Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN

Organisatorisches zum Projekt

Karlsruher Institut für Technologie

- Projektbeginn: 31.5.2016 (heute)
- Bearbeitung in Zweierteams
- Isabelle-Rahmen zu den Projekten sind vorgegeben
- Abgabe: 11.7.2016, 12.00 Uhr via Praktomat
- ab jetzt: freiwillige Dienstagstermine mit weiterführendem Material
  Wünsche dazu gerne an uns.
- bei Problemen frühzeitig melden!
- Projektpräsentation: 19.7.2016, 16.00 Uhr (SR 010)
- Infos dazu: 12.7.2016, 14.00 Uhr (Poolraum -143)

Teil XXI

Formale Semantik

Was ist Semantik?

Karlsruher Institut für Technologie

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

Was ist Semantik?

Karlsruher Institut für Technologie

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?

Karlsruher Institut für Technologie

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
- *Small-Step-Semantik*:
  Programm (= initiale Anweisung) + Startzustand wertet je einen Schritt zu Folgeprogramm + Folgezustand aus
  Syntax: $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$
  Anweisung c in Zustand σ wertet zu Anweisung c' und Zustand σ' aus
- *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 `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`» und «`Sub`» für ==, &&, $<$, $+$, $-$

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`» und «`Sub`» für ==, &&, $<$, $+$, $-$

Programmanweisungen:

- `Skip`
- Variablenzuweisung $x ::= e$
- sequentielle Komposition (Hintereinanderausführung) $c_1;; c_2$
- if-then-else `IF` (b) c_1 `ELSE` c_2
- while-Schleifen `WHILE` (b) c'

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`» und «`Sub`»
  für ==, &&, $<$, $+$, $-$

Programmanweisungen:

- `Skip`
- Variablenzuweisung $x ::= e$
- sequentielle Komposition (Hintereinanderausführung) $c_1 ;; c_2$
- if-then-else `IF` (b) c_1 `ELSE` c_2
- while-Schleifen `WHILE` (b) c'

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

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ

Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 `+ true`)

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \rightarrow \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \rightarrow \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

$$\langle \texttt{Skip};; c, \sigma\rangle \rightarrow \langle c, \sigma\rangle \qquad \frac{\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle}{\langle c;; c'', \sigma\rangle \rightarrow \langle c';; c'', \sigma'\rangle}$$

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ

Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \rightarrow \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

$$\langle \texttt{Skip};;c, \sigma\rangle \rightarrow \langle c, \sigma\rangle \qquad \frac{\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle}{\langle c;;c'', \sigma\rangle \rightarrow \langle c';;c'', \sigma'\rangle}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \rightarrow \langle c, \sigma'\rangle} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \rightarrow \langle c', \sigma'\rangle}$$

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ

Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \rightarrow \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

$$\langle \texttt{Skip};; c, \sigma\rangle \rightarrow \langle c, \sigma\rangle \qquad \frac{\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle}{\langle c;; c'', \sigma\rangle \rightarrow \langle c';; c'', \sigma'\rangle}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \rightarrow \langle c, \sigma'\rangle} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \rightarrow \langle c', \sigma'\rangle}$$

$$\langle \texttt{WHILE}\ (b)\ c, \sigma\rangle \rightarrow \langle \texttt{IF}\ (b)\ c;; \texttt{WHILE}\ (b)\ c\ \texttt{ELSE}\ \texttt{Skip}, \sigma\rangle$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma\rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma\rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma\rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma\rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'}$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma\rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma\rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c', \sigma\rangle \Rightarrow \sigma' \qquad \langle \texttt{WHILE}\ (b)\ c', \sigma'\rangle \Rightarrow \sigma''}{\langle \texttt{WHILE}\ (b)\ c', \sigma\rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{WHILE}\ (b)\ c', \sigma\rangle \Rightarrow \sigma}$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma \rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{WHILE}\ (b)\ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{WHILE}\ (b)\ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{WHILE}\ (b)\ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c;; c', \sigma \rangle \Rightarrow \sigma''}$$

Ausführungsdauer

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

Konstanten:	1
Variablen:	1
je bin. Operation:	1
Skip:	1
LAss:	1 + Auswertung des arith. Ausdrucks
If:	1 + Auswertung des bool. Ausdrucks + Dauer des gew. Zweigs
While-False:	1 + Auswertung des bool. Ausdrucks
While-True:	1 + Auswertung des bool. Ausdrucks + Dauer für Rumpf + Rest-Dauer

Formalisierung in Isabelle

Siehe Rahmen

Teil XXII

Typsystem

Typsystem

Typsystem ordnet jedem Ausdruck Typ zu

Zwei Typen: `Boolean` und `Integer`

Typumgebung Γ: Map von Variablen nach Typ

Zwei Stufen:

