A

Aufgaben

Dieser Anhang enthält die Aufgaben, die im Kurs gestellt wurden.

A.1 Regelbasierte Systeme – überall (ohne Bewertung)

1. Nennen Sie alle – oder doch möglichst viele – Fälle, in denen Sie mit informationsverarbeitenden Systemen zu tun hatten, deren Zustände und/oder Verhalten mit Regeln spezifiziert war.
   Klassifizieren Sie die Regeln:
   a) Welche Art von *Objekten* beschrieben die Zustände?
   b) Welcher Art waren die Regeln?
   c) Handelte es sich um eine Grammatik oder um ein Ersetzungssystem?
2. Illustrieren Sie mindestens eines der unter 1 genannten Systeme mit einem Beispiel.

A.2 Urnen-Orakel und Hüllen (zum 1. 11. 2010)

1. Griechische Urnen (nach Dershowitz und Plaisted [DP01]).
   Eine Urne enth"alt 150 schwarze und 75 wei"se Bohnen. Es werden immer zwei Bohnen zugleich entnommen; sind sie beide schwarz oder weiß, wird eine schwarze in die Urne getan, sonst die wei"se. (Man muss also noch bis zu 37 schwarze Bohnen neben der Urne haben.)
   Definieren Sie ein Reduktionssystem, das dieses Spiel realisiert: Eine Menge, die die Objekte repräsentiert, und eine Reduktionsrelation auf dieser Mege, die die Spielregeln nachbildet.
   Ist die Farbe der letzten Bohne in der Urne determiniert (also vorher bestimmt), und – wenn ja – welche ist es?
2. Welche Abschlussoperationen sind kommutativ:

a) Gleicht die reflexive Hülle der transititven Hülle einer Relation der transitiven Hülle ihrer reflexiven Hülle? Das heißt, gilt $(\to^{+})^{=} = (\to^{=})^{+}$, und entspricht dies der transitiv-reflexiven Hülle $\to^{*}$?
b) Wie ist das mit transitiver und symmetrischer Hülle? Das heißt, stimmen $(\leftarrow \cup \to)^{+}$ und $\leftarrow^{+} \cup \to^{+}$ überein?

A.3 Wort-Grammatiken für Namen (zum 8. 11. 2010)

Namen sind Ausdrücke, deren Ergebnis eine Variable ist und also auf der linken Seite einer Zuweisung stehen dürfen (*lvalue* in C).

Definition A.1 (Namen). Die Menge aller *Namen* sei wie folgt definiert:

1. Wenn x ein anderswo vereinbarter Name für eine Variable vom Typ t ist, so ist x auch ein Name vom Typ t.
2. Wenn a der Name einer anderswo vereinbarten Feldvariable vom Typ $()t$ ist und e ein ein ganzzahliger Ausdruck, dann ist der Feldzugriff $a(e)$ ein Name vom Typ t.
3. Wenn s eine anderswo vereinbarte Strukturvariable vom Typ
$$\textbf{struct}\{t_1 f_1; \ldots; t_k f_k\};$$
ist (mit $k \geqslant 1$), ist für jeden der Feldselektor-Bezeichner f_i $(1 \leqslant i \leqslant k)$ auch $s.f_i$ ein Name vom Typ t_i.
4. Wenn p eine anderswo vereinbarte Zeigervariable vom Typ t^* ist, so ist p^* ein Name vom Typ t.

Definieren Sie Wortgrammatiken für die oben beschriebenen Namen.

1. Definieren Sie ein kontextfreie Grammatik, die die unterliegende kontextfreie Struktur von Namen beschreibt. Dabei können sie weder die Eigenschaft, dass die Variablen anderswo vereinbart sind, noch die Typregeln berücksichtigen.
   Weshalb?
2. Definieren Sie eine van-Wijngaarden-Grammatik für die vollständige Syntax von Namen.
   Nehmen Sie dabei der Einfachheit halber an, die Grammatik habe eine Startregel $\langle z \rangle \to \langle \textit{name in} \ldots \rangle$, wobei im Wort "…" aus Meta-Terminalen die Vereinbarungen der Variablen repräsentiert sind, die für den Namen gelten sollen.
   Dabei kann "…" ein *Environment* sein ähnlich wie in Beispiel 3.8, nur dass das Meta-Nichtterminal auch zusammengesetzte Typen repräsentieren muss.
   (In der vollständigen Grammatik einer Programmiersprache würde "…" bei der Erzeugung von Variablenvereinbarungen generiert.)

A.4 Bedingte Ausdrücke (zum 15. 11. 2010)

Betrachten Sie folgende Metaregeln, die monomorphe algebraische Datentypen definieren:

$$
\begin{aligned}
E &::= \Box \mid X\ of\ T\ E \mid T\ of\ S\ E \\
T &::= type\ bool \mid type\ int \mid type\ intlist \mid \dots \\
S &::= P \mid P\ plus\ S \\
P &::= K\ Ts \\
Ts &::= \Box \mid T\ Ts \\
X &::= id\ a \mid id\ x \mid \dots \\
K &::= con\ true \mid con\ false \mid con\ nil \mid con\ cons \mid con\ leaf \mid con\ branch \mid \dots
\end{aligned}
$$

Umgebungen E repräsentieren Sequenzen von Definitionen: Variablenbezeichnern X wird ein Typbezeichner T zugeordnet, bzw. Typbezeichnern (außer den vordefinierten Typbezeichnern *type bool* und *type int*) werden Summentypen S zugeordnet. Summentypen S sind nicht leere Listen von Produkttypen P, die jeweil aus Konstruktoren K bestehen, und wiederum auf Sequenzen Ts von Typbezeichnern angewendet werden.

Ein Typ für Listen ganzer Zahlen könnten darin definiert sein als

$$type\ inlist\ of\ con\ nil\ plus\ con\ cons\ type\ int\ type\ intlist.$$

Verallgemeinern Sie die van-Wijngaarden-Grammatik aus Beispiel 3.3 auf Ausdrücke von monomorphen algebraischen Typen, und fügen Sie *Muster* und *Fallunterscheidungen* hinzu:

1. Ein Muster (*pattern*) des Typs t (repräsentiert als Typbezeichner T) ist Ausdruck über Konstruktorbezeichnern und Wertbezeichnern:
   a) Zahlen ($\langle number \rangle$) sind Muster vom Typ *type int*.
   b) Boole'sche Werte (`false` und `true`) sind Muster vom Typ *type bool*.
   c) Ein Wertbezeichner x (repräsentiert durch das Metanon-Nichtterminal X) ist ein Muster des erwarteten Typs T. Der Wertbezeichner wird *gebunden*, d.h., neu in die Umgebung eingefügt.
   d) Wenn k ein Konstruktorbezeichner ist (repräsentiert vom Metan-Nichtterminal K) und $p_1, \dots, p_k$ Muster der Typen $T_1, \dots, T_k$ (repräsentiert durch das Metanichtterminal T), dann ist $k\, p_1 \dots p_k$ ein Muster vom Typ T, wenn T als ein Summentyp S definiert ist, der den Produkttypen $k\, T_1 \dots T_k$ enthält .
   e) Wenn p ein Muster vom Typ T ist, ist auch das geklammerte Muster (p) ein Muster vom Typ T.
2. Erweitern Sie Ausdrücke um folgende Regeln:
   a) Wenn k ein Konstruktorbezeichner ist und $e_1, \dots, e_k$ Ausdrücke der Typen $T_1, \dots, T_k$ (repräsentiert durch das Metan-Nichtterminal T), dann ist $k\, e_1 \dots e_k$ ein Ausdruck vom Typ T, wenn T als ein Summentyp S definiert ist, der den Produkttypen $k\, T_1 \dots T_k$ enthält.

b) Wenn e ein Ausdruck und $p_1, \dots, p_k$ Muster vom Typ T' sind und $e_1, \dots, e_k$ Ausdrücke vom Typ T, so ist der Ausdruck

$$\textbf{case } e \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$$

ein Auswahlausdruck vom Typ T.

