Lehr- und Forschungsgebiet Informatik II
Prof. Dr. J. Giesl

Terminierungsanalyse von Haskellprogrammen

Stephan Swiderski
Mat-Nr.: 222466

Diplomarbeit in Informatik an der
Rheinisch-Westfälischen Technischen Hochschule Aachen,
vorgelegt der Fakultät für Mathematik, Informatik und
Naturwissenschaften

Gutachter:

Prof. Dr. J. Giesl
Lehr- und Forschungsgebiet Informatik II
RWTH Aachen

Prof. em. Dr. K. Indermark
Lehrstuhl Informatik II
RWTH Aachen

Danksagung

Ich bedanke mich herzlich bei

- meinem Betreuer Herrn Prof. Dr. Jürgen Giesl für seine freundliche Unterstützung und die Möglichkeit, an einem interessanten Thema innerhalb des AProVE-Projekts zu arbeiten. Die Mitarbeit am AProVE-Projekt war eine schönes und spannendes Erlebnis.
- meinem Zweitgutachter Herrn Prof. em. Dr. Klaus Indermark für die Bereitschaft, diese Arbeit zu bewerten.
- den Assistenten Peter Schneider-Kamp und René Thiemann, die immer ein offenes Ohr für mich hatten und keine Mühe gescheut haben, mir mit Rat und Tat und einem Guiness beiseite zu stehen.
- den Assistenten Volker Stolz und Michael Weber für ihre Unterstützung bei so manchen Haskell- oder LaTeX-Fragen.
- meinen Kollegen Ralf Behle, Carsten Fuhs, Christian Käunicke, Martin Mertens und Matthias Sondermann für ihre freundliche Unterstützung in so vielen Lebenslagen, vom Kaffee bis zum „Listener“ nach 14 Uhr.
- meinen Eltern Gisela und Siegfried Swiderski für ihre Unterstützung und die Mühe, diese Arbeit Korrektur zu lesen.
- den Mitgliedern des gesamten AProVE-Teams für eine gute und freundliche Zusammenarbeit.

Hiermit versichere ich, daß ich die Arbeit selbständig verfaßt und keine anderen als die angegeben Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 8. Dezember 2005

(Stephan Swiderski)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Bedeutung von Computersystemen für die Gesellschaft nimmt seit Jahren stetig zu. Es werden immer mehr Bereiche mit Hilfe der elektronischen Informationsverarbeitung optimiert. Dabei erhöht sich die Abhängigkeit von den verwendeten Computersystemen und deren Vernetzung, wie etwa in firmeninternen Netzen oder gar dem World Wide Web. Durch den Ausfall einer oder mehrer Computer kann es zu ernsten Folgen für den betroffen Personenkreis kommen. Da inzwischen nicht nur der Luxus-Sektor mit Computersystemen ausgestattet ist, kann sich ein Ausfall nicht nur in Unannehmlichkeiten äußern, sondern durchaus lebensbedrohliche Folgen haben, wie etwa im medizinischen Bereich. Meist wird bei solch kritischen Anwendungen ein Backup-System integriert, das die Aufgaben in einem Störfall ausreichend übernehmen soll. Problematisch wird diese Methode der Absicherung durch Backup-Systeme dann, wenn nicht nur die Hardware dieser Systeme ausfällt, sondern schon deren Programmierung nicht korrekt ist. Die Systeme verhalten sich in diesen Fällen exakt so, wie es deren Programmierung vorgibt, nur erfüllen sie die gewünschte Aufgabe nicht. Um Fehler zu vermeiden, wird die Qualitätssicherung von den meisten Softwareherstellern mit Hilfe von Tests durchgeführt. Leider können Tests nicht alle im realen Anwendungsfeld einer Software auftretenden Situationen abdecken. Eine andere Möglichkeit ist es, einen Korrektheitsbeweis der Software zu führen, um so alle Situationen des Anwendungsfelds zu berücksichtigen. Bei der Größe moderner Softwareprodukte ist dies ein schwieriges und sehr zeitaufwendiges Unterfangen. Daher ist es wünschenswert, Verifikationen dieser Form weitestgehend automatisch zu führen.

Ein wichtiger Teilbereich der automatischen Programmverifikation ist die Terminierungsanalyse. Allerdings ist die Terminierung eines Programms unentscheidbar, so daß es kein vollständiges und korrektes Terminierungsanalyseverfahren gibt. Es ist aber durchaus möglich, unvollständige Terminierungskriterien zu entwickeln und zu überprüfen, ob ein Programm diese Kriterien erfüllt. Seit ein paar Jahren werden Terminierungsanalyse-Tools entwickelt, die mit Hilfe von Sammlungen solcher Terminierungskriterien versuchen, die Terminierung möglichst vie-

ler Programme nachzuweisen. Die Terminierungsanalyse-Tools arbeiten größtenteils mit Termersetzungssystemen als Eingabe. Das heißt, ein Programmierer kann ein solches Tool erst dann verwenden, wenn er sein Programm in ein entsprechendes Termersetzungssystem übersetzt. Diese Übersetzung ist für die meisten verwendeten modernen Programmiersprachen nicht einfach, denn diese enthalten Konzepte und Strukturen, welche dem Anwender entgegenkommen, aber gleichzeitig eine Übersetzung erschweren. Eine weitere Schwierigkeit ist es, die Übersetzung so zu gestalten, daß die Terminierung der resultierenden Termersetzungssysteme möglichst einfach nachzuweisen ist. Daher ist es wünschenswert, die Übersetzung ebenfalls automatisch erledigen zu lassen.

Das Ziel dieser Arbeit ist eine Erweiterung des erfolgreichen Terminierungsanalyse-Tools AProVE [GTSKF04] um ein Modul für die Behandlung von Haskellprogrammen, denn Haskell [J+98] ist eine der gebräuchlichsten funktionalen Programmiersprachen. Das neue Modul soll Haskellprogramme in Dependency-Pair-Probleme, wie sie in [GTSK05] beschrieben sind, überführen. Auf diese Dependency-Pair-Probleme sollen die bereits in AProVE implementierten Techniken angewendet werden. Aus dem erfolgreichen Endlichkeitsnachweis dieser Probleme soll die Terminierung der ursprünglichen Haskellprogramme folgen.

Ein ähnliches Modul für Logik-Programme wurde von Christian Käunicke im AProVE-Projekt realisiert und ist in seiner Diplomarbeit „Automatic Termination Analysis of Logic Programs“ [Käu05] beschrieben. Ebenso gibt es zwei Module für die Sprachen FP (funktional) und IPAD (imperativ), welche im Rahmen der Diplomarbeit „Transformation Techniques to Verify Imperative and Functional Programs“ von Christian A. Haselbach [Has04] implementiert worden sind.

Durch das Haskell-Modul wird das erste Mal eine Unterstützung für eine vollwertige, reale Programmiersprache in AProVE eingebracht. Speziell für Haskell liegen dabei die Schwierigkeiten in den Konzepten, die dem Programmierer angeboten werden:

- higher-order
- lazy evaluation
- Polymorphismus
- Typen mit Klassen

Diese Konzepte wurden so bislang in keiner Eingabesprache von AProVE berücksichtigt. Die Idee, Core-Haskell statt Haskell als Eingabesprache zu wählen, wurde verworfen, weil Core-Haskell so angelegt ist, daß eine realer Lauf eines nach Core-Haskell übersetzten Haskellprogramms möglichst einfach und effizient gestaltet

werden kann, während Aspekte, die die Terminierungsanalyse erheblich vereinfachen, nur unzureichend berücksichtigt werden, so daß die Terminierungsanalyse unnötigerweise erschwert wird.

Am Anfang dieser Arbeit werden in Kapitel 2 die notwendigen formalen Grundlagen für Haskellprogramme eingeführt. Kapitel 3 behandelt semantikerhaltende Reduktionen und deren Kombination zur Vereinfachung der Haskellprogramme. Diese so reduzierten Haskellprogramme kommen den späteren Analysen entgegen. Im Kapitel 4 wird der eigentliche Begriff der Terminierung von Haskellprogrammen präzisiert. Anschließend werden die Narrowing-Graphen eingeführt, welche das Auswertungsverhalten von reduzierten Haskellprogrammen repräsentieren und eine Erweiterung der von Sven Eric Panitz und Manfred Schmidt-Schauß in [PSS97] vorgestellten Termination-Tableaux sind, wobei Olivier Fissore, Isabelle Gnaedig und Hélène Kirchner einen ähnlichen Ansatz mit ihrem Tool Cariboo [FGK02] verfolgen. Anhand der Narrowing-Graphen wird in Kapitel 5 ein Terminierungskriterium für die reduzierten Haskellprogramme entwickelt. Anschließend wird erklärt, wie Dependency-Pair-Probleme aus einem Narrowing-Graph abgelesen werden können, so daß, wenn diese endlich sind, das Terminierungskriterium des Haskellprogramms, zu dem der Narrowing-Graph gehört, erfüllt ist. Das Kapitel schließt mit einem Korrektheitsbeweis, der uns bestätigen wird, daß aus der Endlichkeit der abgelesenen Dependency-Pair-Probleme die Erfüllung des Terminierungskriteriums folgt. Eine Strategie zum Erstellen von Narrowing-Graphen aus reduzierten Haskellprogrammen wird in Kapitel 6 vorgestellt. Diese Strategie erstellt einen Narrowing-Graph so, daß die Dependency-Pair-Probleme, die später aus diesem abgelesen werden, möglichst günstig für den Endlichkeitsnachweis mit Hilfe von AProVE sind. Im letzten Kapitel werden eine Zusammenfassung und schließlich einige Ausblicke gegeben.

Kapitel 2

Haskellprogramme

In diesem Kapitel werden Haskellprogramme und deren Grundstrukturen eingeführt. Die dafür notwendigen Definitionen sind so von uns angelegt worden, daß sie die Informationen, die später für die Terminierungsanalyse von Haskellprogrammen relevant sind, möglichst einfach repräsentieren. Da die Definitionen teilweise zyklisch sind, werden hier schon einmal die benutzten Mengen vorgestellt:

- V: Variablen,
- N: Konstruktornamen,
- $\mathsf{D}(\mathsf{N})$: Typdefinitionen,
- $\mathsf{T}(\mathsf{D},\mathsf{V})$: Haskelltypen,
- $\mathsf{CC}(\mathsf{D},\mathsf{V})$: Klassenbedingungen,
- $\mathsf{S}(\mathsf{D},\mathsf{V})$: Typschemata,
- $\mathsf{P}_\mathsf{B}(\mathsf{D},\mathsf{V})$: Basispatterns,
- $\mathbb{A}$: ASCII-Literale,
- $\mathsf{P}(\mathsf{D},\mathsf{V})$: Patterns,
- $\mathsf{P}_\mathsf{S}(\mathsf{D},\mathsf{V})$: Spezial-Patterns,
- $\mathsf{A}(\mathsf{D},\mathsf{V})$: Atome von Basis-Haskelltermen,
- $\mathsf{H}_\mathsf{B}(\mathsf{D},\mathsf{V})$: Basis-Haskellterme,
- $\mathsf{H}(\mathsf{D},\mathsf{V})$: Haskellterme,
- $\mathsf{CR}(\mathsf{D},\mathsf{V})$: Bedingte Regeln,

- $\mathsf{R}(\mathsf{D},\mathsf{V})$: Regeln,
- $\mathsf{R}_1(\mathsf{D},\mathsf{V})$: Regeln mit genau einem Pattern,
- $\mathsf{F}(\mathsf{D},\mathsf{V})$: Funktionen,
- $\mathsf{C}(\mathsf{D},\mathsf{V})$: Klassen,
- $\mathsf{I}(\mathsf{D},\mathsf{V})$: Instanzen,
- $\mathsf{PC}(\mathsf{D},\mathsf{V})$: Programmkonstrukte und
- $\mathsf{SUB}(\mathsf{D},\mathsf{V})$: Substitutionen

Die definierten Begriffe werden der Verständlichkeit wegen anhand des folgenden Beispiels eines Haskellprogramms verdeutlicht:

Beispiel 2.1

```
data Nat = Succ !Nat | Zero
data List a = Cons a (List a) | Nil

length :: List a → Nat
length Nil = Zero
length (Cons x xs) = Succ (length xs)

contains :: Equal a ⇒ List a → a → Bool
contains Nil = False
contains (Cons x xs) y | equal x y = True
                       | otherwise = contains xs y

class Equal a where
  equal :: a → a → Bool
  notEqual :: a → a → Bool
  notEqual x y = not (equal x y)

instance Equal Nat where
  equal Zero Zero = True
  equal (Succ x) (Succ y) = equal x y
  equal Zero (Succ x) = False
  equal (Succ x) Zero = False

instance Equal a ⇒ Equal (List a) where
  equal Nil Nil = True
  equal (Cons x xs) (Cons y ys) = equal x y && equal xs ys
  equal (Cons x xs) Nil = False
  equal Nil (Cons x xs) = False
```

Definition 2.1 (Variablen) *Die Menge der Variablen* V *ist definiert als*

$$\mathsf{V} := \mathsf{V_F} \uplus \mathsf{V_L} \uplus \mathsf{V_T}$$

wobei

- $\mathsf{V_F}$ *die Funktionsvariablen,*
- $\mathsf{V_L}$ *die lokalen Variablen und*
- $\mathsf{V_T}$ *die Typvariablen*

sind. Zusätzlich sind $\mathsf{V_F}$, $\mathsf{V_L}$ *und* $\mathsf{V_T}$ *der Einfachheit halber paarweise disjunkt und unendliche abzählbare Mengen.*

Im Beispiel 2.1 gibt es die Funktionsvariablen `length`, `contains`, `equal` und `notEqual`, die Typvariable a und die lokalen Variablen x,y und xs. Ein Haskellprogramm in dem Typvariablen und Variablen mit gleichen Namen vorkommen, können o.B.d.A immer so umgeschrieben werden, daß die Namen der Typvariablen und Variablen disjunkt sind.

Definition 2.2 (Konstruktornamen) N *ist eine Menge von Namen, für die*

- $\rightarrow \in \mathsf{N}$ *und*
- $* \in \mathsf{N}$ *und*
- $\mathsf{N} \cap \mathsf{V} = \emptyset$

gilt.

Später werden aus dieser Menge von Namen die einzelnen Namen für Konstruktoren, Typkonstruktoren und Klassen bezogen.

Die Typdefinition beinhaltet alle Datentypen eines Haskellprogramms. Die Stelligkeit der Konstruktoren wird hier noch nicht definiert, denn diese wird später mit Hilfe der Typschemata der einzelnen Konstruktoren eingeführt.