1. jeder Ausdruck e bekommt unter Typumgebung Γ Typ T zugeordnet
   Syntax: $\Gamma \vdash e : T$
2. Anweisung c ist typbar unter Typumgebung Γ
   Syntax: $\Gamma \vdash c$

auch Typsystem definiert als induktive Menge

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 «*Less*»: Unterausdrücke *Integer*, ganzer Operator *Boolean*

Regeln

Karlsruher Institut für Technologie

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 «*Less*»: Unterausdrücke *Integer*, ganzer Operator *Boolean*

Anweisungen:

- `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 *Boolean*

Formalisierung in Isabelle

Siehe Rahmen

Teil XXIII

Projekt: Konstantenfaltung und -propagation

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
- wir beschränken uns auf Konstantenfaltung und -propagation rein auf Semantikebene

Konstantenfaltung

Optimierung für Ausdrücke

- Wenn Berechnungen nur auf Konstanten, Ergebnis einfach einsetzen:
  `Val(Intg 5) «Add» Val(Intg 3)` wird zu `Val(Intg 8)`
  `Val(Intg 4) «Eq» Val(Intg 7)` wird zu `Val false`
- Wenn mind. eine Variable, einfach beibehalten:
  `Var y «Sub» Val(Intg 3)` bleibt `Var y «Sub» Val(Intg 3)`
- nicht sinvolle Ausdrücke auch beibehalten:
  `Val(Intg 5) «And» Val true` bleibt `Val(Intg 5) «And» Val true`
- Wenn Ausdruck nur Konstante oder Variable, auch beibehalten:
  `Val(Intg 5)` bleibt `Val(Intg 5)`, `Var y` bleibt `Var 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

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$\texttt{IF}\ (\texttt{Var}\ x \ «Eq»\ \texttt{Var}\ y)$
$\quad (y ::= \texttt{Var}\ x \ «Add»\ \texttt{Val(Intg 2)})$
$\texttt{ELSE}\ (y ::= \texttt{Var}\ x \ «Sub»\ \texttt{Var}\ z);;$
$z ::= \texttt{Var}\ y$

wird zu

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Var}\ x;;$
$b ::= \texttt{Var}\ x \ «Eq»\ \texttt{Var}\ y;;$
$\texttt{IF}\ (\texttt{Var}\ b)$
$\quad (z ::= \texttt{Var}\ x \ «Add»\ \texttt{Var}\ y)$
$\texttt{ELSE}\ (z ::= \texttt{Var}\ x)$

wird zu

Beispiele

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$\texttt{IF}\ (\texttt{Var}\ x\ «Eq»\ \texttt{Var}\ y)$
$\quad (y ::= \texttt{Var}\ x\ «Add»\ \texttt{Val(Intg 2)})$
$\texttt{ELSE}\ (y ::= \texttt{Var}\ x\ «Sub»\ \texttt{Var}\ z);;$
$z ::= \texttt{Var}\ y$

wird zu

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$y ::= \texttt{Val(Intg 7)}\ «Sub»\ \texttt{Var}\ z;;$
$z ::= \texttt{Var}\ y$

finale Map: $(x \mapsto \texttt{Val(Intg 7)})$

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Var}\ x;;$
$b ::= \texttt{Var}\ x\ «Eq»\ \texttt{Var}\ y;;$
$\texttt{IF}\ (\texttt{Var}\ b)$
$\quad (z ::= \texttt{Var}\ x\ «Add»\ \texttt{Var}\ y)$
$\texttt{ELSE}\ (z ::= \texttt{Var}\ x)$

wird zu

Beispiele

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$\mathrm{IF}\ (\texttt{Var}\ x \ll\!Eq\!\gg \texttt{Var}\ y)$
$\quad (y ::= \texttt{Var}\ x \ll\!Add\!\gg \texttt{Val(Intg 2)})$
$\mathrm{ELSE}\ (y ::= \texttt{Var}\ x \ll\!Sub\!\gg \texttt{Var}\ z);;$
$z ::= \texttt{Var}\ y$

wird zu

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$y ::= \texttt{Val(Intg 7)} \ll\!Sub\!\gg \texttt{Var}\ z;;$
$z ::= \texttt{Var}\ y$

finale Map: $(x \mapsto \texttt{Val(Intg 7)})$

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Var}\ x;;$
$b ::= \texttt{Var}\ x \ll\!Eq\!\gg \texttt{Var}\ y;;$
$\mathrm{IF}\ (\texttt{Var}\ b)$
$\quad (z ::= \texttt{Var}\ x \ll\!Add\!\gg \texttt{Var}\ y)$
$\mathrm{ELSE}\ (z ::= \texttt{Var}\ x)$

wird zu

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Val(Intg 2)};;$
$b ::= \texttt{Val true};;$
$z ::= \texttt{Val(Intg 4)}$

finale Map: $(x \mapsto \texttt{Val(Intg 2)},$
$y \mapsto \texttt{Val(Intg 2)},\ b \mapsto \texttt{Val true},$
$z \mapsto \texttt{Val(Intg 4)})$