A.5 Terminierend? Konfluent? (zum 1. 12. 2010)

Betrachten Sie die Funktionssymbole $\Sigma_0 = \{\mathit{false}, \mathit{true}\}$, $\Sigma_1 = \{\neg\}$ und $\Sigma_2 = \{\wedge, \vee\}$, die darüber konstruierten Termen und die folgenden Termersetzungsregeln über diesen Termen, die durch Umschreiben einer Gleichungsdefinition für Wahrheitswerte entstanden ist:

$$
\begin{aligned}
\rho_1 &: \neg \mathit{true} \to \mathit{false} \\
\rho_2 &: \neg(\neg x) \to x \\
\rho_3 &: \mathit{true} \wedge x \to x \\
\rho_4 &: \mathit{false} \wedge x \to \mathit{false} \\
\rho_5 &: \neg \mathit{false} \to \mathit{true} \\
\rho_6 &: x \vee y \to \neg(\neg x \wedge \neg y)
\end{aligned}
$$

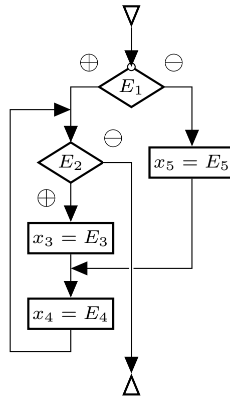
1. Definieren diese Regeln terminierende Ersetzungen?
   a) Finden Sie Argumente dafür, dass die Ersetzungen alle Funktionssymbole $\neg$, $\wedge$ und $\vee$ "verschwinden lassen".
   b) Definieren Sie eine lexikographische Pfadordnung, die mit den Regeln ρ_1 bis ρ_5 verträglich ist. (Siehe Satz 4.20 im Script.)
   c) Finden Sie ein Argument, weshalb dann auch Regel ρ_6 mit den anderen Regeln zusammend terminierend ist.
2. Sind die Regeln linkslinear?
3. Sind die Regeln nicht-überlappend?
4. Falls einige Regeln einander überlappen, geben Sie deren kritische Paare an und überprüfen Sie, ob diese Paare konvergent sind, d.h., ob sie sich mit den Regeln zu einem Term reduzieren lassen.
5. Definieren diese Regeln also insgesamt ein terminierendes und konfluentes Termersetzungssystem? Oder wie? Oder was?

A.6 Unstrukturierte Flussdiagramme (zum 7. 12. 2010)

Befehle (*commands*) haben in maschinennahen Programmen, z.B. in der Originalfassung von Fortran, die folgende Form:

$$\begin{aligned} C ::= {} & x = E \\ \mid {} & \mathbf{goto}\ \ell \\ \mid {} & \mathbf{if}\ E\ \mathbf{then}\ \mathbf{goto}\ \ell \\ \mid {} & \ell : C \\ \mid {} & \ell : \qquad\qquad \text{(Marke am Programmende)} \\ \mid {} & C;\ C \end{aligned}$$

1. Definieren Sie eine kontextuelle Sterngrammatik, welche die – im Allgemeinen unstrukturierten – Flussdiagramme für diese Art von Befehlen generiert.
   In der Graphdarstellung stellen Knoten Programmstellen dar, so dass im Graph jede Programmstelle schon implizit markiert ist.
2. Geben Sie eine Ableitung des folgenden Flussdiagramms an:



3. Gibt es in jedem Flussdiagramm einen Pfad vom Start zum Stopp? Wenn nicht, geben Sie ein Beispiel!
4. Geben Sie Argumente dafür, weshalb diese Sprache nicht mit einer Sterngrammatik erzeugt werden kann.

A.7 Programmgraph-Bedingungen (zum 21. 12. 2010)

Betrachten Sie die Bedingungen an Programmgraphen in dem Artikel [Hof10].[1]

Definieren Sie die Eigenschaften P_1 bis P_9 als rekursive Graphbedingugnen, soweit dies nicht schon in der Vorlesung geschehen ist.

Dabei dürfen Sie die Sternregeln $\mathcal{Q}$ in Abbildung A.1 benutzen. (Und Sie dürfen einfach "+" statt $+^{x,z}$ schreiben.)

$$\mathcal{Q} = \left\{ (x) \rightarrow [+^{x,z}] \rightarrow (z) ::= (x) \xrightarrow{a} (z) \;\middle|\; (x) \xrightarrow{b} (y) \rightarrow [+^{y,z}] \rightarrow (z) \;\middle|\; x, y, z \in \dot{\Sigma}, a, b \in \bar{\Sigma}, \right\}$$

Fig. A.1. The generic star rules defining the path query +

Hier die zu realisierenden Bedingungen aus [Hof10, Definition]:

Definition A.2 (Program Graph). A graph G is a *program graph* if it has the following properties:

P_1. ... (entfällt)
P_2. ... (entfällt)
P_3. The subgraph $\bar{G}$ induced by $\longrightarrow$-edges of G is a spanning tree of G; the root of $\bar{G}$ is a class.
P_4. If an expression refers to a method m, m must be contained in some class of the graph.
(Diese Bedingung haben wir in der Vorlesung behandelt, siehe unten)
P_5. If an expression e accesses a variable v contained in a class c, e must be a sub-expression of a body b that is contained in a sub-class of c.
P_6. If an expression e accesses a parameter p of a method m, e must be a sub-expression of a body that implements m.
P_7. If a method body b implements a method signature m, b must be contained in a sub-class of the class c containing m.
P_8. For every method signature m, every class contains at most one body implementing m.
P_9. ... (entfällt)

The class of program graphs is denoted by Π.

Example A.3 (A Program Graph). Program graphs are typed over the type graph Σ shown in **??**. The nodes of a program graph represent syntactic entities of a program: classes, variables, method signatures, method bodies, and expressions. Edges establish relations between these entities. Those drawn as straight arrows "$\longrightarrow$" represent "*part of*" relations, corresponding to the

[1] Die ersten 4 Seiten dieses Artikels wurden vor zwei Wochen als Kopie verteilt, zusammen mit dem Artikel [HM10].

```
class Cell is
  var cts: Any;
  method get() Any is
    return cts;
  method set(var n: Any) is
    cts := n
subclass ReCell of Cell is
  var backup: Any;
  method restore() is
    cts := backup;
  override set(var n: Any) is
    backup := cts;
    super.set(n)
```

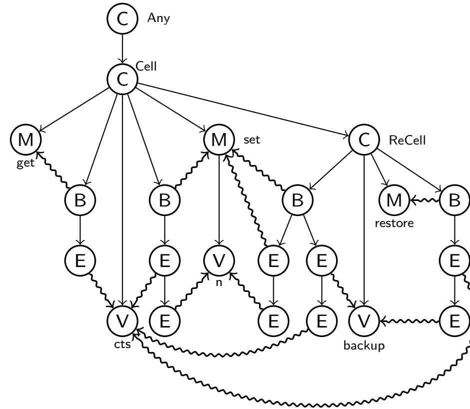


Fig. A.2. A program (on the left) and its program graph (on the right)

hierarchical structure of programs (their abstract syntax): classes consist of subclasses, and of declarations of variables, method signatures and method bodies; method signatures have variables representing their formal parameters, method bodies consist of expressions; expressions may have expressions as arguments.