Definition 2.3 (Typdefinition) *Die Typdefinition* D *ist definiert durch:*

$$\mathsf{D} := \langle \underline{\mathsf{TyCons}}_\mathsf{D}, \underline{\mathsf{Class}}_\mathsf{D}, \underline{\mathsf{Cons}}_\mathsf{D}, \mathrm{constr}_\mathsf{D} \rangle$$

wobei

- $\{*, \rightarrow\} \subseteq \underline{\mathsf{TyCons}}_\mathsf{D} \subseteq \mathsf{N}$,
- $\underline{\mathsf{Class}}_\mathsf{D} \subseteq \mathsf{N}$,

- $\underline{\mathsf{Cons}}_{\mathsf{D}} \subseteq \mathsf{N} und$
- $\emptyset = \underline{\mathsf{TyCons}}_{\mathsf{D}} \cap \underline{\mathsf{Class}}_{\mathsf{D}} = \underline{\mathsf{Class}}_{\mathsf{D}} \cap \underline{\mathsf{Cons}}_{\mathsf{D}} = \underline{\mathsf{TyCons}}_{\mathsf{D}} \cap \underline{\mathsf{Cons}}_{\mathsf{D}}$

gilt. Dabei sind $\underline{\mathsf{TyCons}}_{\mathsf{D}}$ *die Typkonstruktoren,* $\underline{\mathsf{Class}}_{\mathsf{D}}$ *die Klassennamen und* $\underline{\mathsf{Cons}}_{\mathsf{D}}$ *die Datenkonstruktoren, die innerhalb des zugehörigen Haskellprogramms auftreten. Die Funktion* $\mathrm{constr}_{\mathsf{D}} : \underline{\mathsf{TyCons}}_{\mathsf{D}} \longrightarrow \mathrm{Pot}(\underline{\mathsf{Cons}}_{\mathsf{D}})$ *bildet jeden Typkonstruktor auf die Menge seiner Konstruktoren ab. Außerdem wird die Menge der Typdefinitionen mit* $\mathsf{D}(\mathsf{N})$ *bezeichnet.*

Die Typdefinition D_{ex} für das Beispiel 2.1 ist:

$$\begin{aligned}\mathsf{D}_{ex} := \langle &\{\rightarrow, \texttt{Bool}, \texttt{Nat}, \texttt{List}\},\\ &\{\texttt{Equal}\},\\ &\{\texttt{True}, \texttt{False}, \texttt{Succ}, \texttt{Zero}, \texttt{Cons}, \texttt{Nil}\},\\ &\mathrm{constr}_{\mathsf{D}_{ex}}\rangle\end{aligned}$$

mit

- $\mathrm{constr}_{\mathsf{D}_{ex}}(\rightarrow) = \emptyset$,
- $\mathrm{constr}_{\mathsf{D}_{ex}}(\texttt{Bool}) = \{\texttt{True}, \texttt{False}\}$,
- $\mathrm{constr}_{\mathsf{D}_{ex}}(\texttt{Nat}) = \{\texttt{Succ}, \texttt{Zero}\}$,
- $\mathrm{constr}_{\mathsf{D}_{ex}}(\texttt{List}) = \{\texttt{Cons}, \texttt{Nil}\}$,

Es ist zu beachten, daß der Datentyp `Bool` in jedem Haskellprogramm vorkommt, ohne daß dieser explizit angeben werden muß oder benutzt werden muß. Genauso werden automatisch vom Parser die speziellen Typen und Konstruktoren für die verwendeten Tupel jeder Stelligkeit erzeugt. Für die vordefinierten Listen die durch die Konstruktoren [] und : erzeugt werden, ist ebenfalls der Typ [a] vorhanden. Ungeachtet der besonderen Schreibweise von Tupeln und vordefinierten Listen, können diese immer in ihrer Prefixform geschrieben werden, so daß hier keine spezielle Unterscheidung zu anderen Konstruktoren unternommen wird. O.B.d.A. kann also statt des Tupels (a, b, c) auch der Term $(,,)\ a\ b\ c$ geschrieben werden und statt $a : as$ kann $(:)\ a\ as$ verwendet werden. Genauso kann für den Typ [a] auch $[]\ a$ geschrieben werden. Beide Schreibweisen sind in Haskell erlaubt und haben dieselbe Semantik.

Definition 2.4 (Haskelltypen) *Die Menge der Haskelltypen* $\mathsf{T}(\mathsf{D}, \mathsf{V})$ *ist die kleinste Menge für die gilt:*

- $\mathsf{V}_{\mathsf{T}} \subseteq \mathsf{T}(\mathsf{D}, \mathsf{V})$
- $\underline{\mathsf{TyCons}}_{\mathsf{D}} \subseteq \mathsf{T}(\mathsf{D}, \mathsf{V})$
- $(\tau_1\ \tau_2) \in \mathsf{T}(\mathsf{D}, \mathsf{V})$, *wenn* $\tau_1, \tau_2 \in \mathsf{T}(\mathsf{D}, \mathsf{V})$

Der Funktions-Typkonstruktor $\rightarrow \in \mathsf{N}$ *wird üblicherweise in Infix-Notation geschrieben, assoziert dabei nach rechts und hat niedrigere Priorität als alle anderen Typkonstruktoren. Z.B. steht* $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ *abkürzend für* $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ *und für* $((\rightarrow\ \tau_1)((\rightarrow\ \tau_2)\ \tau_3))$*. Außerdem ist* $\mathsf{V}(\tau)$ *die Menge der Typvariablen, die in dem Typ* τ *vorkommen. Es ist zu beachten, daß unpassende Typen vom Haskell-Typchecker abgefangen und deswegen später nicht mehr auftreten werden.*

Im Beispiel 2.1 hat die Funktion `contains` den Typ

$$\texttt{List}\ a \rightarrow a \rightarrow \texttt{Bool}$$

Die Klassenbedingung `Equal` a nicht Teil des Typs ist.

Um die Typschemata der Funktion und Konstruktoren vollständig definieren zu können, benötigen wir die Menge der Klassenbedingungen.

Definition 2.5 (Klassenbedingungen) *Die Menge der Klassenbedingungen* $\mathsf{CC}(\mathsf{D},\mathsf{V})$ *ist definiert durch:*

$$\mathsf{CC}(\mathsf{D},\mathsf{V}) := \underline{\mathsf{Class}_\mathsf{D}} \times \mathsf{T}(\mathsf{D},\mathsf{V})$$

Für ein Element (c,τ) *aus* $\mathsf{CC}(\mathsf{D},\mathsf{V})$ *wird auch die Schreibweise* $c\ \tau$ *benutzt. Im Beispiel 2.1 gibt es neben anderen die Klassenbedingung* `Equal a` *.*

Jede Funktion und jeder Konstruktor innerhalb eine Haskellprogramms besitzt ein Typschema, welches entweder direkt im Programm angegeben wird oder durch den Typchecker erzeugt wird.

Definition 2.6 (Typschema) *Die Menge der Typschemata* $\mathsf{S}(\mathsf{D},\mathsf{V})$ *ist eine Teilmenge von* $\mathrm{Pot}(\mathsf{V}_\mathsf{T}) \times \mathrm{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \times \mathsf{T}(\mathsf{D},\mathsf{V})$ *und enthält Elemente der Form* $(Q, \underline{cs}, \tau)$*, wenn für die*

- $Q \subseteq \mathsf{V}(\tau)$ *und*
- $\left(\bigcup_{(c\ \tau') \in \underline{cs}} \mathsf{V}(\tau')\right) \subseteq \mathsf{V}(\tau)$

gelten. Q *ist die Menge von allquantifizierten Variablen des Typs* τ*, das heißt, daß die Variablen aus* Q *des Typschemas* $(Q, \underline{cs}, \tau)$ *immer in dessen Typ* τ *enthalten sein müssen. Das gilt ebenfalls für alle Variablen der Klassenbedingungen aus* $\underline{cs}$*. Außerdem werden für das Typschema* $(Q, \underline{cs}, \tau) \in \mathsf{S}(\mathsf{D},\mathsf{V})$ *die folgenden Schreibweisen benutzt:*

- $\forall a_1, \ldots, a_n.\underline{cs} \Rightarrow \tau$*, wenn* $Q = \{a_1, \ldots, a_n\}$
- $\underline{cs} \Rightarrow \tau$*, falls* $Q = \emptyset$

Zusätzlich weist die Funktion schema : $\underline{\mathsf{Cons}_\mathsf{D}} \cup \underline{\mathsf{TyCons}_\mathsf{D}} \longrightarrow \mathsf{S}(\mathsf{D},\mathsf{V})$ *jedem Konstruktor und Typkonstruktor ein Typschema zu. Die Funktion* schema *wird später durch den Typchecker festgelegt, wobei hier nicht von Interesse ist, wie dieser dabei im Detail vorgeht. Funktionsvariablen erhält ihr Typschema durch die an sie gebundenen Funktionen, die in Definition 2.19 erklärt werden.*

Das Typschema der Funktion `contains` des Beispiels 2.1 ist

$$\forall a.\{\texttt{Equal}\ a\} \Rightarrow \texttt{List}\ a \rightarrow a \rightarrow \texttt{Bool}$$

welches nicht mit ihrem Typ

$$\texttt{List}\ a \rightarrow a \rightarrow \texttt{Bool}$$

zu verwechseln ist. Im Beispiel 2.1 werden den Konstruktoren diese Typschemata zugewiesen:

- schema(`Zero`) $= \emptyset \Rightarrow$ `Nat`
- schema(`Succ`) $= \emptyset \Rightarrow$ `Nat` $\rightarrow$ `Nat`
- schema(`Nil`) $= \emptyset \Rightarrow$ `List` a
- schema(`Cons`) $= \emptyset \Rightarrow a \rightarrow$ `List` $a \rightarrow$ `List` a

Für die Typkonstruktoren sind es die folgenden:

- schema(`Nat`) $= \emptyset \Rightarrow *$
- schema(`List`) $= \emptyset \Rightarrow * \rightarrow *$
- schema($TyArrow$) $= \emptyset \Rightarrow * \rightarrow * \rightarrow *$
- schema($*$) $= \emptyset \Rightarrow *$

Der Typkonstruktor $*$ ist der Typ der Typen, es ist eine sogenannte Art. In dieser Arbeit werden Arten nicht weiter benötigt und seien hier nur erwähnt, um zu erklären, wie die Typschemata der Typkonstruktoren aussehen, damit später die Stelligkeit von Typkonstruktoren definiert werden kann. Im Haskell-98-Report [J+98] sind die Arten (englisch: Kinds) genauer beschrieben.

Die Stelligkeit der Konstruktoren wird benötigt, weil innerhalb der Patterns von Regeln diese exakt eingehalten werden müssen.

Definition 2.7 (Stelligkeiten von Typschemata und Konstruktoren)
Die Stelligkeit arity(τ) *des Typs τ ist definiert durch*

$$\text{arity}(\tau) := n$$

wenn $\tau = \tau_1 \to \ldots \to \tau_n \to \tau_0$ *gilt, wobei* $\to$ *der Funktions-Typkonstruktor ist und* τ_0 *nicht die Form* $\tau_0' \to \tau_0''$ *hat.*

Die Stelligkeit eines Typschemas $(Q, \underline{cs}, \tau) \in \mathsf{S}(\mathsf{D}, \mathsf{V})$ *ist als die Stelligkeit des darin enthaltenen Typs* τ *durch* $arity((Q, \underline{cs}, \tau)) := \text{arity}(\tau)$ *definiert. Nun ist die Stelligkeit eines Konstruktors* $\underline{co} \in \underline{\mathsf{Cons}}_\mathsf{D} \cup \underline{\mathsf{TyCons}}_\mathsf{D}$ *die Stelligkeit seines Typschemas* $\text{schema}(\underline{co})$*, also:*

$$arity(\underline{co}) := \text{arity}(\text{schema}(\underline{co}))$$

Definition 2.8 (Striktheit) *In Haskell wird für Parameter von Konstruktoren festgelegt, ob sie jeweils strikt (*$!$*) oder nicht strikt (*$\circ$*) sind. Die Striktheiten eines Konstruktors lassen sich in dem Wort* $s_1 \ldots s_n$ *aus* $\{!, \circ\}^*$ *zusammenfassen, wobei* n *der Konstruktor-Stelligkeit entsprechen muß. Die Funktion* $\text{strictness} :: \underline{\mathsf{Cons}}_\mathsf{D} \mapsto \{!, \circ\}^*$ *gibt diese Festlegungen wieder und ist definiert durch:*

$$\text{strictness}(\underline{co}) := s_1 \ldots s_n$$

wobei

- $\underline{co} \in \underline{\mathsf{Cons}}_\mathsf{D}$
- $n = arity(\underline{co})$ *und*
- $s_1, \ldots, s_n \in \{!, \circ\}$

gilt.

Die Striktheiten für die Konstruktoren des Beispiels 2.1 sind:

- $\text{strictness}(\texttt{Succ}) = !$
- $\text{strictness}(\texttt{Zero}) = \epsilon$
- $\text{strictness}(\texttt{Cons}) = \circ\circ$
- $\text{strictness}(\texttt{Nil}) = \epsilon$

Die Basispatterns sind die Patterns, die, nachdem die Reduktionen aus Kapitel 3 angewendet wurden, nur noch in einem Haskellprogramm vorhanden sein werden.

Definition 2.9 (Basispatterns) *Die Menge der Basispatterns* $\mathsf{P}_\mathsf{B}(\mathsf{D}, \mathsf{V})$ *ist die kleinste Menge, für die gilt:*

- $\mathsf{V}_\mathsf{L} \subseteq \mathsf{P}_\mathsf{B}(\mathsf{D}, \mathsf{V})$
- $(\underline{co}\, p_1 \ \ldots \ p_n) \in \mathsf{P}_\mathsf{B}(\mathsf{D}, \mathsf{V})$*, wobei*

  - $p_1, \ldots, p_n \in \mathsf{P_B}(\mathsf{D}, \mathsf{V})$
  - $p_1, \ldots, p_n$ *paarweise Variablendisjunkt sein müssen, um die Linearität der Patterns zu gewährleisten.*
  - $\underline{co} \in \underline{\mathsf{Cons}}_\mathsf{D}$ *und*
  - $n = \text{arity}(\underline{co})$

Definition 2.10 (ASCII-Literale) *Die Menge der ASCII-Literale wird mit* $\mathbb{A}$ *bezeichnet. Ein ASCII-Literal ist ein ASCII-Zeichen in Hochkommata. So sind* $'\mathtt{A}', '\mathtt{B}', \ldots \in \mathbb{A}$*. Die Funktion* $\text{ord} : \mathbb{A} \longrightarrow \mathbb{N}$ *bildet ein ASCII-Literal auf die Position des ASCII-Zeichens im ASCII-Code ab. Beispielsweise ist* $\text{ord}('\mathtt{A}') = 65$*.*

Hier wird die Menge der linearen Patterns eingeführt, welche die Basis für die Auswertung von Haskellprogrammen darstellen, da Regeln anhand ihrer Patterns ausgewählt werden.

Definition 2.11 (Patterns) *Die Menge der Patterns* $\mathsf{P}(\mathsf{D}, \mathsf{V})$ *ist die kleinste Menge für die*

- $\mathsf{V_L} \subseteq \mathsf{P}(\mathsf{D}, \mathsf{V})$
- $\mathbb{Z} \subseteq \mathsf{P}(\mathsf{D}, \mathsf{V})$
- $\mathbb{Q} \subseteq \mathsf{P}(\mathsf{D}, \mathsf{V})$
- $\mathbb{A} \subseteq \mathsf{P}(\mathsf{D}, \mathsf{V})$
- $\_ \in \mathsf{P}(\mathsf{D}, \mathsf{V})$
- $\sim p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$*, wenn* $p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$
- $x@p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$*, wenn* $x \in \mathsf{V_L}$*,* $p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$ *mit* $x \notin \mathsf{V}(p)$
- $(x + n) \in \mathsf{P}(\mathsf{D}, \mathsf{V})$*, wenn* $x \in \mathsf{V_L}$*,* $n \in \mathbb{N}$ *und* $n > 0$
- $\underline{co}\, p_1 \ \ldots \ p_n \ \in \mathsf{P}(\mathsf{D}, \mathsf{V})$*, wobei*
  - $p_1, \ldots, p_n \in \mathsf{P}(\mathsf{D}, \mathsf{V})$ *paarweise variablendisjunkte Patterns sind,*
  - $\underline{co} \in \underline{\mathsf{Cons}}_\mathsf{D}$ *und*
  - $n = \text{arity}(\underline{co})$

gilt, wobei $\mathsf{V}(p)$ *die Menge der Variablen ist, die in dem Pattern* p *vorkommen.*

Im Beispiel 2.1 gibt es neben anderen die folgenden Patterns:

(`Succ` x)

(`Cons` $x\ xs$)

`Nil`

Definition 2.12 (Spezial-Patterns) *Die Menge der Spezial-Patterns* $\mathsf{P_S(D,V)}$ *ist definiert durch:*

$$\mathsf{P_S(D,V)} = \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A} \cup \{(x+n) \mid x \in \mathsf{V_L}, n \in \mathbb{N}\}$$

Die vier Patterns 1, 1.002, $'a'$ und $(x+3)$ sind zum Beispiel Spezial-Patterns.

Definition 2.13 (Atome) *Die Menge der Atome* $\mathsf{A(D,V)}$ *ist definiert durch:*

$$\mathsf{A(D,V)} := \mathsf{V_L} \cup \mathsf{V_F} \cup \underline{\mathsf{Cons_D}}$$

Die Menge der Basis-Haskellterme ist eine reduzierte Menge der Haskellterme. Basis-Haskellterme sind die einzige Art von Termen, die nach Anwendung der vereinfachenden Reduktionen aus Kapitel 3 innerhalb eines Haskellprogramms noch vorkommen. Dadurch wird die spätere Terminierungsanalyse, welche nur mit Basis-Haskelltermen arbeitet, einfacher gehalten, weil weniger Sonderfälle beachtet werden müssen.