`WHILE`

Wie `IF` könnte man auch `WHILE` vereinfachen:

- falls Prädikat konstant *`false`*, komplettes `WHILE` durch `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 `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 ::= Val(Intg 5);; y ::= Val(Intg 1);;
WHILE (Var x «Less» Val(Intg 7))
  (IF (Var y «Eq» Val(Intg 4))
    (x ::= Val(Intg 9))
   ELSE Skip;;
   y ::= Var y «Add» Val(Intg 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, verwenden also *`empty`* Map
- 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: “Endzustand ist der Gleiche, egal ob man im gleichen Anfangszustand Originalanweisung oder resultierende Anweisung der Konstantenpropagation verwendet”
4. Beweis, dass sich die Ausführungsgeschwindigkeit durch Konstantenpropagation erhöht
5. Beweis, dass zwei-/mehrfache Anwendung der Konstantenpropagation das Programm nicht weiter verändert
6. Beweis, dass Konstantenpropagation Typisierung erhält anders gesagt: “Wenn Originalanweisung typt, dann auch resultierende Anweisung der Konstantenpropagation”

Beweise sollten verständlich und (komplett) in Isar geschrieben werden

Hinweise

- Isabelle-Dokumentation verwenden! Vor allem die Tutorials zu Isabelle/HOL, Isar und Function Definitions sollten helfen
- erst formalisieren, dann beweisen!
  Beispiele mittels *`value`* prüfen (z.B. Beispielprogramme in Semantics.thy)
- verwendet *`quickcheck`*, *`nitpick`* oder *`sledgehammer`* um Aussagen *vor* einem Beweis zu prüfen (spart oft unnötige Arbeit)
- falls Funktionsdefinitionen mit *`fun`* nicht funktionieren:
  - oftmals Probleme mit Termination
  - Fehlermeldung genau ansehen (wo Probleme mit Termination?)
    oft hilft eigene [simp] Regel
  - auch möglich: Zu *`function`* übergehen und versuchen, Termination explizit zu zeigen (siehe Tutorial zu Function Definitions)
- für die Beweise überlegen: welche Beziehungen müssen zwischen Semantikzustand, Typumgebung und Konstantenmap existieren?

Hinweise II

- *case*-Ausdrücke statt *if-then-else* verwenden wo möglich
  $\Rightarrow$ Entsprechende *split*-Regeln verwenden
  $\Rightarrow$ Mehr Automatismus

Beispiel

lemma "case v of None $\Rightarrow$ f 0 | Some x $\Rightarrow$ f x $\Longrightarrow$ $\exists$n. f n"
  by (cases v) auto

Hinweise II

- *case*-Ausdrücke statt *if-then-else* verwenden wo möglich
  $\Rightarrow$ Entsprechende *split*-Regeln verwenden
  $\Rightarrow$ Mehr Automatismus

Beispiel

lemma *"case v of None ⇒ f 0 | Some x ⇒ f x ⟹ ∃n. f n"*
  by *(auto split: option.splits)*

Teil XXIV

Projekt: Eulerkreise

Hintergrund

Definition:
Ein Eulerkreis eines Graphen ist ein geschlossener Pfad, der jede Kante eines Graphen genau einmal enthält.

Satz:
Ein nicht-leerer gerichteter Multi-Graph hat genau dann einen Eulerkreis, wenn er stark zusammenhängend ist und für jeden Knoten Eingangsgrad und Ausgangsgrad übereinstimmen.

Hintergrund

Definition:
Ein Eulerkreis eines Graphen ist ein geschlossener Pfad, der jede Kante eines Graphen genau einmal enthält.

Satz:
Ein nicht-leerer gerichteter Multi-Graph hat genau dann einen Eulerkreis, wenn er stark zusammenhängend ist und für jeden Knoten Eingangsgrad und Ausgangsgrad übereinstimmen.

Das wollen wir formalisieren.

