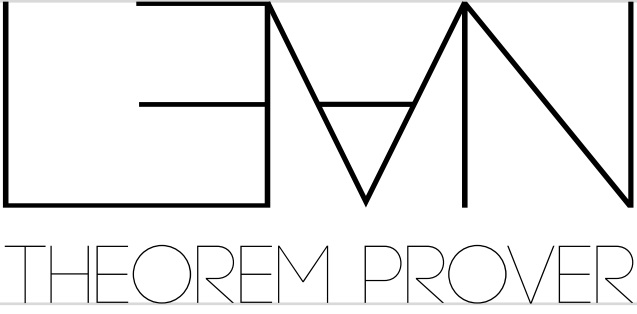
Karlsruhe Institute of Technology

Theorembeweiserpraktikum

Group Projects

Jakob von Raumer, Sebastian Ullrich | SS 2021

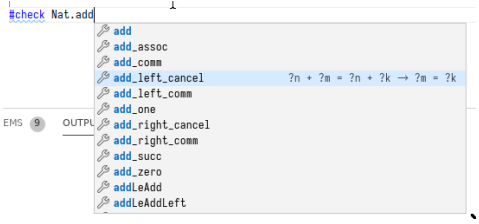


Organisatorisches zum Projekt

- Projektbeginn: 1.6.2021 (heute)
- Bearbeitung zu zweit oder dritt
- Lean-Rahmen zu den Projekten jetzt im Repo, diese Folien auf unserer Webseite
- Abgabe: 5.7.2021, 12:00 Uhr via Praktomat
- ab jetzt: freiwillige Dienstagstermine mit weiterführendem Material
  Wünsche dazu gerne an uns.
- bei Problemen frühzeitig melden!
  - während den restlichen Terminen, in Zulip oder per Mail
- Projektpräsentation: Woche vom 13.7.2021, Termin(e) und Medium TBA
- Infos dazu: 6.7.2021, 14:00 Uhr

Projektbearbeitung

- Beweise sollten lesbar und vollständig strukturiert (siehe `lecture4`) sein
- Zusammenarbeit regeln ist euch überlassen, z.B. über Git
  - Für Gitpod müssen Repos public sein, bitte nach dem Praktikum dann auf private ändern
  - Wir stehen auch bei allen Fragen zu lokaler Installation von Lean bereit
- Theoreme aus der stdlib könnten hilfreich sein, auffindbar z.B. über code completion



Arbeiten mit Lean-Packages

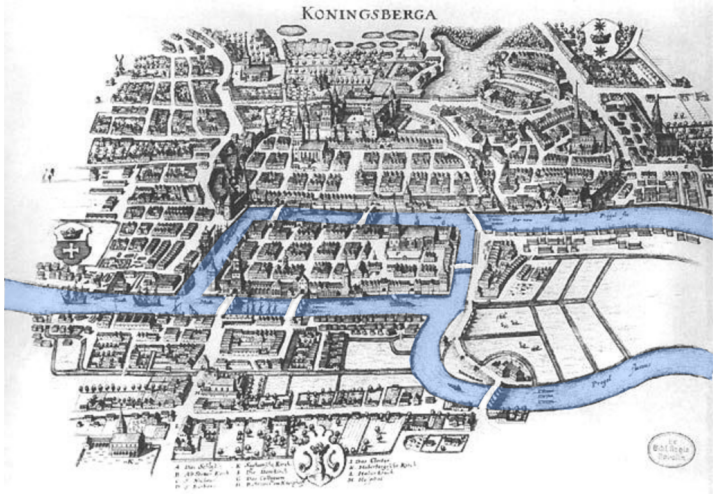
Die Vorlagen bestehen aus mehreren Dateien, die mit `import` verknüpft sind. Z.B. verweist `import TBA.While.Com` auf `TBA/While/Com.lean`.

- Das Wurzelverzeichnis (das mit `TBA.lean`) *muss* mit `File > Open Folder...` geöffnet werden, sonst funktionieren die Imports nicht
  - mach Gitpod freundlicherweise schon automatisch
- Änderungen sind in anderen Dateien erst sichtbar, nachdem die Änderung gespeichert wurde und in der importierenden Datei `Lean 4: Refresh File Dependencies` (`Strg+Shift+X`) ausgeführt wurde
  - am einfachsten nur die `Solution.lean` bearbeiten

Part I

Eulerian Circuits in Graphs

The Seven Bridges of Königsberg



Euler Circuits

Definition

Let G be a directed multigraph. A path in G is called a *Euler path* if it contains every edge of G exactly once. A closed Eulerian path is called a *Euler circuit*.