Definition 2.14 (Basis-Haskellterme) *Die Menge der Basis-Haskellterme* $\mathsf{H_B(D,V)}$ *ist definiert durch:*

- $\mathsf{A(D,V)} \subseteq \mathsf{H_B(D,V)}$,
- $(t_1\ t_2) \in \mathsf{H_B(D,V)}$, *wenn* $t_1, t_2 \in \mathsf{H_B(D,V)}$

Für Terme der Form $(\ldots((t_1\ t_2)\ t_3)\ldots\ t_n)$ *wird im folgenden auch die abkürzende Schreibweise* $t_1\ \ldots\ t_n$ *benutzt.* $\mathsf{V}(t)$ *ist die Menge der Variablen, die in dem Basis-Haskellterm* t *vorkommen. Außerdem bezeichnet* $\mathsf{H_B^G(D,V)}$ *die Menge der Basis-Haskellgrundterme, in denen keine lokalen Variablen auftreten. Sie ist folgendermaßen definiert:*

$$\mathsf{H_B^G(D,V)} := \{t \in \mathsf{H_B(D,V)} \mid \mathsf{V}(t) \cap \mathsf{V_L} = \emptyset\}$$

Ein Term $t \in \mathsf{H_B(D,V)}$ *hat die Form* $h\ t_1\ \ldots\ t_n$ *für* $h \in \mathsf{A(D,V)}$, *dabei wird das Atom* h *als Kopf bezeichnet.*

Da die Basis-Haskellterme als Arbeitsgrundlage der Terminierungsanalyse dienen, benötigen wir für diese einige spezielle Relationen, die unter anderem auf der Menge der Stellen eines Terms arbeiten.

Definition 2.15 (Stellen) *Für einen Term $t \in \mathsf{H_B}(\mathsf{D},\mathsf{V})$ ist die Menge der gültigen Stellen $Occ(t) \subseteq \mathbb{N}^*$ die kleinste Menge, für die gilt:*

- $\epsilon \in Occ(t)$
- $\{i\pi \in \mathbb{N}^* \mid i \in \{1,\ldots,n\}, \pi \in Occ(t_i)\} \subseteq Occ(t)$, *wenn* $t = h\ t_1\ \ldots\ t_n$ *und* $h \in \mathsf{A}(\mathsf{D},\mathsf{V})$

Der Teilterm des Terms t an der Stelle $\pi \in Occ(t)$ wird mit $t|_\pi$ bezeichnet, wobei

- $t|_\epsilon = t$
- $h\ t_1\ \ldots\ t_n|_{i\pi} = t_i|_\pi$ *für* $h \in \mathsf{A}(\mathsf{D},\mathsf{V})$ *und* $\pi \in Occ(t_i)$

gilt.
$t[s]_\pi$ bezeichnet den Term, der durch die Ersetzung des Teilterms $t|_\pi$ mit dem Term s entsteht und ist so definiert:

- $t[s]_\epsilon = s$
- $h\ t_1\ \ldots\ t_n[s]_{i\pi} = h\ t_1\ \ldots\ t_i[s]_\pi\ \ldots\ t_n$ *für* $h \in \mathsf{A}(\mathsf{D},\mathsf{V})$

Zwei Stellen $\pi, \pi' \in Occ(t)$ können wie folgt in Relation stehen:

- $\pi <_{\text{lex}} \pi'$ *ist die lexikographische Ordnung auf* $\mathbb{N}^*$
- $\pi \leq_{\text{pre}} \pi'$ *gdw.* $\exists w \in \mathbb{N}^*$: $\pi w = \pi'$

Die allgemeine Teiltermrelation $\trianglelefteq$ ist definiert wie folgt:

- $s \trianglelefteq t$ *gdw.* $\exists \pi \in Occ(t)$: $t|_\pi = s$
- $s \triangleleft t$ *gdw.* $s \trianglelefteq t$ *und* $s \neq t$

Der Term $C\big[t\big]$ bezeichnet einen Term, der durch einen Termkontext $C\big[\ \big]$ und den Term t zusammengesetzt wurde, das heißt $C\big[t\big]$ ist ein Term, in dem es eine Stelle π gibt, für die $C\big[t\big]|_\pi = t$ gilt.

Die eigentlichen Haskellterme werden benötigt, um die Reduktionen aus Kapitel 3 zu erklären.

Definition 2.16 (Haskellterme) *Die Menge der Haskellterme $\mathsf{H}(\mathsf{D},\mathsf{V})$ ist die kleinste Obermenge der Mengen*

- $\mathsf{H_B}(\mathsf{D},\mathsf{V})$
- $\mathbb{Z}$,
- $\mathbb{Q}$ *und*

- $\mathbb{A}$,

welche die Elemente

- (t),
- $(t_1\ t_2)$,
- $\lambda p_1\ \ldots\ p_n \to t$,
- `if` t `then` t_1 `else` t_2,
- `case` t `of` $\underline{rs}$ *und*
- `let` $\underline{fs}$ `in` t

enthält, wenn

- $t, t_1, t_2 \in \mathsf{H}(\mathsf{D}, \mathsf{V})$,
- $p_1, \ldots, p_n \in \mathsf{P}(\mathsf{D}, \mathsf{V})$,
- $\underline{rs} \subseteq \mathsf{R}_1(\mathsf{D}, \mathsf{V})$ *und*
- $\underline{fs} \subseteq \mathsf{F}(\mathsf{D}, \mathsf{V})$

gilt. $\mathsf{R}_1(\mathsf{D}, \mathsf{V})$ *bezeichnet dabei die Menge der Regeln mit einem Pattern und* $\mathsf{F}(\mathsf{D}, \mathsf{V})$ *die Menge der Funktionen. Beide Mengen werden später in den Definitionen 2.18 und 2.19 erklärt. Genauso wie bei den Typen, ist hier zu beachten, daß nicht typkorrekte Terme durch den Haskell-Typchecker abgefangen werden und deswegen später nicht mehr auftreten.*

Haskellterme des Beispiels 2.1 sind unter anderen:

```
contains xs y
```

```
True
```

```
equal x y && equal xs ys
```

Definition 2.17 (Bedingte Regeln) *Die Menge aller bedingten Regeln* $\mathsf{CR}(\mathsf{D}, \mathsf{V})$ *ist definiert durch:*

$$\mathsf{CR}(\mathsf{D}, \mathsf{V}) := \mathsf{H}(\mathsf{D}, \mathsf{V}) \times \mathsf{H}(\mathsf{D}, \mathsf{V})$$

Der erste Term c des Paars $(c, t) \in \mathsf{CR}(\mathsf{D}, \mathsf{V})$ wird als Bedingung und der zweite Term t als Ergebnisterm bezeichnet. Für ein Wort $(c_1, t_1) \ldots (c_n, r_n)\ \in \mathsf{CR}(\mathsf{D}, \mathsf{V})^*$ wird auch die Schreibweise

$$|c_1 = t_1 \ldots |c_n = t_n$$

verwendet. Speziell für das Wort $(\texttt{True}, t)$ *aus* $\mathsf{CR}(\mathsf{D},\mathsf{V})^*$ *kann die Bedingung auch weggelassen werden und es wird* $= t$ *geschrieben. Bedingte Regeln innerhalb eines* `case`*-Konstrukts werden auch in dieser Notation geschrieben:*

$$|c_1 \ \rightarrow \ t_1 \ldots |c_n \ \rightarrow \ t_n$$

Diese Schreibweise ist nur eine syntaktische Besonderheit von Haskell, während die Semantik davon unberührt beleibt, deswegen wird innerhalb dieser formalen Definition auf eine solche Unterscheidung verzichtet.

Die Funktion `contains` des Beispiels 2.1 enthält in ihrer letzten Regel die beiden bedingten Regeln:

$|\texttt{equal}\ x\ y = \texttt{True}$

$|\texttt{otherwise} = \texttt{contains}\ xs\ y$

Die Funktion `length` des Beispiels enthält die bedingte Regel $=$ `Zero` welche abkürzend für $|\texttt{True} = \texttt{Zero}$ steht.

Definition 2.18 (Regeln) *Die Menge aller Regeln* $\mathsf{R}(\mathsf{D},\mathsf{V})$ *ist definiert durch:*

$$\mathsf{R}(\mathsf{D},\mathsf{V}) := \mathsf{P}(\mathsf{D},\mathsf{V})^* \times \mathsf{CR}(\mathsf{D},\mathsf{V})^+ \times \mathrm{Pot}(\mathsf{F}(\mathsf{D},\mathsf{V}))$$

Für $r \in \mathsf{R}(\mathsf{D},\mathsf{V})$ *mit* $r = (p_1 \ldots p_n, \underline{cl}, \underline{fs})$ *ist die Stelligkeit* $\mathrm{arity}(r)$ *gleich* n*. Die dritte Komponente* $\underline{fs} \in \mathrm{Pot}(\mathsf{F}(\mathsf{D},\mathsf{V}))$ *ist die Menge der lokal definierten Haskell-Funktionen.*

Außerdem wird statt $(p_1 \ldots p_n, |c_1 = t_1 \ldots |c_m = t_m, \underline{fs})$ *auch*

$$\begin{array}{l} p1 \ \ldots \ p_n \ |c_1 = t_1 \\ \qquad\qquad\ \vdots \\ \qquad\qquad\ |c_m = t_m \ where \ \underline{fs} \end{array}$$

geschrieben. Die Menge der Regeln mit genau einem Pattern $\mathsf{R}_1(\mathsf{D},\mathsf{V})$ *wird definiert durch:*

$$\mathsf{R}_1(\mathsf{D},\mathsf{V}) := \mathsf{P}(\mathsf{D},\mathsf{V}) \times \mathsf{CR}(\mathsf{D},\mathsf{V})^+ \times \mathrm{Pot}(\mathsf{F}(\mathsf{D},\mathsf{V}))$$

Die letzte Regel der Funktion `contains` des Beispiels 2.1 ist:

$$\begin{array}{l} (\texttt{Cons}\ x\ xs)\ y \ |\texttt{equal}\ x\ y \ = \texttt{True} \\ \qquad\qquad\qquad\qquad\ \ |\texttt{otherwise} = \texttt{contains}\ xs\ y \end{array}$$

Die `where`-Klausel wird weggelassen, wenn die Menge der lokalen Funktionen, wie in dieser Regel, leer ist.

Definition 2.19 (Funktionen) *Die Menge aller Haskellfunktionen* $\mathsf{F}(\mathsf{D},\mathsf{V})$ *ist definiert durch:*

$$\mathsf{F}(\mathsf{D},\mathsf{V}) := \mathsf{V_F} \times \mathsf{S}(\mathsf{D},\mathsf{V}) \times \mathsf{R}(\mathsf{D},\mathsf{V})^*$$

Für $(v, s, r_1 \ldots r_n) \in \mathsf{F}(\mathsf{D},\mathsf{V})$ *wird auch*

$$\begin{pmatrix} v :: s \\ v\ r_1 \\ \vdots \\ v\ r_n \end{pmatrix}$$

geschrieben.

Die an die Funktionsvariable `contains` aus Beispiel 2.1 gebundene Funktion $f_{\texttt{contains}}$ ist:

$$f_{\texttt{contains}} := \begin{pmatrix} \texttt{contains} :: \texttt{Equal}\ a \Rightarrow \texttt{List}\ a\ \to\ a\ \to\ \texttt{Bool} & & \\ \texttt{contains}\ \texttt{Nil} & y & = \texttt{False} \\ \texttt{contains}\ (\texttt{Cons}\ x\ xs)\ y & \mid \texttt{equal}\ x\ y & = \texttt{True} \\ & \mid \texttt{otherwise} & = \texttt{contains}\ xs\ y \end{pmatrix}$$

Da die Typen innerhalb eine Haskellprogramms bei der Terminierungsanalyse mit betrachtet werden sollen, benötigen wir auch die Definitionen der Klassen und Instanzen eines Haskellprogramms.

Definition 2.20 (Klassen) *Die Menge aller Klassen* $\mathsf{C}(\mathsf{D},\mathsf{V})$ *ist die kleinste Teilmenge von* $\mathrm{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \times \mathsf{CC}(\mathsf{D},\mathsf{V}) \times \mathrm{Pot}(\mathsf{F}(\mathsf{D},\mathsf{V}))$*, welche nur Elemente der Form* $(\{c_1\ a, \ldots, c_n\ a\}, c\ a, \underline{fs})$ *enthält, wenn*

- $a \in \mathsf{V_T}$ *und*
- $c\ a \in \underline{cs}$ *für alle* $(v, \forall Q.\underline{cs} \Rightarrow \tau, r) \in \underline{fs}$

gilt und $\underline{fs}$ *endlich ist. Die Klassenbedingungen c a müssen also im Typschema jeder Member-Funktion enthalten sein, während* $\{c_1\ a, \ldots, c_n\ a\}$ *die Oberklassen sind. Funktionsvariablen die an die Member-Funktionen gebunden sind, werden auch kurz als Member-Variablen einer Klasse bezeichnet.*

Im Beispiel 2.1 wäre die Menge der Klassen folgende:

$$\{(\emptyset, \texttt{Equal a}, \underline{fs})\}$$

wobei $\underline{fs} = \{\big(\ \texttt{equal} :: \forall a.\{\texttt{Equal}\ a\} \Rightarrow a \to a \to \texttt{Bool}\ \big),$
$$\begin{pmatrix} \texttt{notequal} :: \forall a.\{\texttt{Equal}\ a\} \Rightarrow a \to a \to \texttt{Bool} \\ \texttt{notequal}\ x\ y = \texttt{not}(\texttt{equal}\ x\ y) \end{pmatrix}\}$$

Definition 2.21 (Instanzen) *Die Menge aller Instanzen* $\mathsf{I}(\mathsf{D},\mathsf{V})$ *ist die kleinste Teilmenge von* $\mathrm{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \times \mathsf{CC}(\mathsf{D},\mathsf{V}) \times \mathrm{Pot}(\mathsf{F}(\mathsf{D},\mathsf{V}))$*, welche nur Elemente der Form* $(\{c_1\ a_{i_1},\ldots,c_n\ a_{i_n}\}, c\ (c'\ a_1\ \ldots\ a_m), \underline{fs}) \in \mathsf{I}(\mathsf{D},\mathsf{V})$ *enthält, wenn*

- $a_1,\ldots,a_m \in \mathsf{V_T}$ *und alle paarweise verschieden,*
- $i_1,\ldots,i_n \in \{1,\ldots,m\}$*,*
- $c' \in \underline{\mathsf{TyCons}}_{\mathsf{D}}$*,*
- $\mathrm{arity}(c') \geq m$

gilt und $\underline{fs}$ *endlich ist. Die Typen* $a_{i_1},\ldots,a_{i_n}$ *der Klassenbedingungen dürfen nur die Variablen* $a_1,\ldots,a_n$ *sein, die auch in der Instanz* $c\ (c'\ a_1\ \ldots\ a_m)$ *vorkommen. Der Typ* $(c'\ a_1\ \ldots\ a_m)$ *der Instanz muß als Kopf direkt den Typkonstruktor* c' *enthalten. Dessen Parameter dürfen nur paarweise verschiedene Variablen sein. Außerdem muß der Typkonstruktor nicht immer vollständig mit Parametern befüllt sein (*$\mathrm{arity}(c') \geq m$*), um auch den Konstruktorklassen in Haskell gerecht zu werden. Z.B. ist* `Monad Maybe` *eine gültige Instanz der Klasse* `Monad`*, unabhängig davon, daß* `Maybe` *ein einstelliger Typkonstruktor ist.*

Im Beispiel 2.1 ist die Menge der Instanzen folgende:

$$\{(\emptyset, \texttt{Equal Nat}, \underline{fs}_1),$$

$$(\{\texttt{Equal}\ a\}, \texttt{Equal}\ (\texttt{List}\ a), \underline{fs}_2)\}$$

wobei

$$\underline{fs}_1 = \{ \begin{pmatrix} \texttt{equal} :: \emptyset \Rightarrow \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Bool} \\ \texttt{equal Zero Zero} \ = \ \texttt{True} \\ \vdots \end{pmatrix}, \begin{pmatrix} \texttt{notequal} :: \emptyset \Rightarrow \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Bool} \\ \texttt{notequal}\ x\ y \ = \ \texttt{not}(\texttt{equal}\ x\ y) \\ \vdots \end{pmatrix} \}$$

und

$$\underline{fs}_2 = \{ \begin{pmatrix} \texttt{equal} :: \forall a.\{\texttt{Equal}\ a\} \Rightarrow \texttt{List}\ a \rightarrow \texttt{List}\ a \rightarrow \texttt{Bool} \\ \texttt{equal Nil Nil} \ = \ \texttt{True} \\ \vdots \end{pmatrix}, \begin{pmatrix} \texttt{notequal} :: \forall a.\{\texttt{Equal}\ a\} \Rightarrow \texttt{List}\ a \rightarrow \texttt{List}\ a \rightarrow \texttt{Bool} \\ \texttt{notequal}\ x\ y \ = \ \texttt{not}(\texttt{equal}\ x\ y) \\ \vdots \end{pmatrix} \}$$

Jetzt wird die Menge der Programmkonstrukte eingeführt, um später die freien Variablen von diesen definieren zu können, welche für einige Reduktionen aus Kapitel 3 gebraucht werden.