Edges drawn as winding arrows "⇝" represent "*references*", of method bodies and expressions to declarations of method signatures and variables. A method body *implements* a method signature. An expression represents just data flow: it may *call* a method, *use* the value of a variable, or *update* it with the value of an expression.

Figure A.2 shows a simple object-oriented program from [AC96] and its representation as a program graph. When drawing program graphs, we abbreviate node labels by their initial letters, and omit edge labels altogether. The source and targets of edges, and the style in which they are drawn will make clear which type of edge is meant.

Nodes of type Class are called "class nodes" or just "classes", and so forth for the other nodes. We say that a class c' is a *super-class* of a class c if either c' equals c, of if some subclass c'' of c' is a super-class of c. In a similar way, we talk of the *sub-expression* of a body or expression.

In a program graph, every entity is represented by a unique node so that names of classes, variables, methods and parameters are not relevant in the graph. (In Figure A.2, names are only ascribed to these nodes in order to show the correspondence between the program text and the graph.)

Teile der Lösung

Bedingung P_3 für Programmgraphen kann so definiert werden:

$$\bigwedge_{\ell,\ell',\ell''\in\dot{\Sigma}} \forall\left(\textcircled{\ell}\!\to\!\textcircled{\ell'}, \nexists\, \textcircled{\ell}\!\to\!\textcircled{\ell'}\!\leftarrow\!\textcircled{\ell''}\right)$$

$$\exists\left(\textcircled{C}, \bigwedge_{\ell\in\dot{\Sigma}} \nexists\, \textcircled{\ell}\!\to\!\textcircled{C} \wedge \left(\forall\left(\textcircled{C}\quad \textcircled{\ell}, \exists\, \textcircled{C}\!\to\!\boxed{+}\!\to\!\textcircled{\ell}\right)\right)\right)$$

Fig. A.3. Constraints of program graphs

Example A.4 (Constraints for Program Graphs). For defining properties of program graphs, we use recursive graph conditions over a family $Q = \{+^{x,y} \mid x, y \in \dot{\Sigma}\}$ of path queries defined by the rules $\mathcal{Q}$ shown in Figure A.1. The constraint shown in Figure A.3 specifies Property P_3 of **??** of program graphs. (In examples, we drop the indices of queries $+$ as they can be inferred from the context.) Note that the constraint (4) is equivalent to the infinite nested constraint shown in **??**.

In these conditions, we use a shorthand notation for morphisms. Instead of drawing the morphisms completely (with domain, name, and codomain)

$$\bigwedge_{\ell,\ell',\ell''\in\dot{\Sigma}} \forall\left(\{\} \overset{a}{\twoheadrightarrow} C, \nexists\, C \overset{c}{\twoheadrightarrow} C'\right)$$

we just draw their codomains

$$\bigwedge_{\ell,\ell',\ell''\in\dot{\Sigma}} \forall\,(C, \nexists\, C')$$

since the way the morphisms map nodes and edges is unique in this case.

Tipp: Überprüfen Sie anhand des Graphen in Figure A.2, ob die Bedingungen wie gewünscht funktionieren.

A.8 Refaktorierung (zum 19. 1. 2011)

Die *Refactoring*-Operation *PushDownMethod* aus Fowler's Katalog tut (annähernd) das Gegenteil der Operation *PullUpMethod*, die in der Vorlesung besprochen wurde:

1. Sei c eine Klasse, in der eine Implementierung b für eine Methode mit der Signatur m definiert wird.
2. Lösche b in c, und füge Kopien von b in allen direkten Unterklassen $c_1, \dots, c_n$ von c ein (wobei $n \geqslant 1$).
3. Dabei darf keine der Unterklassen $c_1, \dots, c_n$ die Implementierung von m überschreiben, und keine der Oberklassen von c darf bereits eine Implementierung von m enthalten.

Definieren Sie die Operation *PushDownMethod* als eine Graphtransformationsregel mit Graphvariablen und mehrfachen Teilgraphen.

- Legen Sie die Programmgraphen zugrunde, die als Darstellung für objektorientierte Programme in der Vorlesung eingeführt wurde.
- Sie sollten zunächst das Definieren der Bedingungen in Punkt 3 außer Acht lassen.
  Sie können sie nachträglich als rekursive Graphbedngungen an das Muster der Regel formulieren.
- Weshalb braucht man die Bedingungen in Punkt 3, um *behaviour preservation* sicherzustellen, d. h., dass das Programm hinterher dasselbe tut wie vorher? (Skizzieren Sie Beispiele.)
- Welche Bedingungen müssen die Klassen c_i ($1 \leqslant i \leqslant n$) noch erfüllen, damit die Implementierung b dort eingeführt werden kann, ohne das Verhalten des Programms zu verändern? (Denken Sie an die in b aufgerufenen Methoden, die dynamisch gebiunden werden.)
- Wie müssten Sie in der Regel das Muster (und analog die Ersetzung) erweitern, damit das Muster aus dem **Cls**-Stern ableitbar und *hinreichend allgemein* ist, so dass die Regel in allen möglichen Situationen angewendet werden kann? (*Hierfür gibt es 2 Zusatzpunkte.*)

B

Lösungen von Aufgaben

Dieses Kapitel enthält Lösungen für einige der im Text gestellten Aufgaben.

B.1 Bekannte Regel-basierte Systeme

1. Folgende regelbasierte Systeme könnten Studierenden schon begegnet sein:
   - Axiome und Gleichungen (in *Mathematik für Informatiker*) sind quantifizierte Regelschemata über mathematischen Formeln.
   - Chomsky-Grammatiken, endliche Automaten, Kellerautomaten, Turingmaschinen (in *Theoretische Informatik* oder *Übersetzer*) sind Wortgrammatiken bzw. Reduktionssysteme auf Worten und "Konfigurationen", die mit Regeln (ohne Variablen) definiert werden.
   - Haskell-Programme (in *Praktische Informatik 3*) sind applikative Termersetzungssysteme, deren Regeln Platzhalter für Werte und Funktionen (spezielle Werte) enthalten.
   - Prolog-Programme (in *Praktische Informatik 3* oder *Künstliche Intelligenz*) definieren eine Datewbasis von Klauseln (Schlussregeln) und Fakten, die Anfragen mit Unifikation und Resolution auswerten.
   - Protokolle (in *Technische Informatik* oder *Rechnernetze*) sind Sprachen über einem Vokabular, die zur Kommunikatioin zwischen Rechnern dienen.
   - Logische Inferenzregeln (in *Logik*) sind Metaregeln zur Transformation von logischen Formeln, die die Gültigkeit (oder Erfüllbarkeit) der Formeln erhalten.
   - Collage-Grammatiken (in *Methoden der Bilderzeugung*) definieren Grammatiken von Bildern auf der Basis von Ersetzungsregeln auf Graphen.
   - ...
2. —
3. —

B.2 Urnen-Orakel und Hüllen

Urnen-Orakel.