Theorem

Every directed multigraph, which

- *is non-empty,*
- *is strongly connected (i. e. there is a path between any two vertices), and*
- *at every vertex has equal in degree and out degree*

has a Eulerian circuit.

Multigraphs in Lean

As a fixed specification you will work with graphs being lists of ordered pairs of a given type with decidable equality:

```
variable {α : Type} (E : List (α × α)) [DecidableEq α]
```

Since order of edges in the list doesn't matter, we provide you with definitions to work with lists up to permutation:

```
def isPermEqvTo : Prop := ∀ a, as.count a = bs.count a
infixl:50 " ≃ " => isPermEqvTo -- Type as \simeq

def isPermSubOf : Prop := ∀ a, as.count a ≤ bs.count a
infixl:50 " ⊆ " => isPermSubOf -- Type as \sub
```

General Advice

- Start by proving the statement on paper!
- Divide and Conquer: Partition the problem into self-contained sub-problems.
- Useful definitions, lemmas. Sometimes it's not too bad if you don't need them all.
- Find a good naming scheme for those.
- You're welcome to ask for advice in the sessions in the coming weeks.

Part II

Formale Semantik

"there are formal semantics, informal semantics, and then there's the C standard" – @johnregehr

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Syntax:
- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Was ist Semantik?

Karlsruhe Institute of Technology

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Syntax:
- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Semantik:
- Bedeutung der einzelnen Sprachkonstrukte
- Bei Programmiersprachen verschiedenste Darstellungsweisen:
  - implizit (über eine Implementierung definiert)
  - informal (Beispiele, erläuternder Text etc.)
  - formal (Regelsysteme, Funktionen etc.)
- Angegeben im Sprachstandard (oft sehr vermischt mit Syntax)

Operationale Semantik

- Simuliert Zustandsübergänge auf abstrakter Maschine
- nahe an tatsächlichem Programmverhalten
- *Big-Step-Semantik*:
  Programm (= initiale Anweisung) + Startzustand wertet zu Endzustand aus
  Syntax: $\langle c, \sigma \rangle \Rightarrow \sigma'$
  Anweisung c in Zustand σ liefert Endzustand σ'

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Val.int i` und `Val.bool b`

- Konstanten `Expr.const v`
- Variablenzugriffe `Expr.var x`
- binäre Operatoren `Expr.binOp` e_1 `BinOp.eq` e_2 , ...
  für $==$, &&, $<$, $+$, $-$

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Val.int i` und `Val.bool b`

- Konstanten `Expr.const v`
- Variablenzugriffe `Expr.var x`
- binäre Operatoren `Expr.binOp` e_1 `BinOp.eq` e_2 , ...
  für $==$, &&, $<$, $+$, $-$

Programmanweisungen:

- `Com.skip`
- Variablenzuweisung `Com.ass x e` (`x ::= e`)
- sequentielle Komposition (Hintereinanderausführung) `Com.seq c c'` (`c;; c'`)
- if-then-else `Com.cond b` c_t c_e
- while-Schleifen `Com.while b c`

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Val.int i` und `Val.bool b`

- Konstanten `Expr.const v`
- Variablenzugriffe `Expr.var x`
- binäre Operatoren `Expr.binOp` e_1 `BinOp.eq` e_2 , ...
  für $==$, $\&\&$, $<$, $+$, $-$

Programmanweisungen:

- `Com.skip`
- Variablenzuweisung `Com.ass x e` (`x ::= e`)
- sequentielle Komposition (Hintereinanderausführung) `Com.seq c c'` (`c;; c'`)
- if-then-else `Com.cond b` c_t c_e
- while-Schleifen `Com.while b c`

Zustand:
beschreibt, welche Werte aktuell in den Variablen (partielle Map)

