, Prof. Beyer)

Hinweis: Diese Lösungsverfahren können natürlich auch verwendet werden, wenn die Programmverifikation nicht mit Horn-Klauseln realisiert ist.

Prädikatenabstraktion [Graf and Saïdi, CAV 1997]

Spezielle abstrakte Domäne!

- Schablonen für Kandidaten für Teile von Invarianten, z.B.
  $\_ = \_ \quad \_ \leq \_ \quad \ldots$
- Instanziierung mit Programmvariablen und Konstanten, z.B.
  $i = n \quad i = 0 \quad i \leq 0 \quad \ldots$
  $p = \texttt{null} \quad p \neq \texttt{null} \quad \ldots$

Prädikatenabstraktion [Graf and Saïdi, CAV 1997]

Spezielle abstrakte Domäne!

- Schablonen für Kandidaten für Teile von Invarianten, z.B.
  $\_ = \_ \quad \_ \leq \_ \quad \dots$
- Instanziierung mit Programmvariablen und Konstanten, z.B.
  $i = n \quad i = 0 \quad i \leq 0 \quad \dots$
  $p = \texttt{null} \quad p \neq \texttt{null} \quad \dots$

Eigenschaften

- ✓ beliebige Prädikate möglich
- ✓ dynamische Anpassung der Abstraktion: Kosten vs Präzision
- ✗ exponentiell viele Kombinationen
- ✗ Disjunktive/bedingte Formeln und Quantoren schwierig auszudrücken

Prädikatenabstraktion [Graf and Saïdi, CAV 1997]

Auszeichnung mit dem CAV Award 2022
(links: Orna Grumberg, rechts: Susanne Graf, Foto: Armin Biere)

Houdini, an Annotation Assistant for ESC/Java

For any field `f` declared in the program, we guess the following candidate invariants for `f`:

Type of `f`	Candidate invariants for `f`
integral type	`//@ invariant f` *cmp expr*`;`
reference type	`//@ invariant f != null;`

array type	`//@ invariant f != null;`
	`//@ invariant \nonnullelements(f);`
	`//@ invariant (\forall int i; 0 <= i && i <` *expr* `==> f[i] != null);`
	`//@ invariant f.length` *cmp expr*`;`

boolean	`//@ invariant f == false;` `//@ invariant f == true;`

Houdini, an Annotation Assistant for ESC/Java

refuted. ESC/Java would then infer that the method never returns and would not check code following a call to this method. Therefore, we only guess the following consistent postconditions for each library method:

Result type	Optimistic postconditions
integral type	`//@ ensures \result >= 0;`
reference type	`//@ ensures \result != null;`
array type	`//@ ensures \result != null;` `//@ ensures \nonnullelements(\result);`

We guess optimistic preconditions and invariants in a similar manner.

Houdini, an Annotation Assistant for ESC/Java

Type of annotation	Preconditions		Postconditions		Invariants		Total	
	guessed	%valid	guessed	%valid	guessed	%valid	guessed	%valid
`f == expr`	2130	18	985	18	435	14	3550	17
`f != expr`	2130	35	985	35	435	38	3550	35
`f < expr`	2130	26	985	27	435	24	3550	26
`f <= expr`	2130	31	985	32	435	36	3550	33
`f >= expr`	2130	25	985	21	435	19	3550	32
`f > expr`	2130	31	985	36	435	35	3550	23
`f != null`	509	92	229	79	983	72	1721	79
`\nonnullelems(f)`	54	81	21	62	36	64	111	72
`(\forall ...)`	841	27	260	37	125	59	1226	32
`f == false`	47	36	51	25	39	10	137	20
`f == true`	47	28	51	24	39	8	137	25
`\fresh(\result)`	0	0	229	30	0	0	229	30
`false`	780	17	0	0	0	0	780	17
exact type	37	19	11	36	14	57	62	31
Total	15095	30	6762	30	3846	40	25703	31

Table 2: Numbers of candidate annotations generated on the Cobalt program by the heuristics of Section 3, and the percentages of these annotations that are valid.