Definition 2.22 (Programmkonstrukt) *Die Menge der Programmkonstrukte* $\mathsf{PC}(\mathsf{D},\mathsf{V})$ *ist definiert durch:*

$$\mathsf{PC}(\mathsf{D},\mathsf{V}) := \mathsf{H}(\mathsf{D},\mathsf{V}) \cup \mathsf{T}(\mathsf{D},\mathsf{V}) \cup \mathsf{P}(\mathsf{D},\mathsf{V}) \cup \mathsf{F}(\mathsf{D},\mathsf{V}) \cup \mathsf{C}(\mathsf{D},\mathsf{V}) \cup \mathsf{I}(\mathsf{D},\mathsf{V})$$

Definition 2.23 (Freie Variablen) *Die Menge der freien Variablen* $\mathsf{V}_{\mathsf{free}}(t)$ *eines Programmkonstrukts* t *ist wie folgt definiert:*

$$\left.\begin{array}{lcl} \mathsf{V}_{\mathsf{free}}(x) & := & \{x\} \\ \mathsf{V}_{\mathsf{free}}((t)) & := & \mathsf{V}_{\mathsf{free}}(t) \\ \mathsf{V}_{\mathsf{free}}((t_1\ t_2)) & := & \mathsf{V}_{\mathsf{free}}(t_1) \cup \mathsf{V}_{\mathsf{free}}(t_2) \end{array}\right\} \textit{Basis-Terme, Typen und Basispatterns}$$

$$\left.\begin{array}{lcl} \mathsf{V}_{\mathsf{free}}(\lambda p_1\ \ldots\ p_n \rightarrow t) & := & \mathsf{V}_{\mathsf{free}}(t) \setminus (\mathsf{V}_{\mathsf{free}}(p_1) \cup \ldots \cup \mathsf{V}_{\mathsf{free}}(p_n)) \\ \mathsf{V}_{\mathsf{free}}(\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2) & := & \mathsf{V}_{\mathsf{free}}(t) \cup \mathsf{V}_{\mathsf{free}}(t_1) \cup \mathsf{V}_{\mathsf{free}}(t_2) \\ \mathsf{V}_{\mathsf{free}}(\texttt{case}\ t\ \texttt{of}\ \underline{rs}) & := & \mathsf{V}_{\mathsf{free}}(t) \cup \mathsf{V}_{\mathsf{free}}(\underline{rs}) \\ \mathsf{V}_{\mathsf{free}}(\texttt{let}\ \underline{fs}\ \texttt{in}\ t) & := & \mathsf{V}_{\mathsf{free}}(t) \cup \bigcup_{f \in \underline{fs}} \mathsf{V}_{\mathsf{free}}(f) \end{array}\right\} \textit{Terme}$$

$$\left.\begin{array}{lcl} \mathsf{V}_{\mathsf{free}}(n) & := & \{\} \\ \mathsf{V}_{\mathsf{free}}(q) & := & \{\} \\ \mathsf{V}_{\mathsf{free}}(a) & := & \{\} \\ \mathsf{V}_{\mathsf{free}}(\_) & := & \{\} \\ \mathsf{V}_{\mathsf{free}}((x+n)) & := & \{x\} \\ \mathsf{V}_{\mathsf{free}}(x@p) & := & \{x\} \cup \mathsf{V}_{\mathsf{free}}(p) \end{array}\right\} \textit{Patterns}$$

$$\left.\mathsf{V}_{\mathsf{free}}(|\ t_1\ =\ t_2) \ := \ \mathsf{V}_{\mathsf{free}}(t_1) \cup \mathsf{V}_{\mathsf{free}}(t_2)\right\} \textit{Bedingte Regeln}$$

$$\left.\begin{array}{l} \mathsf{V}_{\mathsf{free}}((p_1 \ldots p_n, \underline{cr}_1 \ldots \underline{cr}_m, \underline{fs})) := (\mathsf{V}_{\mathsf{free}}(\underline{cr}_1) \cup \ldots \cup \mathsf{V}_{\mathsf{free}}(\underline{cr}_m) \\ \quad \cup \bigcup_{f \in \underline{fs}} \mathsf{V}_{\mathsf{free}}(f)) \setminus (\mathsf{V}_{\mathsf{free}}(p_1) \cup \ldots \cup \mathsf{V}_{\mathsf{free}}(p_n)) \end{array}\right\} \textit{Regeln}$$

$$\left.\mathsf{V}_{\mathsf{free}}((v, s, r_1 \ldots r_n)) := \mathsf{V}_{\mathsf{free}}(r_1) \cup \ldots \cup \mathsf{V}_{\mathsf{free}}(r_n)\right\} \textit{Funktionen}$$

$$\left.\mathsf{V}_{\mathsf{free}}((\underline{cs}, \underline{co}, \underline{fs})) := \bigcup_{f \in \underline{fs}} \mathsf{V}_{\mathsf{free}}(f)\right\} \textit{Klassen und Instanzen}$$

wenn

- $n \in \mathbb{N}$, $q \in \mathbb{Q}$, $a \in \mathbb{A}$, $x \in \mathsf{V}$,
- $t, t_1, t_2 \in \mathsf{H}(\mathsf{D},\mathsf{V})$,
- $p_1, \ldots, p_n \in \mathsf{P}(\mathsf{D},\mathsf{V})$,
- $\underline{cr}_1, \ldots, \underline{cr}_m \in \mathsf{CR}(\mathsf{D},\mathsf{V})$,
- $\underline{rs} \subseteq \mathsf{R}_1(\mathsf{D},\mathsf{V})$ *und* $\underline{fs} \subseteq \mathsf{F}(\mathsf{D},\mathsf{V})$

gilt.

Definition 2.24 (Substitutionen) *Eine Substitution*

$$\sigma : \mathsf{V} \longrightarrow \mathsf{H}(\mathsf{D},\mathsf{V}) \cup \mathsf{T}(\mathsf{D},\mathsf{V})$$

aus der Menge der Substitutionen $\mathsf{SUBS}(\mathsf{D},\mathsf{V})$ *bildet Variablen auf Programmkonstrukte ab. Wird eine Substitution* σ *auf ein Programmkonstrukt* $t \in \mathsf{PC}(\mathsf{D},\mathsf{V})$ *angewendet, wird darin jede freie Variable* $v \in \mathsf{V}_{\mathsf{free}}(t)$ *durch ihr Bild* $\sigma(v)$ *ersetzt. Die Substitution* σ *mit endlichem Domain (Variablen, die von einer Substitution ersetzt werden) kann durch*

$$\sigma := [x_1/t_1, \ldots, x_n/t_n]$$

beschrieben werden, wobei $x_i \in \mathsf{V}$ *und* $t_i \in \mathsf{PC}(\mathsf{D},\mathsf{V})$ *sind. Die Substitution* σ *ersetzt bei Anwendung die Variable* x_i *mit dem Term* t_i*.*

Bei der Auswertung des Haskellterms `contains` (`Cons Zero Nil`) (`Succ Zero`) muß die zweite Regel der Funktion $f_{\texttt{contains}}$ des Beispiels 2.1 angewendet werden. Dazu muß der Matcher zwischen dem Term `contains` (`Cons` x xs) y und dem oberen gefunden werden. Der Matcher ist die folgende Substitution:

$$[x/\texttt{Zero}, xs/\texttt{Nil}, y/(\texttt{Succ Zero})]$$

Nun wird das Haskellprogramm definiert, auf dem die späteren Analysen arbeiten werden.

Definition 2.25 (Haskellprogramm) *Ein Haskellprogramm* HP *ist der* 7*-Tupel*

$$\mathsf{HP} := \langle \mathsf{V}, \mathsf{N}, \mathsf{D}, \mathsf{C}, \mathsf{I}, \mathsf{F}, \text{schema} \rangle$$

wobei

- $\mathsf{V} :=$ *endliche Menge von Variablen, mit den Partitionen* V_T*,* V_F *und* V_L
- $\mathsf{N} :=$ *endliche Menge von Konstruktornamen,*
- $\mathsf{D} \in \mathsf{D}(\mathsf{N})$*, die Typdefinition,*
- $\mathsf{C} \subseteq \mathsf{C}(\mathsf{D},\mathsf{V})$*, die Klassen,*
- $\mathsf{I} \subseteq \mathsf{I}(\mathsf{D},\mathsf{V})$*, die Instanzen,*
- $\mathsf{F} \subseteq \mathsf{F}(\mathsf{D},\mathsf{V})$*, die Funktionen,*
- $\text{schema} : \underline{\mathsf{Cons}}_\mathsf{D} \cup \underline{\mathsf{TyCons}}_\mathsf{D} \longrightarrow \mathsf{S}(\mathsf{D},\mathsf{V})$

und $\mathsf{D},\mathsf{C},\mathsf{I},\mathsf{F}$ *endlich sind. Außerdem sind* V *und* N *disjunkt.*

Im Rahmen dieser Diplomarbeit wurde ein fast vollständiger Parser (das syntaktische Konstrukt der Record-Datentypen wurde nicht mit integriert) für Haskell, wie es im Haskell-98-Report [J+98] beschreiben ist, innerhalb des AProVE-Projekts ([GTSKF04]) angelegt, wobei die Implementierung sich stark an der von `Hugs` 98 [JR98] orientiert und das Modulsystem dem in [DJH02] beschriebenen ähnlich ist.

Der Typchecker prüft ein Haskellprogramm auf dessen Typkorrektheit, während er gleichzeitig jeden darin vorkommenden Term mit dessen konkretem Typ annotiert. Im folgenden wird implizit von jedem vorkommenden Haskellprogramm angenommen, daß es, wenn nicht anders beschrieben, typkorrekt und vollständig annotiert ist.

Definition 2.26 (Typannotationen) *Jeder Term t innerhalb des Haskellprogramms* HP *wird durch den Typchecker mit seinem Typ τ wie folgt annotiert:*

$$t_{|\tau|}$$

Die Typannotationen werden weggelassen, wenn sie keine Relevanz für den Kontext haben und sich die Typen implizit ergeben.

Die Funktion $f_{\texttt{length}}$ aus Beispiel 2.1 ist:

$$\begin{aligned} f_{\texttt{length}} =& (\texttt{length}, \emptyset \Rightarrow \texttt{List}\ a \to \texttt{Nat}, \underline{rs}) \\ =& \begin{pmatrix} \texttt{length} :: \texttt{List}\ a \to \texttt{Nat} \\ \texttt{length}\ \texttt{Nil} = \texttt{Zero} \\ \texttt{length}\ (\texttt{Cons}\ x\ xs) = \texttt{Succ}\ (\texttt{length}\ xs) \end{pmatrix} \end{aligned}$$

Voll annotiert ergibt sich für sie:

$$\begin{aligned} f_{\texttt{length}} &= (\texttt{length}_{|\texttt{List}\ a \to \texttt{Nat}|}, \emptyset \Rightarrow \texttt{List}\ a \to \texttt{Nat}, \underline{rs}) \\ &= \begin{pmatrix} \texttt{length}_{|\texttt{List}\ a \to \texttt{Nat}|} :: \texttt{List}\ a \to \texttt{Nat} \\ \texttt{length}_{|\texttt{List}\ a \to \texttt{Nat}|}\ \texttt{Nil}_{|\texttt{List}\ a|} = \texttt{Zero}_{|\texttt{Nat}|} \\ \texttt{length}_{|\texttt{List}\ a \to \texttt{Nat}|}\ (\texttt{Cons}_{|a \to \texttt{List}\ a \to \texttt{List}\ a|}\ x_{|a|}\ xs_{|\texttt{List}\ a|})_{|\texttt{List}\ a|} = \\ \qquad (\texttt{Succ}_{|\texttt{Nat} \to \texttt{Nat}|}\ (\texttt{length}_{|\texttt{List}\ a \to \texttt{Nat}|}\ xs_{|\texttt{List}\ a|})_{|\texttt{Nat}|})_{|\texttt{Nat}|} \end{pmatrix} \end{aligned}$$

Das gleiche gilt für Member-Funktionen und deren Member-Variablen innerhalb von Klassen und Instanzen.

Zusätzlich soll gelten, daß eine Substitution, die auf einen Term angewendet wird, gleichzeitig auch auf dessen Typannotationen angewendet wird. Dies ist möglich,

weil die Typvariablen $\mathsf{V_T}$ disjunkt zu den Variablen $\mathsf{V_F} \uplus \mathsf{V_L}$ der Terme sind. Beispielsweise ergibt sich für den Term

$$\mathtt{length}_{|\mathtt{List}\ a \to \mathtt{Nat}|}\ xs_{|\mathtt{List}\ a|}$$

nach der Anwendung der Substitution $\sigma := [a/\mathtt{Nat}]$ folgender Term:

$$(\mathtt{length}_{|\mathtt{List}\ a \to \mathtt{Nat}|}\ xs_{|\mathtt{List}\ a|})\sigma = \mathtt{length}_{|\mathtt{List\ Nat} \to \mathtt{Nat}|}\ xs_{|\mathtt{List\ Nat}|}$$

Ein Typchecker für Haskell, der die vom Parser gelieferten Haskellprogramme überprüft, wurde ebenfalls im AProVE-Projekt implementiert, wobei das vollständige Typsystem, wie es im Haskell-98-Report([J+98]) mit Klassen und Konstruktorklassen beschrieben ist, unterstützt wird.

Kapitel 3

Reduktionen

In diesem Kapitel werden die neuen Reduktionen, welche semantikerhaltend auf ein Haskellprogramm angewendet werden können, im einzelnen vorgestellt. Sie dienen dazu, Haskellprogramme in eine strukturell einfachere Form zu bringen, die der späteren Terminierungsanalyse entgegenkommt. Grundsätzlich werden die besonderen Arten von Patterns in ihre äquivalenten Basisformen übersetzt und alle Haskell-Sprachkonstrukte auf Funktionen zurückgespielt, so daß am Ende ein Haskellprogramm nur noch bedingungsfreie Funktionen aus Basispatterns und Basistermen enthält. Um diese Normalform zu erreichen, spielt die Reihenfolge der Reduktionsschritte keine Rolle, solange die jeweiligen Vorbedingungen der eingesetzten Reduktionen eingehalten werden. Eine Reihenfolge, welche die Vorbedingungen berücksichtigt, wird am Ende des Kapitels vorgestellt.