Auswertung von Ausdrücken

Über direkte Rekursion

```
variable (σ : State)
def Expr.eval : Expr → Option Val
  | Expr.const v => some v
  | Expr.var x   => σ x
  | Expr.binop e₁ op e₂ => match eval e₁, eval e₂ with
    | some v₁, some v₂ => op.eval v₁ v₂  -- BinOp.eval : BinOp → Val → Val → Option Val
    | _,       _       => none
```

Einschub: Regelsysteme

Induktive Prädikate werden in der Literatur üblicherweise als *Regelsysteme* präsentiert

```
variable (r : α → α → Prop)
inductive RTC : α → α → Prop where
  | refl : RTC a a
  | trans : r a b → RTC b c → RTC a c
```

$$\frac{}{a \xrightarrow{*} a} \qquad \frac{a \to b \qquad b \xrightarrow{*} c}{a \xrightarrow{*} c}$$

Big-Step-Regeln

$[\![e]\!]\sigma \equiv$ `Expr.eval σ e`

$$\frac{}{\langle \texttt{Com.skip}, \sigma \rangle \Rightarrow \sigma}$$

Big-Step-Regeln

$[\![e]\!]\sigma \equiv$ `Expr.eval σ e`

$$\frac{}{\langle \texttt{Com.skip}, \sigma\rangle \Rightarrow \sigma} \qquad \frac{}{\langle x ::= e, \sigma\rangle \Rightarrow \sigma[x \mapsto [\![e]\!]\sigma]}$$

Big-Step-Regeln

$[\![e]\!]\sigma \equiv$ `Expr.eval σ e`

$$\frac{}{\langle \texttt{Com.skip}, \sigma\rangle \Rightarrow \sigma} \qquad \frac{}{\langle x ::= e, \sigma\rangle \Rightarrow \sigma[x \mapsto [\![e]\!]\sigma]}$$

$$\frac{\langle c, \sigma\rangle \Rightarrow \sigma' \qquad \langle c', \sigma'\rangle \Rightarrow \sigma''}{\langle c;; c', \sigma\rangle \Rightarrow \sigma''}$$

Big-Step-Regeln

$[\![e]\!]\sigma \equiv$ `Expr.eval σ e`

$$\frac{}{\langle \texttt{Com.skip}, \sigma\rangle \Rightarrow \sigma} \qquad \frac{}{\langle x ::= e, \sigma\rangle \Rightarrow \sigma[x \mapsto [\![e]\!]\sigma]}$$

$$\frac{\langle c, \sigma\rangle \Rightarrow \sigma' \qquad \langle c', \sigma'\rangle \Rightarrow \sigma''}{\langle c;; c', \sigma\rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{Com.cond } b\ c\ c', \sigma\rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{some false} \qquad \langle c', \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{Com.cond } b\ c\ c', \sigma\rangle \Rightarrow \sigma'}$$

Big-Step-Regeln

$[\![e]\!]\sigma \equiv$ `Expr.eval σ e`

$$\frac{}{\langle \texttt{Com.skip}, \sigma\rangle \Rightarrow \sigma} \qquad \frac{}{\langle x ::= e, \sigma\rangle \Rightarrow \sigma[x \mapsto [\![e]\!]\sigma]}$$

$$\frac{\langle c, \sigma\rangle \Rightarrow \sigma' \qquad \langle c', \sigma'\rangle \Rightarrow \sigma''}{\langle c;; c', \sigma\rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{Com.cond } b\ c\ c', \sigma\rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{some false} \qquad \langle c', \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{Com.cond } b\ c\ c', \sigma\rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma' \qquad \langle \texttt{Com.while } b\ c, \sigma'\rangle \Rightarrow \sigma''}{\langle \texttt{Com.while } b\ c, \sigma\rangle \Rightarrow \sigma''} \qquad \frac{[\![b]\!]\sigma = \texttt{some false}}{\langle \texttt{Com.while } b\ c, \sigma\rangle \Rightarrow \sigma}$$

Ausführungsdauer

Erweiterung der Auswertungsfunktionen für Ausdrücke und der Big-Step-Semantik um einen Zeitbegriff:

Konstanten:	1
Variablen:	1
je bin. Operation:	1 + Auswertung beider Argumente
skip:	1
ass:	1 + Auswertung des arith. Ausdrucks
seq:	1 + Auswertung beider Anweisungen
if:	1 + Auswertung des bool. Ausdrucks + Dauer des gew. Zweigs
whileFalse:	1 + Auswertung des bool. Ausdrucks
whileTrue:	1 + Auswertung des bool. Ausdrucks + Dauer für Rumpf + Rest-Dauer