Syntaxgetriebene Synthese

Vormals: Schablonen $\_ = \_ \quad \_ \leq \_ \quad \ldots$

Jetzt: beliebige (kontextfreie) Grammatiken, z.B.
$\mathit{Expr}\ e ::= 0 \mid 1 \mid e + e \mid e \cdot e \mid x_i$ wobei x_i Programmvariable

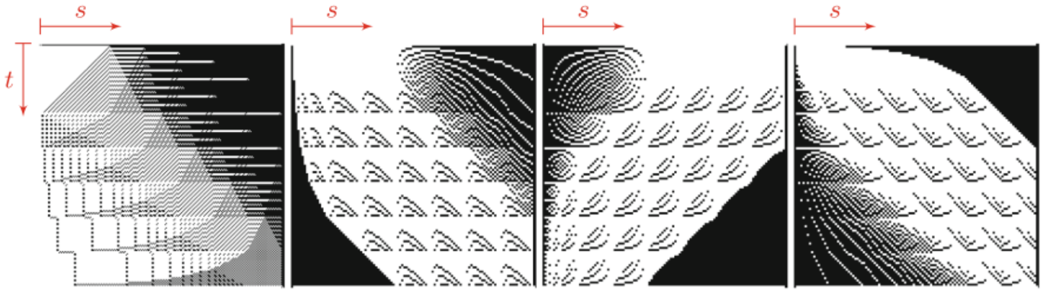
Problembeschreibung:

- finde e mit $\phi(e)$ allgemeingültig
- und e wird von der gegeben Grammatik erzeugt

Lösungsverfahren: when in doubt, use brute force (Ken Thompson)

- Enumeration aller gültigen Ausdrücke gemäß der Grammatik
- Beweisversuch für jeden Kandidaten
- Suchheuristik: kleine Ausdrücke zuerst, Bewertungsfunktion, Auswahl gemäß Wahrscheinlichkeitsverteilung, ...

Invertibility Conditions for Floating-Point Formulas, CAV 2019



(a) $x + s \approx t$ (b) $x \cdot s \approx t$ (c) $x \div s \approx t$ (d) $s \div x \approx t$

Fig. 1. Invertibility conditions for $\{+, \cdot, \div\}$ over $\approx$ for $\mathbb{F}_{3,5}$ and rounding mode RNE.

Tool: CVC4 Anwendung: Finden von Inversen von Fließkommaoperatoren

Invertibility Conditions for Floating-Point Formulas, CAV 2019

Table 2. Invertibility conditions for floating-point operators (excl. fma) with $\approx$.

Literal	Invertibility condition
$x \overset{\mathsf{R}}{+} s \approx t$	$t \approx (t \overset{\mathsf{RTP}}{-} s) \overset{R}{+} s \vee t \approx (t \overset{\mathsf{RTN}}{-} s) \overset{R}{+} s \vee s \approx t$
$x \overset{\mathsf{R}}{-} s \approx t$	$t \approx (s \overset{\mathsf{RTP}}{+} t) \overset{R}{-} s \vee t \approx (s \overset{\mathsf{RTN}}{+} t) \overset{R}{-} s \vee (s \not\approx t \wedge s \approx \pm\infty \wedge t \approx \pm\infty)$
$s \overset{\mathsf{R}}{-} x \approx t$	$t \approx s \overset{R}{+} (t \overset{\mathsf{RTP}}{-} s) \vee t \approx s \overset{R}{+} (t \overset{\mathsf{RTN}}{-} s) \vee s \approx t$
$x \overset{\mathsf{R}}{\cdot} s \approx t$	$t \approx (t \overset{\mathsf{RTP}}{\div} s) \overset{R}{\cdot} s \vee t \approx (t \overset{\mathsf{RTN}}{\div} s) \overset{R}{\cdot} s \vee (s \approx \pm\infty \wedge t \approx \pm\infty) \vee (s \approx \pm 0 \wedge t \approx \pm 0)$
$x \overset{\mathsf{R}}{\div} s \approx t$	$t \approx (s \overset{\mathsf{RTP}}{\cdot} t) \overset{R}{\div} s \vee t \approx (s \overset{\mathsf{RTN}}{\cdot} t) \overset{R}{\div} s \vee (s \approx \pm\infty \wedge t \approx \pm 0) \vee (t \approx \pm\infty \wedge s \approx \pm 0)$
$s \overset{\mathsf{R}}{\div} x \approx t$	$t \approx s \overset{R}{\div} (s \overset{\mathsf{RTP}}{\div} t) \vee t \approx s \overset{R}{\div} (s \overset{\mathsf{RTN}}{\div} t) \vee (s \approx \pm\infty \wedge t \approx \pm\infty) \vee (s \approx \pm 0 \wedge t \approx \pm 0)$
$x \mathbin{\mathsf{rem}} s \approx t$	$t \approx t \mathbin{\mathsf{rem}} s$
$s \mathbin{\mathsf{rem}} x \approx t$	?
$\sqrt[\mathsf{R}]{x} \approx t$	$t \approx \sqrt[R]{(t \overset{\mathsf{RTP}}{\cdot} t)} \vee t \approx \sqrt[R]{(t \overset{\mathsf{RTN}}{\cdot} t)} \vee t \approx \pm 0$
$\lvert x \rvert \approx t$	$\neg \mathsf{isNeg}(t)$
$-x \approx t$	$\top$
$\overset{\mathsf{R}}{\mathsf{rti}}(x) \approx t$	$t \approx \overset{R}{\mathsf{rti}}(t)$