Bevor die Reduktionen im einzelnen erklärt werden, benötigen wir hier noch eine Beschreibung der Semantik von Haskell, um später die Semantikerhaltung der einzelnen Reduktionen zeigen zu können. Die Semantik von Haskell basiert auf der Semantik des `case`-Konstrukts wie sie im Haskell-98-Report([J+98]) definiert und in den Abbildungen 3.1 und 3.2 angegeben ist. Die folgenden Übersetzungen sind laut dem Haskell-98-Report ebenfalls semantikerhaltend:

- λ-Übersetzung (λ):

  $\lambda\ p_1\ \ldots\ p_n \rightarrow\ e = \lambda\ x_1\ \ldots\ x_n \rightarrow \texttt{case}\ (x_1, \ldots, x_n)\ \texttt{of}\ \{(p_1, \ldots, p_n) \rightarrow\ e\}$

  wobei $p_1, \ldots, p_n \in \mathsf{P}(\mathsf{D}, \mathsf{V})$, $e \in \mathsf{H_B}(\mathsf{D}, \mathsf{V})$ und $x_1, \ldots, x_n \in \mathsf{V_L}$

- `if`-Übersetzung (I):

  $$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = \texttt{case}\ e_1\ \texttt{of}\ \{\texttt{True} \rightarrow\ e_2\ ;\ \texttt{False} \rightarrow\ e_3\}$$

  wobei $e_1, e_2, e_3 \in \mathsf{H_B}(\mathsf{D}, \mathsf{V})$

(a) `case` e `of { ` $alts$ `}` $=$ `(\`v `->` `case` v `of { ` $alts$ `})` e
where v is a completely new variable

(b) `case` v `of { ` p_1 $match_1$`;` $\ldots$ `;` p_n $match_n$ `}`
$=$ `case` v `of { ` p_1 $match_1$ `;`
`_ ->` $\ldots$ `case` v `of {`
p_n $match_n$
`_ -> error "No match" }...}`
where each $match_i$ has the form:
`|` $g_{i,1}$ `->` $e_{i,1}$ `;` $\ldots$ `;` `|` g_{i,m_i} `->` e_{i,m_i} `where { ` $decls_i$ `}`

(c) `case` v `of { ` p `|` g_1 `->` e_1 `;` $\ldots$
`|` g_n `->` e_n `where { ` $decls$ `}`
`_ ->` e' `}`
$=$ `case` e' `of`
`{`y `->` (where y is a completely new variable)
`case` v `of {`
p `->` `let { ` $decls$ `} in`
`if` g_1 `then` e_1 $\ldots$ `else if` g_n `then` e_n `else` y
`_ ->` y `}}`

(d) `case` v `of { ~`p `->` e`; _ ->` e' `}`
$=$ `(\`$x'_1 \ldots x'_n$ `->` e_1 `) (case` v `of { ` p`->` x_1 `}) ... (case` v `of { ` p `->` x_n`})`
where $e_1 = e\,[x'_1/x_1, \ldots, x'_n/x_n]$
$x_1, \ldots, x_n$ are all the variables in p; $x'_1, \ldots, x'_n$ are completely new variables

(e) `case` v `of { ` x`@`p `->` e`; _ ->` e' `}`
$=$ `case` v `of { ` p `->` `( \` x `->` e `)` v `; _ ->` e' `}`

(f) `case` v `of { _ ->` e`; _ ->` e' `}` $= e$

Abbildung 3.1: Semantik des `case`-Konstrukts, Teil 1, Kopie aus [J+98], Seite 31

- Funktions-Übersetzung (F):

$$\left(\begin{array}{l} v :: s \\ v\ p_{1,1}\ \ldots\ p_{1,m}\ \underline{cl}_1\ \texttt{where}\ \underline{fs}_1 \\ \vdots \\ v\ p_{n,1}\ \ldots\ p_{n,m}\ \underline{cl}_n\ \texttt{where}\ \underline{fs}_1 \end{array}\right)$$

$$= \left(\begin{array}{l} v\ \ :: s \\ v = \lambda\, x_1\ \ldots\ x_m \rightarrow \\ \qquad \texttt{case}\ (x_1, \ldots, x_m)\ \texttt{of}\ \{\, (p_{1,1}, \ldots, p_{1,m})\ \underline{cl}_1\ \texttt{where}\ \underline{fs}_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad (p_{n,1}, \ldots, p_{n,m})\ \underline{cl}_n\ \texttt{where}\ \underline{fs}_n\} \end{array}\right)$$

wobei $p_{1,1}, \ldots, p_{n,m} \in \mathsf{P(D,V)}$, $\underline{cl}_1, \ldots, \underline{cl}_n \in \mathsf{CR(D,V)}$
und $\underline{fs}_1, \ldots, \underline{fs}_n \in \mathrm{Pot}(\mathsf{F(D,V)})$

(g) `case` v `of { ` $K\ p_1 \ldots p_n$ `-> ` e`; _ -> ` e' `}`
$=$ `case` v `of {`
$K\ x_1 \ldots x_n$ `-> case` x_1 `of {`
p_1 `-> ...` `case` x_n `of { ` p_n `-> ` e `; _ -> ` e' `} ...`
`_ -> ` e' `}`
`_ -> ` e' `}`
at least one of $p_1, \ldots, p_n$ is not a variable; $x_1, \ldots, x_n$ are new variables

(h) `case` v `of { ` k `-> ` e`; _ -> ` e' `}` $=$ `if (`v`==`k`) then` e `else` e'

(i) `case` v `of { ` x `-> ` e`; _ -> ` e' `}` $=$ `case` v `of { ` x `-> ` e `}`

(j) `case` v `of { ` x `-> ` e `}` $=$ `( \ ` x `-> ` e `)` v

(k) `case` $N\ v$ `of { ` $N\ p$ `-> ` e`; _ -> ` e' `}`
$=$ `case` v `of { ` p `-> ` e`; _ -> ` e' `}`
where N is a `newtype` constructor

(l) `case` $\bot$ `of { ` $N\ p$ `-> ` e`; _ -> ` e' `}` $=$ `case` $\bot$ `of { ` p `-> ` e `}`
where N is a `newtype` constructor

(m) `case` v `of { ` K `{ ` f_1 `=` p_1 `, ` f_2 `=` p_2 `, ...} -> ` e `; _ -> ` e' `}`
$=$ `case` e' `of {`
y `->`
`case` v `of {`
K `{ ` f_1 `=` p_1 `} ->`
`case` v `of { ` K `{ ` f_2 `=` p_2 `, ... } -> ` e `; _ -> ` y `};`
`_ -> ` y `}}`
where f_1, f_2, ... are fields of constructor K; y is a new variable

(n) `case` v `of { ` K `{ ` f `=` p `} -> ` e `; _ -> ` e' `}`
$=$ `case` v `of {`
$K\ p_1 \ldots p_n$ `-> ` e `; _ -> ` e' `}`
where p_i is p if f labels the ith component of K, `_` otherwise

(o) `case` v `of { ` K `{} -> ` e `; _ -> ` e' `}`
$=$ `case` v `of {`
K `_ ... _ -> ` e `; _ -> ` e' `}`

(p) `case (`$K'\ e_1 \ldots e_m$`) of { ` $K\ x_1 \ldots x_n$ `-> ` e`; _ -> ` e' `}` $= e'$
where K and K' are distinct `data` constructors of arity n and m, respectively

(q) `case (`$K\ e_1 \ldots e_n$`) of { ` $K\ x_1 \ldots x_n$ `-> ` e`; _ -> ` e' `}`
$=$ `case` e_1 `of { ` x'_1 `-> ... case` e_n `of { ` x'_n `->` $e[x'_1/x_1 \ldots x'_n/x_n]$ `}...}`
where K is a constructor of arity n; $x'_1 \ldots x'_n$ are completely new variables

(r) `case` e_0 `of { ` x`+`k `-> ` e`; _ -> ` e' `}`
$=$ `if` e_0 `>=` k `then let {`x' `=` e_0`-`k`} in` $e[x'/x]$ `else` e' (x' is a new variable)

Abbildung 3.2: Semantik des `case`-Konstrukts, Teil 2, Kopie aus [J+98], Seite 33

- `let`-Übersetzung (L):

$$\texttt{let}\ \{p = e_1\}\ \texttt{in}\ e_0 = \texttt{case}\ e_0\ \texttt{of}\ \{\sim p\ \rightarrow\ e_1\}$$

  wobei $p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$ und $e_0, e_1 \in \mathsf{H}_\mathsf{B}(\mathsf{D}, \mathsf{V})$

Die Übersetzungen sind im Haskell-98-Report ([J+98]) auf den Seiten 16, 18, 22 und 53 bis 54 zu finden sind. Zusätzlich ist in Haskell die β-Übersetzung

$$(\lambda x\ \rightarrow e_0)\ e_1 = e_0[x/e_1]$$

mit $e_0, e_1 \in \mathsf{H}_\mathsf{B}(\mathsf{D}, \mathsf{V})$ und $x \in \mathsf{V}_\mathsf{L}$ gültig. Die Gleichungen der Übersetzungen lassen sich mit den Gleichungen aus Abbildung 3.1 und 3.2 zu der Kongruenzrelation $\equiv$ zusammenfassen, welche so der semantischen Äquivalenz von Haskell entspricht. Zu dieser semantischen Äquivalenz werden nun einige Lemmata gezeigt, um später die Beweise der Semantikerhaltung der neuen Reduktionen einfacher gestalten zu können.

Lemma 3.1 (`let`-Umbenennungen (lU)) *Seien*

- $t, u \in \mathsf{H}(\mathsf{D}, \mathsf{V})$,
- $v \in \mathsf{V}_\mathsf{F}$,
- $v \notin \mathsf{V}_{\mathsf{free}}(u)$ *und*
- $\sigma = [v/u]$

dann gilt:

$$\texttt{let}\ \{v\ =\ u\}\ \texttt{in}\ t \equiv t\sigma$$

Beweis:

$$\begin{aligned}
\texttt{let}\ \{v\ =\ u\}\ \texttt{in}\ t &\overset{(L)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{\sim v\ \rightarrow\ t\} \\
&\overset{(d)}{\equiv} (\lambda\, y\ \rightarrow\ t[v/y])\ (\texttt{case}\ u\ \texttt{of}\ \{v\ \rightarrow\ v\}) \\
&\overset{(j)}{\equiv} (\lambda\, y\ \rightarrow\ t[v/y])\ ((\lambda v\ \rightarrow\ v)\ u) \\
&\overset{(\beta)}{\equiv} (\lambda\, y\ \rightarrow\ t[v/y])\ u \\
&\overset{(\beta)}{\equiv} t[v/y][y/u] \\
&\equiv t[v/u] \\
&\equiv t\sigma
\end{aligned}$$

wobei t und v wie oben gegeben sind. □

Lemma 3.2 (`let`-Erweiterung(lE)) *Sei* $t \in \mathsf{H}(\mathsf{D},\mathsf{V})$, $v \in \mathsf{V_F}$ *und* $v \notin \mathsf{V_{free}}(t)$ *dann gilt:*

$$\texttt{let}\ \{v\ =\ t\}\ \texttt{in}\ v \equiv t$$

Jeder Haskellterm kann in einen `let`*-Ausdruck wie oben eingeschlossen werden. Beweis:*

$$\begin{aligned}
\texttt{let}\ \{v\ =\ t\}\ \texttt{in}\ v &\overset{(L)}{\equiv} \texttt{case}\ t\ \texttt{of}\ \{{\sim}v\ \rightarrow\ v\} \\
&\overset{(d)}{\equiv} (\lambda\ y\ \rightarrow\ y)(\texttt{case}\ t\ \texttt{of}\ \{v\ \rightarrow\ v\}) \\
&\overset{(\beta)}{\equiv} \texttt{case}\ t\ \texttt{of}\ \{v\ \rightarrow\ v\} \\
&\overset{(j)}{\equiv} (\lambda\ v\ \rightarrow\ v)t \\
&\overset{(\beta)}{\equiv} t
\end{aligned}$$

wobei t *und* v *wie oben gegeben sind.* □

Lemma 3.3 (`let`-Substitution(lSUB)) *Seien*

- $v \in \mathsf{V_F}$,
- $t \in \mathsf{H}(\mathsf{D},\mathsf{V})$,
- $f, g \in \mathsf{F}(\mathsf{D},\mathsf{V})$ *und*
- $s \in \mathsf{S}(\mathsf{D},\mathsf{V})$
- $\sigma = [v/t]$

wobei die Funktion f *der Form*

$$f = \begin{pmatrix} v :: s \\ v = t \end{pmatrix}$$

ist, so gilt:

$$\{\ldots; f; g; \ldots\} \equiv \{\ldots; f; g\sigma; \ldots\}$$

Dieses Lemma besagt, daß die Variable v*, die durch die Funktion* f *an den Term* t *gebunden ist, innerhalb der Funktion* g *durch die Substitution* σ *ersetzt werden kann, ohne die Semantik zu ändern. Das gilt insbesondere auch, wenn* f *und* g *verschränkt rekursiv sind.*

Der Beweis kann geführt werden, indem die Rekursionsgleichungen abgerollt werden. Dabei stellt man fest, daß $g\sigma$ *schneller konvergiert als* g*, aber sonst äquivalent ist. Wegen seiner Größe wird der Beweis hier weggelassen.*

Lemma 3.4 (`let`-Lift(lL)) *Sei* $t \in \mathsf{H}(\mathsf{D},\mathsf{V})$ *und seien*

$$f_1,\ldots,f_l,g_1,\ldots,g_m,h_1,\ldots,h_n \in \mathsf{F}(\mathsf{D},\mathsf{V})$$

wobei die Funktionen $f_1,\ldots,f_l$ *nicht von den Funktionen* $g_1,\ldots,g_m$ *benutzt werden und jede Funktion* f_i *der Form*

$$f_i = \left(\begin{array}{l} v_i :: s_i \\ v_i = \lambda\ p_{i,1}\ \ldots\ p_{i,k_i}\ \rightarrow\ t_i \end{array}\right)$$

ist. Zu den Funktionen $f_1,\ldots,f_l$ *gibt es außerdem die Funktionen* $f'_1,\ldots,f'_l \in \mathsf{F}(\mathsf{D},\mathsf{V})$ *der Form*

$$f'_i = \left(\begin{array}{l} v'_i :: s'_i \\ v'_i = \lambda\ x_1\ \ldots\ x_j\ p_{i,1}\ \ldots\ p_{i,k_i}\ \rightarrow\ t_i\sigma \end{array}\right)$$

wobei

- $v'_i \notin \bigcup_{i=1}^{l} \mathsf{V}_{\mathsf{free}}(f_i) \cup \bigcup_{i=1}^{m} \mathsf{V}_{\mathsf{free}}(g_i) \cup \bigcup_{i=1}^{n} \mathsf{V}_{\mathsf{free}}(h_i)$
  (v'_i kommt nicht frei in den Funktionen $f_1,\ldots,f_l,g_1,\ldots,g_m,h_1,\ldots,h_n$ *vor),*
- $\{x_1,\ldots,x_j\} = \mathsf{V}_{\mathsf{free}}(f_1) \cup \cdots \cup \mathsf{V}_{\mathsf{free}}(f_l) \setminus \mathsf{V}_{\mathsf{F}}$ *und*
- $\sigma = [v_1/(v'_1\ x_1\ \ldots\ x_j),\ldots,v_l/(v'_l\ x_1\ \ldots\ x_j)]$
- $s_i, s'_i \in \mathsf{S}(\mathsf{D},\mathsf{V})$

ist. Dann gelten

$$\begin{aligned} &\texttt{let}\ \{h_1;\ldots;h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1;\ldots;f_l;g_1;\ldots;g_m\}\ \texttt{in}\ t\big] \\ &\equiv \texttt{let}\ \{h_1;\ldots;h_n;f'_1;\ldots;f'_l\}\ \texttt{in}\ C\big[\texttt{let}\ \{g_1\sigma;\ldots;g_m\sigma\}\ \texttt{in}\ t\sigma\big] \end{aligned} \tag{1}$$

und

$$\begin{aligned} &\{h_1;\ldots;h_n;v\ \ldots\ =\ C\big[\texttt{let}\ \{f_1;\ldots;f_l;g_1;\ldots;g_m\}\ \texttt{in}\ t\big]\} \\ &\equiv \{h_1;\ldots;h_n;f'_1;\ldots;f'_l,v\ \ldots\ = C\big[\texttt{let}\ \{g_1\sigma;\ldots;g_m\sigma\}\ \texttt{in}\ t\sigma\big]\} \end{aligned} \tag{2}$$

wenn $C\big[\ldots\big]$ *ein* `let`*-Konstrukt-freier Termkontext ist.*

Die beiden Gleichung (1) und (2) sagen aus, daß die möglicherweise verschränkt rekursiven Funktionen $f_1,\ldots,f_l$ *aus einem tieferen Block in den nächst höheren geschoben werden können, auch wenn diese Variablen benutzen, die durch* λ*-Abstraktionen aus dem umschliessenden Termkontext* $C\big[\ldots\big]$ *gebunden sind. Die Funktionen* $f_1,\ldots,f_l$ *müssen dazu leicht verändert werden, wie es die Funktionen* $f'_1,\ldots,f'_l$ *beschreiben. Zusätzlich müssen die Aufrufe der Funktion* $f_1,\ldots,f_l$

innerhalb von $g_1, \ldots, g_m$ durch σ auf die Funktionen $f'_1, \ldots, f'_l$ umgeleitet werden.