Formalisierung in Lean

Siehe Rahmen

Part III

Typsystem

Typsystem

Typsystem ordnet jedem Ausdruck Typ zu

Zwei Typen: `Ty.bool` und `Ty.int`

Typumgebung Γ: Partielle Map von Variablen nach Typ

Zwei Stufen:

1. Ausdruck e hat Typ τ unter Typumgebung Γ
   Syntax: $\Gamma \vdash e : \tau$
2. Anweisung c ist wohltypisiert unter Typumgebung Γ
   Syntax: $\Gamma \vdash c$

auch Typsystem definierbar als induktives Prädikat

Regeln

Ausdrücke:

- Konstanten haben Typ des Werts
- Variablen haben den in Typumgebung gespeicherten Typ
- Operatoren haben, wenn Unterausdrücke Typen passend zu Operator, Typ des Resultats
  z.B. bei `BinOp.lt` : Unterausdrücke `int` , ganzer Operator `bool`

Regeln

Ausdrücke:

- Konstanten haben Typ des Werts
- Variablen haben den in Typumgebung gespeicherten Typ
- Operatoren haben, wenn Unterausdrücke Typen passend zu Operator, Typ des Resultats
  z.B. bei `BinOp.lt` : Unterausdrücke `int` , ganzer Operator `bool`

Anweisungen:

- `Com.skip` typt immer
- $x ::= e$ typt, wenn Typ der Variable x in Typumgebung Γ gleich Typ des Ausdruck e
- Sequenz typt, wenn beide Unteranweisungen typen
- `if` und `while` typen, wenn Unteranweisungen typen und Prädikat vom Typ `bool`

Part IV

Projekt: Konstantenfaltung und -propagation

"The easy part of laundry is putting it in the washer and dryer. The part I dislike about it is the constant folding. :/" – @Andrewanthony91

Motivation

- Konstantenfaltung und -propagation sind wichtige Optimierungen in Compilern
- verringern *Registerdruck* (Anzahl der benötigten Register)
- Korrektheit dieser Optimierungen essentiell
- Korrektheit zu zeigen bzgl. formaler Semantik

Konstantenfaltung

Optimierung für Ausdrücke

- Wenn Berechnungen nur auf Konstanten, Ergebnis einfach einsetzen:
  `5 + 3` wird zu `8`
  `4 == 7` wird zu `false`
- Wenn mind. eine Variable, einfach beibehalten:
  `y - 3` bleibt `y - 3`
- nicht sinvolle Ausdrücke auch beibehalten:
  `5 + true` bleibt `5 + true`
- Wenn Ausdruck nur Konstante oder Variable, auch beibehalten:
  `5` bleibt `5`
  `y` bleibt `y`

Konstantenpropagation

Optimierung für Anweisungen

- Idee: Merken von Variablen, die konstant deklariert sind
- ermöglicht Ersetzen der Variable durch konstanten Wert
- dadurch möglich, `if`- Anweisungen zu vereinfachen
- Benötigt *Map* von Variablen nach Werten
- verwendet auch Konstantenfaltung

Beispiele

```
x := 7;
y := 3;
if (x = y) {
  y := x + 2;
} else {
  y := x - z;
  z := y;
}
```

wird zu

```
x := 2;
y := x;
b := x = y;
if (b) {
  z := x + y;
} else {
  z := x;
}
```

wird zu

Beispiele

```
x := 7;
y := 3;
if (x = y) {
  y := x + 2;
} else {
  y := x - z;
  z := y;
}
```

wird zu

```
x := 7;
y := 3;
y := 7 - z;
z := y;
```

```
x := 2;
y := x;
b := x = y;
if (b) {
  z := x + y;
} else {
  z := x;
}
```

wird zu

finale Map: `[x ↦ Val.int 7]`

Beispiele

```
x := 7;
y := 3;
if (x = y) {
  y := x + 2;
} else {
  y := x - z;
  z := y;
}
```

wird zu

```
x := 7;
y := 3;
y := 7 - z;
z := y;
```

finale Map: `[x ↦ Val.int 7]`

```
x := 2;
y := x;
b := x = y;
if (b) {
  z := x + y;
} else {
  z := x;
}
```

wird zu

```
x := 2;
y := 2;
b := true;
z := 4;
```

finale Map: `[x ↦ 2, y ↦ 2, b ↦ true, z ↦ 4]`

`while`

Wie `if` könnte man auch `while` vereinfachen:

