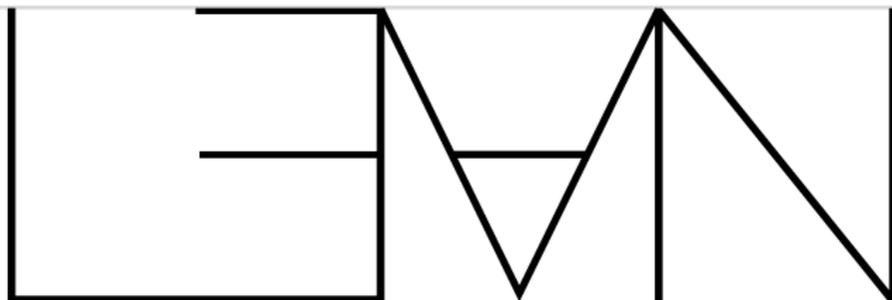




Theorembeweiserpraktikum

Group Projects

Jakob von Raumer, Sebastian Ullrich | SS 2022



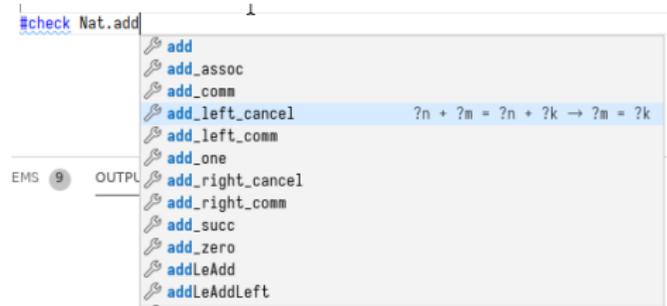
THEOREM PROVER

Organisatorisches zum Projekt

- Projektbeginn: 31.5.2022 (heute)
- Bearbeitung zu dritt
- Lean-Rahmen zu den Projekten jetzt im Repo, diese Folien auf unserer Webseite
- Abgabe: 18.7.2022, 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 oder per Mail
- Projektpräsentation: Woche vom 25.7.2021 (letzte Vorlesungswoche), Termin TBA
- Infos dazu: 19.7.2022, 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



```
#check Nat.add
```

- add
- add_assoc
- add_comm
- add_left_cancel $?n + ?m = ?n + ?k \rightarrow ?m = ?k$
- add_left_comm
- add_one
- add_right_cancel
- add_right_comm
- add_succ
- add_zero
- addLeAdd
- addLeAddLeft

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
 - macht 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 (Ctrl+Shift+X)` ausgeführt wurde
 - am einfachsten nur die `Solution.lean` bearbeiten

Part I

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?

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 e1 BinOp.eq e2 , ...`
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 e ct ce`
- while-Schleifen `Com.while e 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 e1 BinOp.eq e2 , ...`
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 e ct ce`
- while-Schleifen `Com.while e c`

Zustand:

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

Auswertung von Ausdrücken

Über direkte Rekursion

```
variable ( $\sigma$  : State)
def Expr.eval : Expr  $\rightarrow$  Option Val
| .const v => some v
| .var x   =>  $\sigma$  x
| .binop e1 op e2 => match eval e1, eval e2 with
| some v1, some v2 => op.eval v1 v2 -- BinOp.eval : BinOp  $\rightarrow$  Val  $\rightarrow$  Val  $\rightarrow$  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
  
```

$$\frac{}{a \overset{*}{\rightarrow} a} \qquad \frac{a \rightarrow b \quad b \overset{*}{\rightarrow} c}{a \overset{*}{\rightarrow} c}$$

Big-Step-Regeln

$\llbracket e \rrbracket \sigma \equiv \text{Expr.eval } \sigma \ e$

$\overline{\langle \text{Com.skip}, \sigma \rangle \Rightarrow \sigma}$

Big-Step-Regeln

$\llbracket e \rrbracket \sigma \equiv \text{Expr.eval } \sigma \ e$

$$\overline{\langle \text{Com.skip}, \sigma \rangle \Rightarrow \sigma}$$

$$\overline{\langle x ::= e, \sigma \rangle \Rightarrow \sigma[x \mapsto \llbracket e \rrbracket \sigma]}$$

Big-Step-Regeln

$\llbracket e \rrbracket \sigma \equiv \text{Expr.eval } \sigma \ e$

$$\overline{\langle \text{Com.skip}, \sigma \rangle \Rightarrow \sigma} \qquad \overline{\langle x ::= e, \sigma \rangle \Rightarrow \sigma[x \mapsto \llbracket e \rrbracket \sigma]}$$

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

Big-Step-Regeln

$\llbracket e \rrbracket \sigma \equiv \text{Expr.eval } \sigma \ e$

$$\frac{}{\langle \text{Com.skip}, \sigma \rangle \Rightarrow \sigma} \quad \frac{}{\langle x ::= e, \sigma \rangle \Rightarrow \sigma[x \mapsto \llbracket e \rrbracket \sigma]}$$

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

$$\frac{\llbracket b \rrbracket \sigma = \text{some true} \quad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \text{Com.cond } b \ c \ c', \sigma \rangle \Rightarrow \sigma'} \quad \frac{\llbracket b \rrbracket \sigma = \text{some false} \quad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \text{Com.cond } b \ c \ c', \sigma \rangle \Rightarrow \sigma'}$$

Big-Step-Regeln

$\llbracket e \rrbracket \sigma \equiv \text{Expr.eval } \sigma \ e$

$$\frac{}{\langle \text{Com.skip}, \sigma \rangle \Rightarrow \sigma} \quad \frac{}{\langle x ::= e, \sigma \rangle \Rightarrow \sigma[x \mapsto \llbracket e \rrbracket \sigma]}$$

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

$$\frac{\llbracket b \rrbracket \sigma = \text{some true} \quad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \text{Com.cond } b \ c \ c', \sigma \rangle \Rightarrow \sigma'} \quad \frac{\llbracket b \rrbracket \sigma = \text{some false} \quad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \text{Com.cond } b \ c \ c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{\llbracket b \rrbracket \sigma = \text{some true} \quad \langle c, \sigma \rangle \Rightarrow \sigma' \quad \langle \text{Com.while } b \ c, \sigma' \rangle \Rightarrow \sigma''}{\langle \text{Com.while } b \ c, \sigma \rangle \Rightarrow \sigma''} \quad \frac{\llbracket b \rrbracket \sigma = \text{some false}}{\langle \text{Com.while } b \ c, \sigma \rangle \Rightarrow \sigma}$$

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 II

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 III

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 sinnvolle Ausdrücke auch beibehalten:
 - $5 + \text{true}$ bleibt $5 + \text{true}$
- 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;
```

finale Map: $[x \mapsto \text{Val.int } 7]$

```
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;
```

finale Map: $[x \mapsto \text{Val.int } 7]$

```
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: $[x \mapsto 2, y \mapsto 2, b \mapsto \text{true}, z \mapsto 4]$

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 \rightarrow 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?