Beweis von (1):

$$
\begin{aligned}
&\texttt{let}\ \{h_1; \ldots; h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1; \ldots; f_l; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(*1)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^a; \ldots; f_l^a; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(*2)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^b; \ldots; f_l^b; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(*3)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^b; \ldots; f_l^b; f_1''; \ldots; f_l''; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(*4)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^c; \ldots; f_l^c; f_1''; \ldots; f_l''; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(*5)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^c; \ldots; f_l^c; f_1'; \ldots; f_l'; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(*6)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n; f_1'; \ldots; f_l'\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^c; \ldots; f_l^c; g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\overset{(lU)}{\equiv} \texttt{let}\ \{h_1; \ldots; h_n; f_1'; \ldots; f_l'\}\ \texttt{in}\ C\big[\texttt{let}\ \{f_1^c\}\ \texttt{in}\ \ldots \texttt{let}\ \{f_m^c\}\ \texttt{in} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \{g_1; \ldots; g_m\}\ \texttt{in}\ t\big] \\
&\equiv \texttt{let}\ \{h_1; \ldots; h_n; f_1'; \ldots; f_l'\}\ \texttt{in}\ C\big[\texttt{let}\ \{g_1\sigma; \ldots; g_m\sigma\}\ \texttt{in}\ t\sigma\big]
\end{aligned}
$$

wobei

- $f_i = \begin{pmatrix} v_i :: s_i \\ v_i = \lambda\ p_{i,1}\ \ldots\ p_{i,k_i}\ \rightarrow\ t_i \end{pmatrix}$
- $f_i^a = \begin{pmatrix} v_i :: s_i \\ v_i = (\lambda\ x_1\ \ldots\ x_j\ \rightarrow \lambda\ p_{i,1}\ \ldots\ p_{i,k_i}\ \rightarrow\ t_i)\ x_1\ \ldots\ x_j \end{pmatrix}$
- $f_i^b = \begin{pmatrix} v_i :: s_i \\ v_i = (\lambda\ x_1\ \ldots\ x_j\ p_{i,1}\ \ldots\ p_{i,k_i}\ \rightarrow\ t_i)\ x_1\ \ldots\ x_j \end{pmatrix}$
- $f_i^c = \begin{pmatrix} v_i :: s_i \\ v_i = v'_i\ x_1\ \ldots\ x_j \end{pmatrix}$
- $f_i'' = \begin{pmatrix} v'_i :: s'_i \\ v'_i = \lambda\ x_1\ \ldots\ x_j\ p_{i,1}\ \ldots\ p_{i,k_1}\ \rightarrow\ t_i \end{pmatrix}$
- $f_i' = \begin{pmatrix} v'_i :: s'_i \\ v'_i = \lambda\ x_1\ \ldots\ x_j\ p_{i,1}\ \ldots\ p_{i,k_1}\ \rightarrow\ t_i\sigma \end{pmatrix}$

und

$$f_i \overset{(\beta)}{\equiv} f_i^a \tag{$*1$}$$

$$f_i^a \equiv f_i^b \tag{$*2$}$$

$$\{\ldots, f_i'', f_i^b, \ldots\} \overset{(lSUB)}{\equiv} \{\ldots, f_i'', f_i^c, \ldots\} \tag{$*4$}$$

$$\{\ldots, f_i^c, f_u'', \ldots\} \overset{(lSUB)}{\equiv} \{\ldots, f_i^c, f_i', \ldots\} \tag{$*5$}$$

*Zu (*3): Hinzufügen von den neuen Funktionen $f_1'', \ldots, f_l''$ verändert die Semantik nicht, da die Funktionsvariablen $v_1', \ldots, v_l'$ nicht in den Funktionen $f_1, \ldots, f_l$, $g_1, \ldots, g_m$, $h_1, \ldots, h_n$ auftreten.*
In Gleichung ($*4$) wird die Substitution $[v_i'/\lambda\ x_1\ \ldots\ x_j\ p_{i,1}\ \ldots\ p_{i,k_1}\ \rightarrow\ t_i]$, welche durch f_i'' definiert wird, auf f_i^b einmal invers angewendet und man erhält f_i^c. Dies ermöglicht, in Gleichung ($*5$) die Substitution σ zu benutzen, da diese nun durch die Funktionen $f_1^c, \ldots, f_l^c$ zur Verfügung steht und so Lemma 3.3 ($lSUB$) erneut anwendbar ist. Gleichung ($*6$) gilt, weil die Funktionen $f_1', \ldots, f_l'$ nicht von den Funktion $f_1^c, \ldots, f_l^c, g_1, \ldots, g_m$ oder dem Kontext C abhängig sind, wegen $\mathsf{V}_{\mathsf{free}}(f_i') = \emptyset$. □

Beweis von (2): Dieser Beweis ist analog zu dem von Gleichung (1). □

3.1 _-Reduktion

Jedes Joker-Pattern _ kann durch eine frische lokale Variable ersetzt werden. Es kann nicht zurückgewiesen werden, genau wie eine Variable, aber es bindet im Gegensatz zu dieser keine Werte an Variablen, deshalb muß bei der Ersetzung in jedem Fall eine frische Variable verwendet werden. Die _-Reduktion ist definiert durch:

$$\frac{\_}{x}$$

wobei $x \in \mathsf{V}_{\mathsf{L}}$ und x eine frische Variable sein muß.

Satz 3.1 *Die _-Reduktion erhält die Semantik.*

Beweis:
Sei $e, t, u \in \mathsf{H}(\mathsf{D}, \mathsf{V})$ und $x \notin \mathsf{V}_{\mathsf{free}}(t)$ so folgt

$$\begin{aligned}
\texttt{case}\ u\ \texttt{of}\ \{_\ \rightarrow\ t;\ _\ \rightarrow\ e\} &\overset{(a)}{\equiv} (\lambda\, x\ \rightarrow\ \texttt{case}\ v\ \texttt{of}\ \{_\ \rightarrow\ t;\ _\ \rightarrow\ e\})\ u \\
&\overset{(f)}{\equiv} (\lambda\, x\ \rightarrow\ t)\ u \\
&\overset{(j)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{x\ \rightarrow\ t\}
\end{aligned}$$

Da jedes Joker-Pattern letztlich auf $\texttt{case}\ u\ \texttt{of}\ \{\_\ \rightarrow\ t;\ \_\ \rightarrow\ e\}$ *reduziert wird ist der obere Beweis ausreichend.* □

3.2 $x@p$-Reduktion

Das $x@p$-Pattern ermöglicht den Zugriff auf den Wert, der gegen das Pattern p gematcht wird, durch die Variable x. Innerhalb der λ-Abstraktion des Terms

$$(\lambda\, x@(\texttt{Cons}\ y\ \texttt{Nil}) \ \rightarrow\ (x, y))(\texttt{Cons}\ 1\ \texttt{Nil})$$

kann zum Beispiel mit x auf den Wert `Cons 1 Nil` zugegriffen werden, der gegen das Pattern `Cons` y `Nil` gematcht wurde, so daß wir bei einer Auswertung des Terms den Term (`Cons 1 Nil`, `1`) erhalten. Wenn p kein $\sim p$-, _-, $(x + n)$- oder $x@p$-Pattern enthält, kann die Variable x einfach durch das Pattern p im Ergebnisterm der Regel ersetzt werden, ohne daß das Pattern $x@p$ benötigt wird. Diese Idee benutzt die $x@p$-Reduktion, die folgendermaßen definiert ist:

$$\frac{(p_1 \ldots C\big[x@p\big] \ldots p_m, \underline{cs}, \underline{fs})}{(p_1 \ldots C\big[p\big] \ldots p_m, \underline{cs}\sigma, \underline{fs}\sigma)}$$

wobei

- $(p_1 \ldots C\big[x@p\big] \ldots p_m, \underline{cs}, \underline{fs}) \in \mathsf{R}(\mathsf{D}, \mathsf{V})$,
- $x \in \mathsf{V_L}$,
- $p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$,
- p enthält kein _-, $(x + n)$- oder $x@p$-Pattern und
- $\sigma = [x/p]$

gilt. Außerdem wird o.B.d.A angenommen, daß alle Variablen aus $\mathsf{V_{free}}(p_i)$ nicht noch einmal durch andere Patterns gebunden werden in $\underline{cs}$ und $\underline{fs}$. Dieser Zustand ist durch geeignete Umbennungen immer erreichbar. Zu Beachten ist, daß die $x@p$-Reduktion bei verschachtelten $x@p$-Pattern erst nur auf das innerste angewendet werden kann, weil nur dort die Nebenbedingung erfüllt ist, daß es keine weiteren $\sim p$-, _-, $(x + n)$- oder $x@p$-Pattern enthält.

Satz 3.2 *Die $x@p$-Reduktion ist semantikerhaltend.*

Beweis:

$$\begin{aligned}
\texttt{case}\ u\ \texttt{of}\ \{x@p\ \rightarrow\ t;\ _\ \rightarrow\ e\} &\overset{(e)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{p\ \rightarrow\ (\lambda\, x \rightarrow\ t)u;\ _\ \rightarrow\ e\} \\
&\overset{(*)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{p\ \rightarrow\ (\lambda\, x \rightarrow\ t)p;\ _\ \rightarrow\ e\} \\
&\overset{(\beta)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{p\ \rightarrow\ t[x/p];\ _\ \rightarrow\ e\} \\
&\equiv \texttt{case}\ u\ \texttt{of}\ \{p\ \rightarrow\ t\sigma;\ _\ \rightarrow\ e\}
\end{aligned}$$

Zu Schritt $(*)$*:*
Seien $y_1 \dots y_n \in \mathsf{V}(p)$ *mit* $p|_{\pi_i} = y_i$*. Wenn das Pattern* p *den Term* u *matcht, existiert eine Substitution* $\mu = [y_1/u|_{\pi_1}, \dots, y_n/u|_{\pi_n}]$*, für die* $p\mu = u$ *gilt, weil* p *kein* `_`*-,* $(x + n)$*- oder* x`@`p*-Pattern enthält. Die Substitution* μ *wird bei der Auswertung des Terms* `case` u `of` $\{p \to (\lambda\, x \to t)u\}$ *auf den Teilterm* $(\lambda\, x \to t)u$ *angewendet, so daß dieser durch den Term* $(\lambda\ x \to\ t)p$ *ersetzt werden kann, unter der Annahme, daß die Variablen aus* p *nicht durch andere Patterns in* t *erneut gebunden werden.*

Da jedes x`@`p*-Pattern letztlich auf* `case` u `of` $\{x$`@`$p \to t;$ `_` $\to e\}$ *reduziert wird, ist der obere Beweis ausreichend.* □

3.3 $\sim p$-Reduktion

Das $\sim p$-Pattern ist ein unabweisbares Pattern, das heißt, es matcht immer auf den Eingabeterm, unabhängig davon, ob das Subpattern p wirklich matcht. Die Variablen werden, wenn das Subpattern matcht, wie üblich gebunden. Im Falle, daß das Subpattern nicht matcht, werden dessen Variablen an $\bot$ gebunden. Um diese Semantik zu erhalten, ersetzt die $\sim p$-Reduktion jedes $\sim p$-Pattern durch eine neue Variable, welche ebenfalls ein unabweisbares Pattern ist. Die Variablen aus dem Subpattern werden in dem Ergebnisterm durch jeweils einen Aufruf ihrer zugehörigen Funktionen ersetzt, welche versuchen, das Subpattern doch zu binden und, wenn dies gelingt, das Ergebnis einer Variablenbindung zurückzuliefern. Ansonsten liefern diese Funktionen $\bot$ als Ergebnis. Formal ist die $\sim p$-Reduktion wie folgt definiert:

$$\frac{(p_1 \dots C\llbracket \sim p \rrbracket \dots p_m, \underline{cs}, \underline{fs})}{(p_1 \dots C\llbracket y \rrbracket \dots p_m, \underline{cs}\sigma, \underline{fs}\sigma)}$$

wobei

- $(p_1 \dots C\llbracket \sim p \rrbracket \dots p_m, \underline{cs}, \underline{fs}) \in \mathsf{R}(\mathsf{D}, \mathsf{V})$,
- C ist ein Pattern-Kontext,
- $p_1, \dots, p_m \in \mathsf{P}(\mathsf{D}, \mathsf{V})$,
- $p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$,
- y eine frische Variable aus $\mathsf{V_L}$ ist,
- $v_1, \dots, v_n$ frische Variablen aus $\mathsf{V_F}$ sind,

- $\sigma = [x_1/(v_1\ y), \ldots, x_n/(v_n\ y)]$ und
- $\{x_1, \ldots, x_n\} = \mathsf{V}(p)$

gilt und die folgenden Funktionen $f_1, \ldots, f_n$ in denselben Block aufgenommen werden müssen:

$$f_i := \left(\ v_i\ p \ = \ x_i \ \right)$$

Beispiel 3.1 *Der folgende Teil eines Haskellprogramms*

$\texttt{f}\ \sim(\texttt{Cons}\ a\ \texttt{Nil}, \texttt{Just}\ b) = (a, b)$

wird durch die $\sim$p-Reduktion durch den folgenden Teil

$\texttt{f}\ y\ =\ (\texttt{f1}\ y, \texttt{f2}\ y)$
$\texttt{f1}\ (\texttt{Cons}\ a\ \texttt{Nil}, \texttt{Just}\ b)\ =\ a$
$\texttt{f2}\ (\texttt{Cons}\ a\ \texttt{Nil}, \texttt{Just}\ b)\ =\ b$

ersetzt.