- falls Prädikat konstant `false`, komplettes `while` durch `Com.skip` ersetzen
- falls Prädikat konstant `true`, Prädikat umschreiben, ansonsten Schleife beibehalten und in Schleifenkörper weiter Konstanten propagieren

while

Wie `if` könnte man auch `while` vereinfachen:

- falls Prädikat konstant `false`, komplettes `while` durch `Com.skip` ersetzen
- falls Prädikat konstant `true`, Prädikat umschreiben, ansonsten Schleife beibehalten und in Schleifenkörper weiter Konstanten propagieren

Problem: Konstanten im Schleifenkörper beeinflussen auch Prädikat!
Beispiel:

```
x := 5;
y := 1;
while (x < 7) {
  if (y = 4) {
    x := 9;
  }
  y := y + 1;
}
```

Darf das Prädikat von `while` vereinfacht werden?

`while`

- Kompletter Algorithmus bräuchte Fixpunktiteration!
- Zu kompliziert, deshalb Vereinfachung:
  Ist das Prädikat konstant `false`, ist alles in Ordnung. Ansonsten
  löschen wir beim `while` die bisher gesammelte Konstanteninformation, beginnen also mit `Map.empty`
  bei Schleifenbedingung, -körper *und* Restprogramm
- Ergebnis ist immer noch korrekt, aber nicht optimal vereinfacht
- Algorithmus so aber viel einfacher zu formalisieren

Projektaufgaben

1) Beweis, dass die vorgeg. Semantik deterministisch ist
   (sowohl im Endzustand, als auch im Zeitbegriff)
2) Formalisierung von Konstantenpropagation inklusive -faltung
3) Beweis, dass Konstantenpropagation Semantik erhält
   anders gesagt: "Wenn Originalanweisung für gegebenen Anfangszustand terminiert, dann tut das auch die resultierende Anweisung der Konstantenpropagation, mit gleichem Endzustand"
   - Rückrichtung eigentlich auch notwendig, werden wir aber ausklammern
4) Beweis, dass sich die Ausführungsgeschwindigkeit durch Konstantenpropagation nicht verringert
5) Beweis, dass zwei-/mehrfache Anwendung der Konstantenpropagation das Programm nicht weiter verändert

6a)
   1) Formalisierung der Typisierungsregeln aus diesen Folien
   2) Beweis, dass Konstantenpropagation Typisierung erhält
      anders gesagt: "Wenn Originalanweisung typt, dann auch resultierende Anweisung der Konstantenpropagation"

Projektaufgaben

1) Beweis, dass die vorgeg. Semantik deterministisch ist
   (sowohl im Endzustand, als auch im Zeitbegriff)
2) Formalisierung von Konstantenpropagation inklusive -faltung
3) Beweis, dass Konstantenpropagation Semantik erhält
   anders gesagt: "Wenn Originalanweisung für gegebenen Anfangszustand terminiert, dann tut das auch die resultierende Anweisung der Konstantenpropagation, mit gleichem Endzustand"
   - Rückrichtung eigentlich auch notwendig, werden wir aber ausklammern
4) Beweis, dass sich die Ausführungsgeschwindigkeit durch Konstantenpropagation nicht verringert
5) Beweis, dass zwei-/mehrfache Anwendung der Konstantenpropagation das Programm nicht weiter verändert

6b) *Alternatives Vorgehen:*

   1) Typisierung *direkt in* `Expr` *und* `Com` – als induktive Typfamilien über den Typkontext

```
inductive Expr (Γ : Ctxt) : Ty → Type where ...
```

   2) Anpassung der Konstantenpropagation und obigen Beweise an diese Typen

Hinweise

- erst formalisieren, dann beweisen!
  Beispiele mittels `#reduce` prüfen (z.B. Beispielprogramme in `TBA/While/Com.lean`)
- für die Beweise überlegen: welche Beziehungen müssen zwischen Semantikzustand, Typumgebung und Konstantenmap existieren?