Am besten sieht man, was dieses Regelsystem tut, wenn man als Objekte nicht Zeichenketten über W und S wählt, sondern Paare $\langle w, s\rangle$ von natürlichen Zahlen, wobei w die Anzahl der weißen und s die Anzahl der schwarzen Bohnen in der Urne darstellt. Dann könne die vier Originalregeln mit folgenden zwei Regeln modelliert werden:

$$\begin{array}{ll} \langle w, s\rangle \to \langle w-2, s+1\rangle & \text{Regel 1} \\ \langle w, s\rangle \to \langle w, s-1\rangle & \text{Regel 2-4} \end{array}$$

Nun ist es klar, das es bei einem Anfangszustand $\langle 75, 150\rangle$ nicht möglich ist, alle weißen Bohnen herauszunehmen, weil die Anzahl der weißen Bohnen immer ungerade bleibt. Also bleibt am Ende immer eine weiße Bohne zurück.

Hätte man aber mit einer geraden Anzahl weißer Bohnen begonnen, bliebe immer eine schwarze Bohne zurück.

Hüllen.

"Kommutatiuv" ist zwar eine seltsame Bezeichnung für die gefragten Eigenschaften ("gleich" oder "äquivalent" wäre sohl zutreffender), aber egal:

Lemma B.1.

$$(\to^{+})^{=} = (\to^{=})^{+} = \to^{*}$$

Proof. 1. Die Eigenschaft $\to^{*} = (\to^{+})^{=}$ folgt einfach aus den Definitionen für Abschlüsse:

$$\begin{array}{rll} \xrightarrow{*} & = \xrightarrow{+} \cup \xrightarrow{0} & \text{Definition 2.2.5} \\ & = (\xrightarrow{+}) \cup \xrightarrow{0}{}^{=} & \text{Definition 2.2.2} \end{array}$$

2. $((\to^{+})^{=} \subseteq (\to^{=})^{+}.)$ Wenn $a \xrightarrow{+}{}^{=} b$, so ist nach Definition von $\xrightarrow{=}$ entweder $a = b$ und damit auch $a \xrightarrow{=} b$ und $a \xrightarrow{=}{}^{+} b$, oder aber

$$a = a_0 \to a_1 \to \cdots \to a_n = b, \quad (n \geqslant 1).$$

Dann gilt nach Definition 2.2.2 auch

$$a = a_0 \xrightarrow{=} a_1 \xrightarrow{=} \cdots \xrightarrow{=} a_n = b$$

und also auch $a \xrightarrow{=}{}^{+} b$.

3. $((\to^{=})^{+} \subseteq (\to^{+})^{=}.)$ Wir zeigen die Eigenschaft durch Induktion über die Länge der Reduktionssequenzen in

$$a = a_0 \xrightarrow{=} a_1 \xrightarrow{=} \cdots \xrightarrow{=} a_n = b.$$

Für $n = 1$ folgt aus $a \overset{=}{\rightarrow} b$ unmittelbar $a \overset{+=}{\rightarrow} b$. Nehmen wir an, dass $\overset{=}{\rightarrow}^n = \overset{n=}{\rightarrow}$ für irgendein $n \geqslant 1$ gilt, und betrachten wir eine Sequenz der Länge $n + 1$:

$$a_0 \overset{=}{\rightarrow} a_1 \overset{=}{\rightarrow} \cdots \overset{=}{\rightarrow} a_n \overset{=}{\rightarrow} a_{n+1}.$$

Wenn $a_n = a_{n+1}$, gilt nach Annahme auch $a_0 \overset{+=}{\rightarrow} a_n = a_{n+1}$. Wenn $a_n \rightarrow a_{n+1}$, müssen wir zwei Situationen unterscheiden: Wenn $a_0 = a_n$, gilt $a_0 \rightarrow a_{n+1}$ und damit $a_0 \overset{+}{\rightarrow} a_{n+1}$ und $a_0 \overset{+=}{\rightarrow} a_{n+1}$. Andernfalls, wenn $a_0 \overset{+}{\rightarrow} a_n$, gilt auch $a_0 \overset{+}{\rightarrow} a_{n+1}$ und dann auch $a_0 \overset{+=}{\rightarrow} a_{n+1}$. □

Lemma B.2.

$$(\leftarrow \cup \rightarrow)^+ \supsetneq (\overset{+}{\leftarrow} \cup \overset{+}{\rightarrow}).$$

Proof. 1. $((\overset{+}{\leftarrow} \cup \overset{+}{\rightarrow}) \subseteq (\leftarrow \cup \rightarrow)^+.)$ Wenn $a \overset{+}{\rightarrow} b$ oder $b \overset{+}{\rightarrow} a$ gilt, ist offensichtlich dass auch $a \overset{+}{\leftrightarrow} b$ gilt.

2. $(\leftarrow \cup \rightarrow)^+ \neq (\overset{+}{\leftarrow} \cup \overset{+}{\rightarrow})$ Für das Reduktionssystem $\langle A, \rightarrow \rangle$ mit $A = \{a, b, c\}$ und den Reduktionsschritten $a \rightarrow b$ und $a \rightarrow c$ gilt $b \overset{+}{\leftrightarrow} b$, aber weder $b \overset{+}{\rightarrow} c$ oder $c \overset{+}{\rightarrow} b$. □

B.3 Namen als W-Grammatik

Die vier kontextfreien Regeln definieren verschiedene Formen von Namen: einfache Bezeichner (*identifier*) und rekursiv aufgebaute Namen, in denen eine Feldvariable mit einem Ausdruck indiziert, die Komponente eines Verbundes (*record*, *struct* oder *class*) selektiert oder eine Zeigervariable dereferenziert wird.

```
name ::= identifier
       | name [ expression ]
       | name . identifier
       | name *
```

Mit diesen Regeln kann man gültige Namen ableiten:

```
name ⇒ name *
     ⇒ name . identifier *
     ⇒ identifier . identifier *
     ⇒ x . identifier *    ⇒   x . a *
```

Die Metaregeln der Zweistufengrammatik beschreiben die Struktur der Typen.

```
Type ::= bool | int | ...
       | row Type
       | prod Env fields
       | ref Type

 Env ::  □ | X of Type Envs
   X ::  id x | id a ...
```

In den Hyper-Regeln werden die Hyper-Nichtterminale mit Typausdrücken parameterisiert, die gemäß den Metaregeln gebildet sind, aber statt der Nichtterminale Variablen enthalten. In den folgenden Regeln werden die Variablen X vom Typ X, sowie die Variable T vom Typ *Type* und die Variablen F, E, E_1, E_2 vom Typ *Env* benutzt.