Multi-Mengen

Isabelle bietet (in der Theorie `"~~/src/HOL/Library/Multiset"`) einen Datentypen `'a multiset` für Multisets, also für Mengen mit Vielfachheiten. Ein paar Schreibweisen:

`{#}`	für die leere Multi-Menge
`{# x #}`	für Multi-Menge die nur `x` einmal enthält.
`size M`	für die Anzahl der Einträge in `M`.
`set_mset M`	um aus einer Multi-Menge eine Menge zu machen.
`x ∈# M`	wenn `x` mind. einmal in `M` vorkommt.
`M + M'`	für die Vereinigung von zwei Multimengen.
`M ⊆# M'`	wenn eine Menge eine Teil-Multi-Menge einer anderen ist.
`{# x ∈# M . P x #}`	für eine Filter-Operation auf Multi-Mengen.
`mset xs`	um aus Listen Multi-Mengen zu machen.

Unsere Modellierung von Graphen

Eine Kante ist ein geordnetes Paar, und ein Graph ist eine Multi-Menge solcher Paare:

type_synonym *'node graph = "('node × 'node) multiset"*

Unsere Modellierung von Graphen

Eine Kante ist ein geordnetes Paar, und ein Graph ist eine Multi-Menge solcher Paare:

type_synonym *'node graph = "('node × 'node) multiset"*

Ein paar Eigenschaften von Graphen:

definition *inDegree :: "'node graph ⇒ 'node ⇒ nat"*
  where *"inDegree G n = size {# e ∈# G . snd e = n #}"*

definition *outDegree :: "'node graph ⇒ 'node ⇒ nat"*
  where *"outDegree G n = size {# e ∈# G . fst e = n #}"*

definition *nodes :: "'node graph ⇒ 'node set"*
  where *"nodes G = fst ‘ (set_mset G) ∪ snd ‘ (set_mset G)"*

Unsere Modellierung von Wegen

Ein Weg ist eine nicht-leere Liste von Kanten ("('node × 'node) list"), so dass der Zielknoten einer Kante der Anfangsknoten der nächsten Kante ist:

inductive *path :: "'node graph ⇒ ('node × 'node) list ⇒ bool"*
  where *"..."*

Unsere Modellierung von Wegen

Ein Weg ist eine nicht-leere Liste von Kanten (*"('node × 'node) list"*), so dass der Zielknoten einer Kante der Anfangsknoten der nächsten Kante ist:

inductive *path :: "'node graph ⇒ ('node × 'node) list ⇒ bool"*
  where *"..."*

Ein Graph ist stark zusammenhängend, wenn es zu je zwei Knoten v_1, v_2 im Graphen einen Pfad gibt, dessen erste Kante bei v_1 beginnt und dessen letzte Kante bei v_2 endet.

definition *strongly_connected :: "'node graph ⇒ bool"*
  where *"strongly_connected G = ..."*

Unsere Modellierung von Wegen

Ein Weg ist eine nicht-leere Liste von Kanten (*"('node × 'node) list"*), so dass der Zielknoten einer Kante der Anfangsknoten der nächsten Kante ist:

inductive *path :: "'node graph ⇒ ('node × 'node) list ⇒ bool"*
  where *"..."*

Ein Graph ist stark zusammenhängend, wenn es zu je zwei Knoten v_1, v_2 im Graphen einen Pfad gibt, dessen erste Kante bei v_1 beginnt und dessen letzte Kante bei v_2 endet.

definition *strongly_connected :: "'node graph ⇒ bool"*
  where *"strongly_connected G = ..."*

Ein Eulerkreis eines Graphen ist ein geschlossener Pfad, der jede Kante eines Graphen genau einmal enthält:

definition *eulerian :: "'node graph ⇒ bool"* **where** *"eulerian G =
 (∃ es. path G es ∧ fst (hd es) = snd (last es) ∧ G = mset es)"*

Die Aufgabe

Beweisen Sie:

theorem *eulerian_characterization:*
  "$G \neq \{\#\} \Longrightarrow$ *eulerian G* $\longleftrightarrow$
    strongly_connected G $\wedge$ ($\forall$ *v. inDegree G v = outDegree G v*)"

Die Aufgabe

Beweisen Sie:

theorem *eulerian_characterization:*
  "$G \neq \{\#\} \Longrightarrow$ *eulerian* $G \longleftrightarrow$
    strongly_connected $G \wedge (\forall v.$ *inDegree* $G\ v =$ *outDegree* $G\ v)$"

Ein paar Hinweise:

- Die Richtung $\longrightarrow$ ist (scheinbar) leichter.
- Ist *es* ein Pfad, so ist *mset es* ein Graph; so kann man *inDegree* etc. auch damit verwenden.
- Aussagen über Ein- und Ausgangsgrade in Pfaden sind schöner, wenn der Pfad (ggf. künstlich) geschlossen wird!
- Bei komischen Induktionen (z.B. um Schrittweise von *23* auf *42* zu schließen) ist es schön, die Induktionsregel, sofern nicht in der Standardbibliothek vorhanden, als eigenes Lemma zu extrahieren und dann zu benutzen.