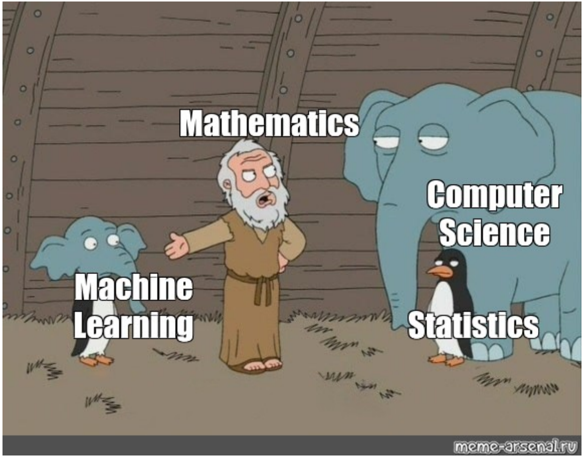
Heute: Inferenz von Invarianten und Kontrakten

1. Teil: Horn-Klauseln zur Programmverifikation
   - Geschickte Codierung von Inferenzproblemen:
     Schleifen, Kontrakte, Nebenfläufigkeit, OOP Spezifikationen, ...
   - SMT-LIB gestützte Tools und Solver
   - Literatur: Bjørner, Gurfinkel, McMillan, Rybalchenko:
     Horn clause solvers for program verification, 2015.

2. Teil: Lösungsverfahren (hauptsächlich Invarianten in LIA)
   - numerische abstrakte Domänen
   - Prädikatenabstraktion, syntaxgetriebene Synthese
   - Machine Learning
   - Ausblick: Softwareverifikation (Mastervorlesung im WS, Prof. Beyer)

Hinweis: Diese Lösungsverfahren können natürlich auch verwendet werden, wenn die Programmverifikation nicht mit Horn-Klauseln realisiert ist.

Machine Learning zur Generierung von Invarianten



Aktueller Stand und Prognose: Spannend! Aber analog zu Copilot für Code

- ✓ wird typische Invarianten und *fast* richtige Kandidaten finden
- ✗ gelegentlich Blödsinn, kann keine anwendungsspezifischen Eigenschaften

Zusammenfassung

Heute gelernt:

- Ansätze und Techniken Inferenz von Invarianten und Vor-/Nachbedingungen

Sehr aktives Forschungsfeld!

Zusammenfassung

Heute gelernt:

- Ansätze und Techniken Inferenz von Invarianten und Vor-/Nachbedingungen

Sehr aktives Forschungsfeld!

Was Sie wissen und können sollten

- Repräsentation von Verifikationsproblemen als Systeme von Horn-Klauseln
- Kenntnis über Tools und Formate (z.B. SMT-LIB)
- Überblick über die Lösungsansätze
- Diskussion: Möglichkeiten und Limitationen

License

These slides are licensed under the creative commons license:
https://creativecommons.org/licenses/by-nc-nd/4.0/

Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0)

- give appropriate credit
- distribute without modifications
- do not use for commercial purposes