$$
\begin{array}{lr}
\langle T\ \mathit{name\ in}\ E\rangle & \\
\quad ::= \langle X\ \mathit{identifier}\rangle\langle \mathit{where}\ X\ \mathit{has\ type}\ T\ \mathit{in}\ E\rangle & (1)\\
\quad\ \ \mid\ \langle \mathit{row}\ T\ \mathit{name\ in}\ E\rangle\underline{\texttt{[}}\langle \mathit{int\ expression\ in}\ E\rangle\underline{\texttt{]}} & (2)\\
\quad\ \ \mid\ \langle \mathit{prod}\ F\ \mathit{fields\ name\ in}\ E\rangle\underline{\texttt{.}}\langle X\ \mathit{identifier}\rangle\langle \mathit{where}\ X\ \mathit{has\ type}\ T\ \mathit{in}\ F\rangle & (3)\\
\quad\ \ \mid\ \langle \mathit{ref}\ T\ \mathit{name\ in}\ E\rangle\underline{\texttt{*}} & (4)
\end{array}
$$

$$
\langle \mathit{where}\ X\ \mathit{has\ type}\ T\ \mathit{in}\ E_1\ X\ \mathit{has}\ T\ E_2\rangle ::= \Box \qquad (5)
$$

Aus $\langle T\ \mathit{name\ in}\ E\rangle$ kann man nur kontextuell sinnvolle Namen ableiten:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\langle T\ \mathit{name\ in}\ E\rangle}\\
\Rightarrow_4 & \langle \mathit{ref}\ T\ \mathit{name\ in}\ E\rangle\underline{\texttt{*}}\\
\Rightarrow_3 & \langle \mathit{prod}\ F\ \mathit{fields\ name\ in}\ E\rangle\underline{\texttt{.}}\langle X\ \mathit{identifier}\rangle\langle \mathit{where}\ X\ \mathit{has\ type}\ T\ \mathit{in}\ F\rangle\underline{\texttt{*}}\\
\Rightarrow & \langle \mathit{prod}\ F\ \mathit{fields\ name\ in}\ E\rangle\underline{\texttt{.a*}}\quad\langle \mathit{where\ id\ a\ has\ type}\ T\ \mathit{in}\ F\rangle\\
\Rightarrow_2 & \langle \mathit{row\ prod}\ F\ \mathit{fields\ name\ in}\ E\rangle\underline{\texttt{[}}\langle \mathit{int\ expression\ in}\ E\rangle\underline{\texttt{].a*}}\\
 & \langle \mathit{where\ id\ a\ has\ type}\ T\ \mathit{in}\ F\rangle\\
\Rightarrow & \langle \mathit{row\ prod}\ F\ \mathit{fields\ name\ in}\ E\rangle\underline{\texttt{[}}\langle \mathit{number}\rangle\underline{\texttt{].a*}}\\
 & \langle \mathit{where\ id\ a\ has\ type}\ T\ \mathit{in}\ F\rangle\\
\Rightarrow & \langle \mathit{row\ prod}\ F\ \mathit{fields\ name\ in}\ E\rangle\underline{\texttt{[42].a*}}\\
 & \langle \mathit{where\ id\ a\ has\ type}\ T\ \mathit{in}\ F\rangle\\
\Rightarrow_1 & \langle X\ \mathit{identifier}\rangle\langle \mathit{where}\ X\ \mathit{has\ type\ row\ prod}\ F\ \mathit{fields\ in}\ E\rangle\underline{\texttt{[42].a*}}\\
 & \langle \mathit{where\ id\ a\ has\ type}\ T\ \mathit{in}\ F\rangle\\
\Rightarrow & \underline{\texttt{x}}\langle \mathit{where\ id\ x\ has\ type\ row\ prod}\ F\ \mathit{fields\ in}\ E\rangle\underline{\texttt{[42].a*}}\\
 & \langle \mathit{where\ id\ a\ has\ type}\ T\ \mathit{in}\ F\rangle
\end{array}
$$

Die beiden Prädikate, die hier noch übrig gelassen wurden, leiten das leere Wort ab, wenn die Metavariablen E und F wie folgt substitutiert werden:

$$
\begin{aligned}
E &\mapsto \langle E_1\ \mathit{id\ x\ has\ type\ row\ prod}\ F\ \mathit{fields}\ E_2\rangle\\
F &\mapsto \langle F_1\ \mathit{id\ a\ has\ type}\ T\ F_2\rangle
\end{aligned}
$$

(Hierbei sind F_1, F_2, E_1, E_2 weitere Meta-Variablen vom Type *Env*.) Die Ableitung muss also mit dem Hypern-Nichtterminal

$$
\langle T\ \mathit{name\ in}\ E_1\ \mathit{id\ x\ is\ row\ prod}\ F_1\ \mathit{id\ a\ is}\ T\ F_2\ \mathit{fields}\ E_2\rangle
$$

beginnen, oder mit irgendeinem Hyper-Nichtterminal, das daraus durch Substitution der Metavariablen erzeugbat ist, wie zum Beispiel

$$
\langle \mathit{int\ name\ in\ id\ x\ is\ row\ prod\ id\ a\ is\ int\ id\ b\ is\ bool\ fields\ id\ y\ is\ bool}\ E_2\rangle.
$$

Bei der Definition des Metan-Nichtterminals *Type* ist es für Produkttypen (*prod Env fields* wichtig, dass hinter dem Meta-Nichtterminal *Env* ein Meta-Terminal wie *fields*, weil die Metagrammatik von *Env* andernfalls *mehrdeutig* wäre. Ohne "*fields*" wäre in dem Hyper-Nichtterminal

$$\langle \mathit{int\ name\ in\ id\ x\ is\ row\ prod\ id\ a\ is\ int\ id\ b\ is\ bool\ id\ y\ is\ bool}\ E_2 \rangle$$

nicht klar, ob *id a*, *id b* und *id y* Attribute des Produkttypen von *id x* oder aber Variablen in der Umgebung sind, zu der auch *id x* gehört. Dann wäre das Prädikat ⟨*where id a has type T in E*⟩ nicht korrekt, weil die Bezeichner sowohl als Attribute des Produkttypen als auch als Variable benutzt werden könnten. Aus dem Hyper-Nichtterminal oben ließe sich also neben `x[42].a*` der Name `a` auch als Variable ableiten.

B.4 W-Grammatik für bedingte Ausdrücke

Wir benutzen folgende Metaregeln, die monomorphe algebraische Datentypen definieren:

$$\begin{aligned}
E &::= \square \mid X\ \mathit{of}\ T\ E \mid T\ \mathit{of}\ S\ E \\
T &::= \mathit{type\ bool} \mid \mathit{type\ int} \mid \mathit{type\ intlist} \mid \ldots \\
S &::= P \mid P\ \mathit{plus}\ S \\
P &::= K\ \mathit{Ts} \\
\mathit{Ts} &::= \square \mid T\ \mathit{Ts} \\
X &::= \mathit{id\ a} \mid \mathit{id\ x} \mid \ldots \\
K &::= \mathit{con\ true} \mid \mathit{con\ false} \mid \mathit{con\ nil} \mid \mathit{con\ cons} \mid \mathit{con\ leaf} \mid \mathit{con\ branch} \mid \ldots
\end{aligned}$$

Umgebungen E repräsentieren Sequenzen von Definitionen: Variablenbezeichnern X wird ein Typbezeichner T zugeordnet, bzw. Typbezeichnern (außer den vordefinierten Typbezeichnern *type bool* und *type int*) werden Summentypen S zugeordnet. Summentypen S sind nicht leere Listen von Produkttypen P, die jeweil aus Konstruktoren K bestehen, und wiederum auf Sequenzen *Ts* von Typbezeichnern angewendet werden.

Die Startregel ist unverändert.