Satz 3.3 *Die $\sim$p-Reduktion erhält die Semantik.*

Beweis:

$$
\begin{aligned}
&\texttt{case}\ u\ \texttt{of}\ \{\sim p\ \to\ t;\ \_\ \to\ e\} \\
&\overset{(a)}{\equiv}\ (\lambda y\ \to\ \texttt{case}\ x\ \texttt{of}\ \{\sim p\ \to\ t;\ \_\ \to\ e\})\ u \\
&\overset{(d)}{\equiv}\ (\lambda y\ \to\ ((\lambda\ y_1 \ldots y_n\ \to\ t[x_1/y_1, \ldots, x_n/y_n]) \\
&(\texttt{case}\ x\ \texttt{of}\ \{p\ \to\ x_1\})\ \ldots\ (\texttt{case}\ x\ \texttt{of}\ \{p\ \to\ x_n\}))))\ u \\
&\overset{(a)}{\equiv}\ (\lambda y\ \to\ ((\lambda\ y_1 \ldots y_n\ \to\ t[x_1/y_1, \ldots, x_n/y_n]) \\
&((\lambda\ z_1\ \to\ \texttt{case}\ z_1\ \texttt{of}\ \{p\ \to\ x_1\})\ x)\ \ldots \\
&((\lambda\ z_n\ \to\ \texttt{case}\ z_n\ \texttt{of}\ \{p\ \to\ x_n\})\ x)))\ u \\
&\overset{(lE)}{\equiv}\ (\lambda y\ \to\ ((\lambda\ y_1 \ldots y_n\ \to\ t[x_1/y_1, \ldots, x_n/y_n]) \\
&((\texttt{let}\ \{v_1 = \lambda\ z_1\ \to\ \texttt{case}\ z_1\ \texttt{of}\ \{p\ \to\ x_1\}\}\ \texttt{in}\ v_1)\ x)\ \ldots \\
&((\texttt{let}\ \{v_n = \lambda\ z_n\ \to\ \texttt{case}\ z_n\ \texttt{of}\ \{p\ \to\ x_n\}\}\ \texttt{in}\ v_n)\ x)))\ u \\
&\overset{(F)}{\equiv}\ (\lambda y\ \to\ ((\lambda\ y_1 \ldots y_n\ \to\ t[x_1/y_1, \ldots, x_n/y_n]) \\
&((\texttt{let}\ \{v_1\ p\ =\ x_1\}\ \texttt{in}\ v_1)\ x)\ \ldots\ ((\texttt{let}\ \{v_n\ p\ =\ x_n\}\ \texttt{in}\ v_n)\ x)))\ u \\
&\overset{(lL)}{\equiv}\ \texttt{let}\ \{v_1\ p\ =\ x_1; \ldots; v_n\ p\ =\ x_n\}\ \texttt{in} \\
&(\lambda y\ \to\ ((\lambda\ y_1 \ldots y_n\ \to\ t[x_1/y_1, \ldots, x_n/y_n])\ (v_1\ x)\ \ldots\ (v_n\ x)))\ u
\end{aligned}
$$

$$\overset{(\beta)}{\equiv} \texttt{let}\ \{v_1\ p\ =\ x_1; \ldots; v_n\ p\ =\ x_n\}\ \texttt{in}\ (\lambda y\ \rightarrow\ t[x_1/(v_1\ x), \ldots, x_n/(v_n\ x)])\ u$$

$$\overset{(j)}{\equiv} \texttt{let}\ \{v_1\ p\ =\ x_1; \ldots; v_n\ p\ =\ x_n\}\ \texttt{in}$$
$$\texttt{case}\ u\ \texttt{of}\ \{y\ \rightarrow\ t[x_1/(v_1\ x), \ldots, x_n/(v_n\ x)]\}$$

$$\overset{(i)}{\equiv} \texttt{let}\ \{v_1\ p\ =\ x_1; \ldots; v_n\ p\ =\ x_n\}\ \texttt{in}$$
$$\texttt{case}\ u\ \texttt{of}\ \{y\ \rightarrow\ t[x_1/(v_1\ x), \ldots, x_n/(v_n\ x)];\ \_\ \rightarrow\ e\}$$

wobei $z_1, \ldots, z_n, y_1, \ldots, y_n$ frische Variablen sind und

- $u, t, e \in \mathsf{H}(\mathsf{D}, \mathsf{V})$ *und*
- $p \in \mathsf{P}(\mathsf{D}, \mathsf{V})$

gilt.

Da jedes $\sim p$-Pattern letztlich auf die Form `case` u `of` $\{\sim p\ \rightarrow\ t;\ \_\ \rightarrow\ e\}$ *reduziert wird, ist der obere Beweis ausreichend.* □

3.4 `if`-Reduktion

Die `if`-Reduktion erzeugt für jedes Auftreten eines `if`-Terms eine neue Funktion im selben Block. Diese Funktion hat zwei Regeln, eine für den `then`-Zweig und eine für den `else`-Zweig. Die Patterns der ersten Regeln sind die freien Variablen, gefolgt von `True` und die der zweiten Regel sind wieder die freien Variablen, aber diesmal gefolgt von `False`. Der `if`-Term selbst wird durch einen Aufruf der neuen Funktion ersetzt, wobei die freien Variablen und die Bedingung des `if`-Terms übergeben werden. Formal ist die `if`-Reduktion wie folgt definiert:

$$\frac{\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2}{v\ x_1\ \ldots\ x_m\ t}$$

wobei

$$\mathsf{V}_{\mathsf{free}}(t_1) \cup \mathsf{V}_{\mathsf{free}}(t_2) \setminus \mathsf{V}_{\mathsf{F}} = \{x_1, \ldots, x_m\}$$

gilt und die folgende Funktion f in den selben Block aufgenommen werden muß:

$$f := \begin{pmatrix} v & x_1 & \ldots & x_m & \texttt{True} & = & t_1 \\ v & x_1 & \ldots & x_m & \texttt{False} & = & t_2 \end{pmatrix}$$

wobei $v \in \mathsf{V}_{\mathsf{F}}$ eine frische Variable ist.

Beispiel 3.2 *Die* `If`*-Reduktion ersetzt den folgende Teil eines Haskellprogramms*

$$\texttt{dec}\ x\ =\ \texttt{if}\ (x\ <=\ 0)\ \texttt{then}\ 0\ \texttt{else}\ (x-1)$$

durch diesen Teil:

$$\begin{aligned}&\texttt{dec}\ \ x\ =\ \texttt{dec}'\ x\ (x\ <=\ 0)\\&\texttt{dec}'\ \ x\ \ \texttt{True}\ \ =\ 0\\&\texttt{dec}'\ \ x\ \ \texttt{False}\ =\ x{-}1\end{aligned}$$

ersetzt.

Satz 3.4 *Die* `if`*-Reduktion ist semantikerhaltend.*

Beweis:

$$\begin{aligned}&\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2\\&\overset{(I)}{\equiv}\ (\texttt{case}\ t\ \texttt{of}\ \{\texttt{True}\ \rightarrow\ t_1;\texttt{False}\ \rightarrow\ t_2\})\ t\\&\overset{(a)}{\equiv}\ (\lambda\, x\ \rightarrow\ \texttt{case}\ x\ \texttt{of}\ \{\texttt{True}\ \rightarrow\ t_1;\texttt{False}\ \rightarrow\ t_2\})\ t\\&\overset{(lE)}{\equiv}\ (\texttt{let}\ \{v'\ =\lambda\, x\ \rightarrow\ \texttt{case}\ t\ \texttt{of}\ \{\texttt{True}\ \rightarrow\ t_1;\texttt{False}\ \rightarrow\ t_2\}\}\ \texttt{in}\ v')\ t\\&\overset{(F)}{\equiv}\ (\texttt{let}\ \{v'\ x\ =\ \texttt{case}\ x\ \texttt{of}\ \{\texttt{True}\ \rightarrow\ t_1;\texttt{False}\ \rightarrow\ t_2\}\}\ \texttt{in}\ v')\ t\\&\overset{(lL)}{\equiv}\ (v\ x_1\ \ldots\ x_m)\ t\end{aligned}$$

wobei v' eine frische Funktionsvariable ist. □

3.5 λ-Reduktion

Jede λ-Abstraktion in einem Haskellprogramm HP entspricht einer annonymen Funktion. Daher kann zu jeder λ-Abstraktion eine äquivalente benannte Funktion erzeugt werden. Die λ-Reduktion nutzt diesen Umstand und erzeugt zu jeder λ-Abstraktion die äquivalente benannte Funktion und legt diese im selben Block ab, während der λ-Term selbst durch einen Aufruf der neuen benannten Funktion ersetzt wird. Konkret ist die λ-Reduktion definiert wie folgt:

$$\frac{\lambda\, p_1\ \ldots\ p_n\ \rightarrow\ t}{v\ x_1\ \ldots\ x_m}$$

wobei

$$\mathsf{V}_{\mathsf{free}}(\lambda\, p_1\ \ldots\ p_n\ \rightarrow\ t) \setminus \mathsf{V}_{\mathsf{F}} = \{x_1, \ldots, x_m\}$$

gilt und die folgende Funktion f in denselben Block aufgenommen werden muß:

$$f = (\ v\ \ x_1\ \ \ldots\ \ x_m\ \ p_1\ \ \ldots\ \ p_n\ \ =\ \ t\ )$$

wobei $v \in \mathsf{V}_{\mathsf{F}}$ eine frische Variable ist.

Beispiel 3.3 *Die λ-Reduktion ersetzt den folgende Teil eines Haskellprogramms*

$$\texttt{decBy}\ \ x\ \ =\ \ \lambda\, y\ \rightarrow\ \texttt{if}\ (y\ >\ x)\ \texttt{then}\ 0\ \texttt{else}\ (y - x)$$

durch diesen Teil:

$$\texttt{decBy}\ \ x = \texttt{decBy}'\ x$$
$$\texttt{decBy}'\ \ x\ \ y\ \ =\ \ \texttt{if}\ (y\ >\ x)\ \texttt{then}\ 0\ \texttt{else}\ (y - x)$$

ersetzt.

Satz 3.5 *Die λ-Reduktion erhält die Semantik.*

Beweis:

$$\begin{aligned}
&\lambda\, p_1\ \ldots\ p_n\ \rightarrow\ t\\
&\overset{(\lambda)}{\equiv} \lambda\, y_1 \ldots\ y_n\ \rightarrow\ \texttt{case}\ (y_1, \ldots, y_n)\ \texttt{of}\ \{p_1\ \ldots\ p_n\ \rightarrow\ t\}\\
&\overset{(lE)}{\equiv} \texttt{let}\ \{v'\ =\ \lambda\, y_1 \ldots\ y_n\ \rightarrow\ \texttt{case}\ (y_1, \ldots, y_n)\ \texttt{of}\ \{p_1\ \ldots\ p_n\ \rightarrow\ t\}\}\ \texttt{in}\ v'\\
&\overset{(F)}{\equiv} \texttt{let}\ \{v'\ p_1\ \ldots\ p_n\ =\ t\}\ \texttt{in}\ v'\\
&\overset{(lL)}{\equiv} (v\ x_1\ \ldots\ x_m)
\end{aligned}$$

wobei $y_1, \ldots, y_n \in \mathsf{V}_{\mathsf{L}}$ *und* $v' \in \mathsf{V}_{\mathsf{F}}$ *frische Variablen sind.* □

3.6 case-Reduktion

Das case-Konstrukt kann ähnlich reduziert werden wie ein λ-Term. Es wird eine neue Funktion eingeführt, die die gleichen Patterns wie das case-Konstrukt hat. Das case-Konstrukt selbst wird durch einen passenden Aufruf der neuen Funktion ersetzt. Die case-Reduktion ist wie folgt definiert:

$$\frac{\texttt{case}\ t\ \texttt{of}\ \{r_1;\ldots;r_n\}}{v\ x_1\ \ldots\ x_m\ t}$$

wobei

$$\mathsf{V}_{\mathsf{free}}(\texttt{case}\ t\ \texttt{of}\ \{r_1;\ldots;r_n\}) \setminus \mathsf{V}_{\mathsf{F}} = \{x_1,\ldots,x_m\}$$

gilt und die folgende Funktion f in denselben Block aufgenommen werden muß:

$$f = \begin{pmatrix} v\ x_1\ \ldots\ x_m\ r_1 \\ \vdots \\ v\ x_1\ \ldots\ x_m\ r_n \end{pmatrix}$$

wobei $v \in \mathsf{V}_{\mathsf{F}}$ eine frische Variable ist.

Beispiel 3.4 *Der folgende Teil eines Haskellprogramms*

```
isTenOrEleven x = case x of
                    10 → True
                    11 → True
                    _  → False
```

ersetzt die case*-Reduktion durch den folgenden Teil:*

```
isTenOrEleven  x  = isTenOrEleven' x
isTenOrEleven' 10 = True
isTenOrEleven' 11 = True
isTenOrEleven' _  = False
```

Satz 3.6 *Die* case*-Reduktion ist semantikerhaltend.*

Beweis:

$$\begin{aligned}
&\texttt{case}\ t\ \texttt{of}\ \{r_1;\ldots;r_n\} \\
&\overset{(a)}{\equiv} (\lambda\, y \rightarrow\ \texttt{case}\ y\ \texttt{of}\ \{r_1;\ldots;r_n\})\ t \\
&\overset{(lE)}{\equiv} (\texttt{let}\ \{v' = \lambda\, y \rightarrow\ \texttt{case}\ y\ \texttt{of}\ \{r_1;\ldots;r_n\}\}\ \texttt{in}\ v')\ t \\
&\overset{(F)}{\equiv} (\texttt{let}\ \{f'\}\ \texttt{in}\ v')\ t \\
&\overset{(lL)}{\equiv} (v\ x_1\ \ldots\ x_m)t
\end{aligned}$$

wobei

$$f' = \begin{pmatrix} v'\ r_1 \\ \vdots \\ v'\ r_n \end{pmatrix}$$

und $v' \in \mathsf{V_F}$ *eine frische Variable ist.* □

3.7 Bedingungsreduktion

Die bedingten Regeln innerhalb einer Regel werden in ihrer syntaktischen Reihenfolge überprüft und bei der ersten Regel, deren Bedingung erfüllt ist, wird der zugehörige Ergebnisterm zurückgeliefert. Wenn keine Bedingung einer bedingten Regel innerhalb einer Regel erfüllt ist, geht der Auswerter zu der nächsten Regel über und überprüft deren bedingte Regeln. Die Bedingungsreduktion ersetzt die Regeln, die bedingte Regeln enthalten, durch Regeln ohne solche. Die Ergebnisterme werden durch Aufrufe neuer Funktionen, welche die Bedingungen überprüfen und den richtigen Ergebnisterm zurückliefern, ersetzt. Außerdem werden Literalpatterns (Patterns aus $\mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A}$) und $(x + n)$-Patterns eliminiert, da diese implizite Bedingungen enthalten, die ebenfalls in den neuen Funktionen überpüft werden müssen.

Die Bedingungen der bedingten Regeln können durch eine Verschachtelung von `if`-Konstrukten simuliert werden. Das heißt, eine Funktion, welche nur Basispatterns benutzt und von der Form

$$\begin{pmatrix} v\ p_{1,1}\ \ldots\ p_{n,1} & |c_{1,1} = t_{1,1} \\ & \vdots \\ & |c_{1,m_1} = t_{1,m_1}\ where\ \underline{fs}_1 \\ & \vdots \\ v\ p_{1,k}\ \ldots\ p_{n,k} & |c_{k,1} = t_{k,1} \\ & \vdots \\ & |c_{k,m_k} = t_{k,m_k}\ where\ \underline{fs}_k \end{pmatrix}$$

ist, kann durch die Funktionen $f_1, \ldots, f_k, f_{k+1}$ simuliert werden, wenn im Haskellprogramm die Funktionsvariable v durch die neue Funktionsvariable v_1 ersetzt wird. Die Funktionen $f_1, \ldots, f_k$ sind dabei von der folgenden Form

$$f_i = \begin{pmatrix} v_i & p_{1,i} & \ldots & p_{n,i} & = u_i\sigma_i\ where\ \underline{fs}_i \\ v_i & y_1 & \ldots & y_n & = v_{i+1,0}\ y_1\ \ldots\ y_n \end{pmatrix}$$

mit

$$u_i = \texttt{if}\ c_{i,1}\ \texttt{then}\ t_{i,1}\ \texttt{else}\ (\ldots \texttt{if}\ c_{i,m_i}\ \texttt{then}\ t_{i,m_i}\ \texttt{else}\ (v_{i+1}\ p_{1,i}\ \ldots\ p_{n,i})\ldots)$$

für alle $1 \leq i \leq k$ und die Abschlußfunktion f_{k+1} hat diese Form:

$$f_{k+1} = \left(\begin{array}{cccc} v_{k+1} & y_1 & \ldots & y_n \end{array} = \texttt{Program error} \right)$$

Nach einem erfolgreichen Patternmatching in der Funktion f_i wird der Term u_i ausgeführt, welcher durch die einzelnen `if`-Konstrukte prüft, welche Bedingungen einer ursprünglichen bedingten Regel gilt. Wenn eine Bedingung erfüllt ist, wird der ursprüngliche Ergebnisterm der bedingten Regel ausgewertet. Für den Fall, daß keine Bedingung erfüllt ist, wird die nächste Funktion f_{i+1} mit denselben Eingaben durch den letzten `else`-Zweig aufgerufen. Sollten am Schluß keine Regeln der ursprünglichen Funktion gepaßt haben, wird die Funktion f_k aufgerufen, welche die Auswertung mit einem `Program error` abbricht, da der Haskellauswerter in der ursprünglichen Funktion genauso mit einem `Program error` abbricht.