$\langle \mathit{program} \rangle ::= \langle T\ \mathit{expression\ in}\ \square \rangle$

Die dritte Regel für Ausdrücke definiert nun Konstruktorausdrücke. Die letzte Regel definiert Fallunterscheidungen.

⟨*T expression in E*⟩
::= ⟨*X identifier*⟩ ⟨*where X is is defined as T in E*⟩
| `false` ⟨*where T equals type bool*⟩
| `true` ⟨*where T equals type bool*⟩
| ⟨*number*⟩ ⟨*where T equals type int*⟩
| ⟨*K constructor*⟩ ⟨*Ts expressions in E*⟩
⟨*where K constructs T from Ts in E*⟩
| ⟨*L expression in E*⟩ ⟨*L x R to T operator*⟩ ⟨*R expression in E*⟩
| `(` ⟨*T expression in E*⟩ `)`
| `let` ⟨*X identifier*⟩ `=` ⟨*L expression in E*⟩
`in` ⟨*T expression in E X of L*⟩ ⟨*unless X occurs in E*⟩
| `case` ⟨T_1 *expression in E*⟩
`of` ⟨*T cases with* T_1 *patterns in E*⟩

Sequenzen von Ausdrücken werden in Konstruktor-Anwendungen gebraucht.
⟨□ *expressions in E*⟩ ::= □
⟨*T Ts expressions in E*⟩ ::= ⟨*T expression in E*⟩ ⟨*Ts expressions in E*⟩

Operatoren sind wie bisher definiert.
⟨*type int x type int to type bool operator*⟩ ::= `>` | `>` | `<=` | `>=` | `=` | `/=`
⟨*type int x type int to type int operator*⟩ ::= `-` | `+` | `div` | `mod`

Fallunterscheidungen bestehen jeweils aus einem Muster und einem Ausdruck, der ausgewertet wird, wenn das jeweilige Muster passt; sie werden mit senkrechten Strichen von einander getrennt. Die im Muster auftretenden Variablen können in den Ausdrücken verwendet werden.[1]
⟨*T cases with* T_1 *patterns in E*⟩
::= ⟨T_1 *pattern binding* E_1 *in E*⟩ `=>` ⟨*T expression in* $E_1 E$⟩
| ⟨T_1 *pattern binding* E_1 *in E*⟩ `=>` ⟨*T expression in* $E_1 E$⟩
`|` ⟨*T cases with* T_1 *patterns in E*⟩

Muster sind entweder die Literale `false` oder `true` des vordefinierten Typs *bool*, oder Zahlen des vordefinierten Typs *int*, oder Variablen, die an den geforderten Typ gebunden werden, oder (möglicherweise geklammerte) Anwendungen von Konstruktoren auf Sequenzen von Mustern.

[1] Eigentlich *überdecken* Namen im Muster gleichnamige globale Namen, so dass die Umgebungen eigentlich geschachtelt sein müssten.

⟨*T pattern binding* □ *in E*⟩
  ::= `false` ⟨*where T equals type bool*⟩
  | `true` ⟨*where T equals type bool*⟩
  | ⟨*number*⟩ ⟨*where T equals type int*⟩
  | ⟨*X identifier*⟩
  | ⟨*K constructor*⟩ ⟨*Ts patterns binding* E_1 *in E*⟩
    ⟨*where K constructs T from Ts in E*⟩
  | (⟨*T pattern binding* E_1 *in E*⟩)

Mustersequenzen werden in Konstruktoranwendungen gebraucht.
⟨□ *patterns binding* □ *in E*⟩ ::= □

⟨*T Ts patterns binding* $E_1 E_2$ *in E*⟩
  ::= ⟨*T pattern binding* E_1 *in E*⟩ ⟨*Ts patterns binding* E_2 *in E*⟩
    ⟨*unless* E_1 *and* E_2 *overlap*⟩

Das Prädikat ⟨*where … is defined as …* ⟩ sucht Definitionen von Variablen oder Typbezeichnern in Umgebungen.
⟨*where X is defined as T in* E_1 *X of T* E_2 ⟩ ::= □
⟨*where T is defined as S in* E_1 *T of S* E_2 ⟩ ::= □

Das Prädikat ⟨*unless … occurs…* ⟩ überprüft, ob ein Variablenbezeichner noch nichzt vergeben ist.
⟨*unless X occurs in* □ ⟩ ::= □
⟨*unless X occurs in* X' *of T E*⟩ ::= ⟨*unless X equals* X' ⟩ ⟨*unless X occurs in E*⟩

Das Prädikat ⟨*where K constructs …* ⟩ mit seinem Hilfsprädikat ⟨*… is summand of…* ⟩ überprüft, ob der Konstruktor in einem Summentyp definiert ist.
⟨*where K constructs T from Ts in* E_1 *T of S* E_2 ⟩
  ::= ⟨*where K Ts is summand of S*⟩

⟨*where K Ts is summand of K Ts*⟩ ::= □
⟨*where K Ts is summand of* K' Ts' *plus S*⟩
  ::= ⟨*unless K equals* K' ⟩ ⟨*where K TT is summand of in S*⟩

Die in Teilmuster eines Musters auftretenden Variablen müssen verschieden sein; dies überprüft das Prädikat ⟨*unless* E_1 *and* E_2' *overlap*⟩.
⟨*unless* □ *and* E_2 *overlap*⟩ ::= □
⟨*unless X of T* E_1 *and* E_2 *overlap*⟩
  ::= ⟨*unless X occurs in* E_2 ⟩ ⟨*unless* E_1 *and* E_2 *overlap*⟩

Das Prädikat ⟨*where … equals…* ⟩ macht die Gleichheitsüberprüfung für Typen explizit.
⟨*where T equals T*⟩ ::= □

Ungleichheit von Bezeichnern und Konstruktoren wird vorausgesetzt.

$$\langle \textit{unless id x equals id y} \rangle ::= \Box$$
$$\langle \textit{unless con nil equals con cons} \rangle ::= \Box$$
$$\dots$$

B.5 Terminierend! Konfluent! (zum 1. 12. 2010)

Wir betrachten die Funktionssymbole $\Sigma_0 = \{\mathit{false}, \mathit{true}\}$, $\Sigma_1 = \{\neg\}$ und $\Sigma_2 = \{\wedge, \vee\}$, die darüber konstruierten Termen und die folgenden Termersetzungsregeln über diesen Termen:

$$\begin{aligned}
\rho_1 &: \quad \neg \mathit{true} \to \mathit{false} \\
\rho_2 &: \quad \neg(\neg x) \to x \\
\rho_3 &: \quad \mathit{true} \wedge x \to x \\
\rho_4 &: \quad \mathit{false} \wedge x \to \mathit{false} \\
\rho_5 &: \quad x \vee y \to \neg(\neg x \wedge \neg y) \\
\rho_6 &: \quad \neg \mathit{false} \to \mathit{true}
\end{aligned}$$

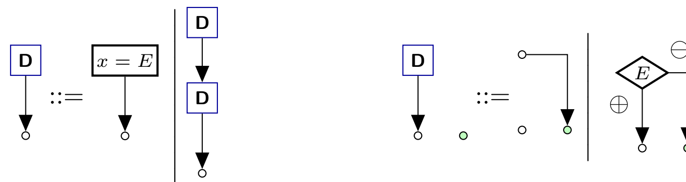
1. Definieren diese Regeln terminierende Ersetzungen?
   a) Finden Sie Argumente dafür, dass die Ersetzungen alle Funktionssymbole $\neg$, $\wedge$ und $\vee$ "verschwinden lassen".
   b) Definieren Sie eine lexikographische Pfadordnung, die mit den Regeln verträglich ist. (Siehe Satz 4.20 im Script.)
2. Sind die Regeln linkslinear?
3. Sind die Regeln nicht-überlappend?
4. Falls einige Regeln einander überlappen, geben Sie deren kritische Paare an und überprüfen Sie, ob diese Paare konvergent sind, d.h., ob sie sich mit den Regeln zu einem Term reduzieren lassen.
5. Definieren diese Regeln also ein terminierendes und konfluentes Termersetzungssystem?

B.6 Unstrukturierte Flussdiagramme

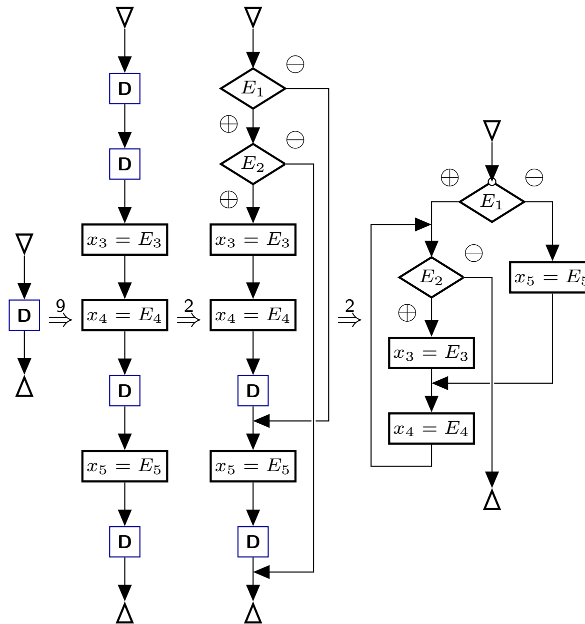
Befehle (*commands*) haben in maschinennahen Programmen, z.B. in der Originalfassung von Fortran, die folgende Form:

$$\begin{array}{lll}
C ::= & x = E & \text{(Zuweisung)} \\
\mid & \textbf{goto}\ \ell & \text{(Sprung)} \\
\mid & \textbf{if}\ E\ \textbf{then goto}\ \ell & \text{(bedingter Sprung)} \\
\mid & \ell : C & \text{(markierter Befehl)} \\
\mid & \ell : & \text{(markiertes Programmende)} \\
\mid & C;\ C & \text{(Befehlssequenz)}
\end{array}$$

1. Wie für die strukturierten Flussdiagramme brauchen wir ein Nichtterminal **D**. Die Sternregeln für Zuweisung und Befehlssequenz können unverändert übernommen werden. Bedingte und unbedingte Sprünge können mit kontextuellen Sternregeln erzeugt werden. Markierungen brauchen nicht erzeugt werden, weil der Kontextknoten der Sprungbefehle eine beliebige Programmstelle sein kann.

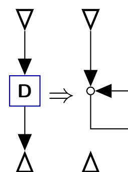


2. Auch der Startgraph ist der gleiche wie für strukturierte Flussdiagramme. Das Flussdiagramm aus der Aufgabe kann so abgeleitet werden:



Die ersten Schritte benutzen sechs Mal die Sequenzregel, um einen **D**-Stern für jeden der nachfolgenden Ableitungsschritte zu erzeugen: zwei für bedingte Sprünge, drei für Zuweisungen, und zwei für unbedingte Sprünge. Danach wird drei Mal die Zuweisungsregel angewendet.
Dann werden die beiden bedingten Sprünge eingefügt, und das Sprungziel richtig gesetzt (vor die Zuweisung an x_5 für den ersten, und vor den Stoppknoten für den zweiten bedingten Sprung.
Die verbliebenen **D**-Sterne werden in unbedingte Sprünge an die entsprechenden Stellen umgewandelt: vor die Bedingung E_2 im ersten Fall, und vor die Zuweisung an x_4 im zweiten Fall.

3. Es gibt Flussdiagramme, in denen der Stoppknoten nicht vom Start erreicht werden kann. Das liegt an der Regel für unbedingte Sprünge, wie die folgende Ableitung zeigt:



4. Folgende Überlegungen legen nahe, dass unstrukturierte Flussdiagramma nicht mit einer Sterngrammatik erzeugt werden können:
Die Regeln für (bedingte) Sprünge müssen zu Sternregeln gemacht werden. Dazu müssten alle Sprungziele, die bei Anwendung dieser Regeln als Kontextknoten gewählt werden könnten, an den **D**-Stern gehängt werden. Da Flussdiagramme beliebig groß werden, müsste an diesem Stern also auch beliebig viele Knoten hängen. Das geht aber nicht, weil Sterne eine feste Anzahl von anhängenden Knoten haben müssen. (Die Knoten müssten auch noch voneinander unterschieden werden können, also an verschieden markierten Armen hängen.)

B.7 Programmgraph-Bedingungen (zum 21. 12. 2010)

Die Bedingungen an Programmgraphen in dem Artikel [Hof10] können wie folgt definiert werden. (Diese Lösung stammt von Fynn Feldpausch.)

Eigenschaft P_4

If an expression refers to a method m, m must be contained in some class of the graph.

$$\forall \left(\{\} \twoheadrightarrow \text{E} \rightsquigarrow \text{M}, \exists\, \text{E} \rightsquigarrow \text{M} \twoheadrightarrow \text{E} \rightsquigarrow \text{M} \leftarrow \text{C} \right)$$

Eigenschaft P_5

If an expression e accesses a variable v contained in a class c, e must be a sub-expression of a body b that is contained in a sub-class of c.

Eigenschaft P_6

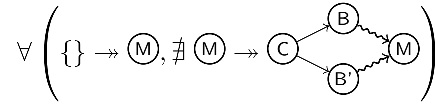
If an expression e accesses a parameter p of a method m, e must be a sub-expression of a body that implements m.

Eigenschaft P_7

If a method body b implements a method signature m, b must be contained in a sub-class of the class c containing m.

Eigenschaft P_8

For every method signature m, every class contains at most one body implementing m.



B.8 Refaktorierung (zum 19. 1. 2011)

Die *Refactoring*-Operation *PushDownMethod* aus Fowler's Katalog tut (annähernd) das Gegenteil der Operation *PullUpMethod*, die in der Vorlesung besprochen wurde:

1. Sei c eine Klasse, in der eine Implementierung b für eine Methode mit der Signatur m definiert wird.
2. Lösche b in c, und füge Kopien von b in allen direkten Unterklassen $c_1, \ldots, c_n$ von c ein (wobei $n \geqslant 1$).
3. Dabei darf keine der Unterklassen $c_1, \ldots, c_n$ die Implementierung von m überschreiben, und keine der Oberklassen von c darf bereits eine Implementierung von m enthalten.

Definieren Sie die Operation *PushDownMethod* als eine Graphtransformationsregel mit Graphvariablen und mehrfachen Teilgraphen.

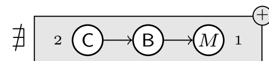
***PushDownMethod* als Graphtransformationsregel**

Bei der Operation wird die Implementierung b einer Methode mit der Signatur m aus einer Klasse c entfernt und dafür in alle Unterklassen von c eingefügt. Die Klasse c muss mindestens eine Unterklasse haben – ansonsten verschwände die Implementierung b vollständig. Deshalb muss der multiple Teilgraph die Angabe "+" tragen, und eine zusaätzliche negative Anwendungsbedingung sicherstellen, dass c keine weiteren direkten Unterklassen hat.

Programmgraph-Bedingung 8

Damit die Programme nach der Refaktorierung noch Bedingung 8 erfüllen, darf keine der direkten Unterklassen von c die Implementierung von m überschreiben. Wäre dies in einer der direkten Unterklassen c_i der Fall, enthielte diese Klasse danach zwei Implementierungen von m. (Bei den Bedingungen nutzen wir wieder die abkürzenden Schreibweise, statt eines (injektiven) Morphismus $L \twoheadrightarrow P$ nur die zu L hinzugefügten Teile von P hinzuschreiben, und an die "Anknüfungsknoten" dieser neuen Teile Nummern zu schreiben, die zu denen in L korrespondieren.)

Die Bedingung muss an alle Instanzen des multiplen Knotens 1 gestellt werden



Darüber hinaus darf keine Oberklasse von c bereits eine Implementierung von m enthalten. Denn hier würde das Entfernen der Implementierung aus der Klasse c eine Verhaltensänderung zur Folge haben. Ein Aufruf der Methode in der Klasse c führt dann zu einem Aufruf der Implementierung aus der Oberklasse. Die Bedingung lautet also wie folgt:

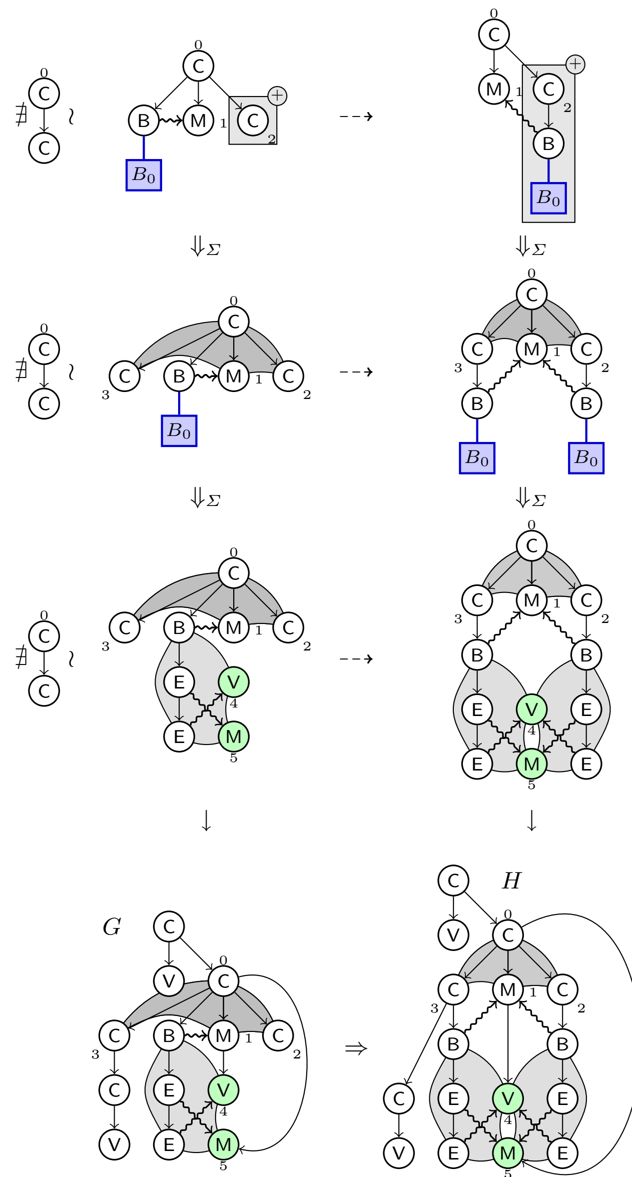
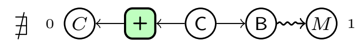


Fig. B.1. A *Push-Down Method* refactoring of a program graph



Weitere Bedingungen

Dynamisch gebundene Methoden stellen bei der *Refactoring*-Operation *PushDownMethod* in den Klassen c_i ein Problem dar. Werden solche Methoden innerhalb der Implementierung b aufgerufen, dann kann sich das Verhalten von b allein durch das Verschieben der Implementierung von c in eine Unterklasse c_i verändern (eben durch die dynamische Bindung). Als weitere Bedingung muss also der Verzicht auf die Nutzung dynamisch gebundener Methoden (die ggf. überschrieben werden) gefordert werden.

C

Material

Hier fassen wir noch einmal zusammen, wo der Stoff des Kurses nachgelesen werden kann. Die angegebenen Quellen finden sich auf der Heimatseite der Veranstaltung im Netz, bis auf die Arbeit [Plu99b], die in der Vorlesung verteilt wurde.

Regel

- Abstrakte Reduktion [Plu99b, Abschnitt 2.2].

Wort

Kapitel 3 im Skript

- Chomsky-Grammatiken, kontextfreie Grammatiken
- Van-Wijngaarden-Grammatiken und Attributgrammatiken [MS97, Abschnitte 4 und 5]

Term

Abschnitte 4.1 und 4.2 im Script

- Definition und Beispiele [Klo92, Seite 11-19]
- Termination [Klo92, Seite 29-40]
- Konfluenz und kritische Paare [Klo92, Seite 40-55]

Graph

(Hier ist das Skript leider nicht hilfreich, weil zu sehr Baustelle!)

- (Hyper-) Graphen [Plu99b, Abschnitt 2.3].
- (Hyper-) Graphersetzung, Ausschneiden und EInbettung von Ableitungen [Plu99b, Kapitel 12].
- Parallele und sequenzielle Unabhängigkeit [Plu99b, Kapitel 13].
- Konfluenz und kritische Paare [Plu99b, Kapitel 13].
- Attributierte Graphersetzung [PS04b, bis Abschnitt 3].
- Sterngrammatiken [HM10, Abschnitt 3]. (Hintergrund: Hyperkantenersetzung [DHK97])
- Programmgraphen allgemein [Hof10, Abschnitt 1,2].
- rekursive Graphbedingungen ("HR-conditions") [HR10].
- bedingte kontextuelle Sterngrammatiken [HM10, ab Abschnitt 4].
- Graphersetzungsregeln mit Variablen [DHJ+07] (In der Vorlesung wurde die Gestalt der Graphen mit bedingten kontextuellen Graphgrammatiken definiert anstatt mit adaptiven Sterngrammatiken wie in diesem Papier.) Mit Sterngrammatiken (Hyperkantenersetzung wurden sie in [Hof01] definiert.)
- Berechnungen mit Graphprogrammaen [PS04b, ab Abschnitt 4].

Nachlieferung

Zum Thema "Gestalt bewahrende Graphtransformation auf der Basis von bedingten kontextuellen Sterngrammatiken" sind gerade zwei Papier in Arbeit.

Alle TeilnehmerInnen der Veranstaltung werden es bekommen, sobald es fertig ist. (*Auf Wunsch auch signiert*;-)