Die Patterns der ursprünglichen Funktion sind im allgemeinen nicht nur Basispatterns, sondern enthalten möglicherweise noch Literalpatterns oder $(x+n)$-Patterns (also Spezial-Patterns), so daß deren impliziten Bedingungen während der Auswertung ebenfalls überprüft werden müssen. Dazu werden statt der Funktion f_i die Funktionen $f_{i,0}, \ldots, f_{i,r_i}$ erstellt, die diese Überprüfung zusätzlich übernehmen. Jede Funktion $f_{i,j}$ überprüft dabei genau eine implizite Bedingung eines Spezial-Patterns. Die Funktionen sind folgendermaßen aufgebaut:

- $f_{i,0} = \left(\begin{array}{ccccc} v_{i,0} & p'_{1,i,0} & \ldots & p'_{n,i,0} & = v_{i,1}\ b_{i,0}\ p'_{1,i,0}\ \ldots\ p'_{n,i,0} \\ v_{i,0} & y_1 & \ldots & y_n & = v_{i+1,0}\ y_1\ \ldots\ y_n \end{array} \right)$
  für alle $1 \leq i \leq k$
- $f_{i,j} = \left(\begin{array}{ccccc} v_{i,j} & \texttt{True} & p'_{1,i,j} & \ldots & p'_{n,i,j} & = v_{i,j+1}\ b_{i,j}\ p'_{1,i,j}\ \ldots\ p'_{n,i,j} \\ v_{i,j} & y_0 & y_1 & \ldots & y_n & = v_{i+1,0}\ y_1\ \ldots\ y_n \end{array} \right)$
  für alle $1 \leq i \leq k$ und $1 < j < r_i$
- $f_{i,r_i} = \left(\begin{array}{ccccc} v_{i,r_i} & \texttt{True} & p'_{1,i,r_i} & \ldots & p'_{n,i,r_i} & = t_i\sigma_i\ where\ \underline{fs}_i\sigma_i \\ v_{i,r_i} & y_0 & y_1 & \ldots & y_n & = v_{i+1,0}\ y_1\ \ldots\ y_n \end{array} \right)$
  für alle $1 \leq i \leq k$
- $f_{k+1,0} = \left(\begin{array}{cccc} v_{k+1,0} & y_1 & \ldots & y_n \end{array} = \texttt{Program error} \right)$

Dabei werden die Terme $b_{i,0}, \ldots, b_{i,r_{i-1}}$ die Bedingungen aufnehmen und die Patterns $p'_{1,i,j}, \ldots, p'_{n,i,j}$ so erstellt, daß diese zum richtigen Zeitpunkt den Auswerter zur Bedingungsprüfung unterbrechen. Die Funktion f_{i,r_i} überprüft keine weiteren Bedingungen von Spezial-Patterns und wählt, wie die Funktion f_i im einfachen Fall ohne Spezial-Patterns, den richtigen Ergebnisterm der ursprünglichen bedingten Regel mit Hilfe des durch die Substitution σ_i angepaßten Terms t_i aus. Dabei ist der Term

$$t_i = \texttt{if}\ c_{i,1}\ \texttt{then}\ t_{i,1}\ \texttt{else}\ (\ldots \texttt{if}\ c_{i,m_i}\ \texttt{then}\ t_{i,m_i}\ \texttt{else}\ (v_{i+1,0}\ p'_{1,i,r_i}\ \ldots\ p'_{n,i,r_i})\ldots)$$

ähnlich zu dem Term u_i aus der Funktion f_i aufgebaut. Die Substitution σ_i dient, dazu die Variablen der $(x+n)$-Patterns mit den richtigen Werten zu binden, denn die Variable x muß den Wert $y-n$ bekommen, wenn y gegen das Pattern $(x+n)$ gematcht wird.

Im folgenden wird der komplexe Aufbau der Patterns $p'_{1,i,j}, \ldots, p'_{n,i,j}$ der Funktion $f_{i,j}$ motiviert. Danach werden die Terme $b_{i,0}, \ldots, b_{i,r_{i-1}}$ der Bedingungen und die Substitution σ_i erklärt. Zum schluß dieses Abschnitts wird dann die Bedingungsreduktion zusammengesetzt.

Die Patterns $p'_{1,i,j}, \ldots, p'_{n,i,j}$ sind direkt abhängig von den Patterns $p_{1,i}, \ldots, p_{n,i}$ der Regel i aus der ursprünglichen Funktion.
Der Einfachheit halber fassen wir die Patterns $p_{1,i}, \ldots, p_{n,i}$ der Regel i in dem Tupel $\underline{pats}_i$ wie folgt zusammen:

$$\underline{pats}_i := (p_{1,i}, \ldots, p_{n,i})$$

Wenn das Pattern $\underline{pats}_i$ gegen einen Term gematcht wird, werden die Subpatterns aus $\underline{pats}_i$ durch den Auswerter in der lexikographischen Reihenfolge ihrer Stellen gematcht.(Die Zusammenfassung in den Tupel $\underline{pats}_i$ ist zulässig, weil sich genau dieselbe Reihenfolge ergibt, wenn die Regel i der ursprünglichen Funktion gegen n Eingabeterme gematcht wird.)

Um die Spezial-Patterns aus $\underline{pats}_i$ zu entfernen, muß deren Verhalten anders emuliert werden. Das erfordert das Unterbrechen des Auswerters an den jeweiligen richtigen Stellen, um dort deren nötige Bedingungen zu überprüfen. Sei nun $\pi_{i,0}$ die erste Stelle (nach der lexikographischen Reihenfolge $<_{\text{lex}}$) mit $\underline{pats}_i|_{\pi_{i,0}} \in \mathsf{P_S}(\mathsf{D}, \mathsf{V})$. So wird, um den Auswerter an dieser Stelle zu unterbrechen, an jeder nachfolgenden Stelle von $\pi_{i,0}$ eine Variable in $\underline{pats}_i$ eingesetzt und man erhält das Pattern $\underline{pats'}_i$, welches formal wie folgt definiert ist:

$$\underline{pats'}_i := \text{varrep}(\pi_{i,0}, \underline{pats}_i)$$

mit

$$\begin{aligned}\text{varrep}(\rho, t) := t[z_n]_{\rho_n} \ldots [z_1]_{\rho_1}, \text{ wobei } &\{\rho_1, \ldots, \rho_n\} = \{\rho' \in Occ(t) \mid \rho <_{\text{lex}} \rho'\},\\ &\rho_1 <_{\text{lex}} \ldots <_{\text{lex}} \rho_n \text{ und}\\ &z_1, \ldots, z_n \in \mathsf{V_L} \text{ frische Variablen sind.}\end{aligned}$$

Die Funktion varrep setzt an alle nachfolgenden Stellen von ρ eine frische Variable. Dieses Vorgehen stoppt tatsächlich das Patternmatching an der Stelle $\pi_{i,0}$, weil das Matching einer Variablen gegen einen Term keine weiteren Auswertungen anstößt und somit neutral ist. Die Reihenfolge der Ersetzungen im Ausdruck $t[z_n]_{\rho_n} \ldots [z_1]_{\rho_1}$ ist umgekehrt, da möglicherweise die Position $\rho_i <_{\text{lex}} \rho_j$ über

der lexikographisch nachfolgenden Position ρ_j steht, es gilt also möglicherweise $\rho_i \leq_{\text{pre}} \rho_j$, so daß nach einer Ersetzung mit einer frischen Variablen an Position ρ_i die Stelle ρ_j nicht mehr existiert. Eine Regel, die einen Term gegen $\underline{pats'}_i[x]_{\pi_{i,0}}$ matcht, kann nun die Bedingung für das ersten Spezial-Pattern in $\underline{pats}$ prüfen, da die Variable x an den richtigen Wert gebunden ist. Beispielsweise sei

$$\underline{pats}_i = ((\texttt{Just}\ u) : xs, \texttt{Just}\ 1, [\,], 0)$$

mit $u, xs \in \mathsf{V_L}$. So ergibt sich für $\underline{pats'}_i$ folgender Wert:

$$\underline{pats'}_i = ((\texttt{Just}\ u) : xs, \texttt{Just}\ 1, y, z)$$

Alle Positionen nach dem ersten Spezial-Pattern 1 an der Stelle $\pi_{i,0} = 2\ 1$ sind durch die frischen Variablen $y, z \in \mathsf{V_L}$ ersetzt worden. Das Pattern 1 wird nun noch durch die neue Variable $x \in \mathsf{V_L}$ ersetzt, um den Wert, gegen den 1 gematcht werden wird, für die Bedingungsprüfung verfügbar zu machen. Zusammen ergibt sich so das Pattern:

$$\underline{pats'}_i[x]_{\pi_{i,0}} = ((\texttt{Just}\ u) : xs, \texttt{Just}\ x, y, z)$$

mit $u, xs, x, y, z \in \mathsf{V_L}$

Nach dem Matching mit Pattern $\underline{pats'}_i[x]_{\pi_{i,0}}$ kann die Bedingung $b_{i,0}$ geprüft werden. Für das Beispiel ist es die Bedingung $x == 1$.

Um nach der Bedingungsprüfung für das Spezial-Pattern an Stelle $\pi_{i,0}$ das Patternmatching an der unmittelbar nächsten Stelle wieder aufzunehmen, wird das Pattern $\underline{pats''}_i[x']_{\pi_{i,1}}$ gegen den Term $\underline{pats'}_i[x]_{\pi_{i,0}}$ gematcht, wobei $x' \in \mathsf{V_L}$ eine frische Variable ist, $\pi_{i,1}$ die Stelle des nächsten Spezial-Pattern nach $\pi_{i,0}$ ist und

$$\underline{pats''}_i = \text{varrep}(\pi_{i,1}, \underline{pats}_i[x]_{\pi_{i,0}})$$

gilt. Außerdem kann das Pattern $\underline{pats'}_i[x]_{\pi_{i,0}}$ als Term aufgefaßt werden, da es keine Spezial-Patterns mehr enthält.

Da in dem Beispiel das Pattern 0 das nächste Spezial-Pattern an Stelle $\pi_{i,1}$ nach $\pi_{i,0}$ ist, gilt

$$\underline{pats''}_i = ((\texttt{Just}\ u) : xs, \texttt{Just}\ x, [\,], 0)$$

und

$$\underline{pats''}_i[x']_{\pi_{i,1}} = ((\texttt{Just}\ u) : xs, \texttt{Just}\ x, [\,], x')$$

Das erneute Matchen an Stellen kleiner oder gleich $\pi_{i,0}$ ist erfolgreich und dort werden auch keine weiteren Auswertungen angestoßen, da der Term $\underline{pats''}_i[x']_{\pi_{i,1}}$

und das Pattern $\underline{pats}'_i[x]_{\pi_{i,0}}$ an diesen Stellen gleich sind. So werden nur die Terme zwischen $\pi_{i,0}$ und $\pi_{i,1}$ gegen die Pattern zwischen $\pi_{i,0}$ und $\pi_{i,1}$ gematcht und eventuell Auswertungen dabei erzwungen. Die Patterns ab Stelle $\pi_{i,1}$ sind nur Variablen und können so keine Auswertung anstoßen, so daß bei erfolgreichem Matching die Bedingung $b_{i,1}$ für das Spezial-Pattern an Stelle $\pi_{i,1}$ überprüft werden kann. Danach wird das Patternmatching nach Stelle $\pi_{i,1}$ wieder aufgenommen bis dieses wieder an Stelle $\pi_{i,2}$ unterbrochen wird und so weiter.

Nach dem oben vorgestellten Prinzip werden die Patterns $p'_{1,i,j}, \ldots, p'_{n,i,j}$ der Funktion $f_{i,j}$ wie folgt festgelegt:

$$(p'_{1,i,j}, \ldots, p'_{n,i,j}) := \text{varrep}(\pi_{i,j}, \underline{pats}_i[y_{i,0}]_{\pi_{i,0}} \ldots [y_{i,j}]_{\pi_{i,j}})$$

wobei

- $\{\pi_{i,0}, \ldots, \pi_{i,r_i-1}\} := \{\pi \in Occ(\underline{pats}_i) \mid (\underline{pats}_i|_\pi) \in \mathsf{P_S}(\mathsf{D}, \mathsf{V})\}$,
- $\underline{pats}_i := (p_{1,i}, \ldots, p_{n,i})$ und
- $\pi_{i,0} <_{\text{lex}} \ldots <_{\text{lex}} \pi_{i,r_i-1}$ gilt und
- $y_{i,0}, \ldots, y_{i,r_i-1}$ frische Variablen sind.

Die Stellen $\pi_{i,0}, \ldots, \pi_{i,r_i-1}$ aller Spezial-Patterns sind also lexikographisch aufsteigend geordnet, und in $\underline{pats}_i$ gibt es keine weiteren Spezial-Patterns.

Das Patternmatching der Spezial-Patterns $\mathsf{P_S}(\mathsf{D}, \mathsf{V})$ besteht aus der Prüfung bestimmter boolescher Bedingungen, welche aus Aufrufen der Funktionen == oder >= zusammengesetzt sind. Die Bedingungen $b_{i,0}, \ldots, b_{i,r_i-1}$ in den Funktionen $f_{i,0}, \ldots, f_{i,r_i-1}$ können nun das Matching der Spezial-Patterns emulieren, da diese die zu matchenden Terme gebunden an die Variablen $y_{i,0}, \ldots, y_{i,r_i-1}$ vorfinden und der Auswerter im richtigen Moment unterbrochen wurde. Die Bedigungen sind wie folgt festgelegt:

$$b_{i,j} := \begin{cases} y_{i,j} >= n & \text{falls } \underline{pats}_i|_{\pi_{i,j}} = (x+n), \\ y_{i,j} == \underline{pats}_i|_{\pi_{i,j}} & \text{falls } \underline{pats}_i|_{\pi_{i,j}} \in \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A} \end{cases}$$

Wenn Bedingung $b_{i,j}$ zu `True` ausgewertet wird, nimmt die Funktion $f_{i,j+1}$ das Patternmatching an der direkten Stelle nach $\pi_{i,j}$ wieder auf.

Zusätzlich muß bei den $(x+n)$-Patterns die Variable x an den richtigen Wert gebunden werden. Diese Aufgabe übernimmt die Substitution $\sigma_i := \sigma_{i,0} \ldots \sigma_{i,r_i-1}$, wobei

$$\sigma_{i,j} := \begin{cases} [x/(y_{i,j}-n)] & \text{falls } \underline{pats}_i|_{\pi_{i,j}} = (x+n), \\ id & \text{falls } \underline{pats}_i|_{\pi_{i,j}} \in \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A} \end{cases}$$

gilt.

Die Funktion f_{i,r_i} überprüft keine weitere Bedingung und ihre Patterns sind folgendermaßen festgelegt:

$$(p'_{1,i,r_i}, \ldots, p'_{n,i,r_i}) := \underline{pats}_i[y_{i,0}]_{\pi_{i,0}} \ldots [y_{i,r_{i-1}}]_{\pi_{i,r_{i-1}}}$$

Es werden also die Patterns nach dem letzten Spezial-Pattern an Stelle $\pi_{i,r_{i-1}}$ geprüft bevor der Ergebnisterm $t_i\sigma_i$ ausgwertet wird.

Da nun alle Teile erklärt sind, kann die Bedingungsreduktion wie folgt zusammengesetzt werden:

$$\frac{\left(\begin{array}{ll} v\ p_{1,1}\ \ldots\ p_{n,1} & |c_{1,1} = t_{1,1} \\ & \vdots \\ & |c_{1,m_1} = t_{1,m_1}\ where\ \underline{fs}_1 \\ & \vdots \\ v\ p_{1,k}\ \ldots\ p_{n,k} & |c_{k,1} = t_{k,1} \\ & \vdots \\ & |c_{k,m_k} = t_{k,m_k}\ where\ \underline{fs}_k \end{array}\right)}{\left(\begin{array}{l} v\ p'_{1,1,0} \ldots p'_{n,1,0} = v_{1,0}\ p'_{1,1,0}\ \ldots\ p'_{n,1,0} \\ \vdots \\ v\ p'_{1,k,0} \ldots p'_{n,k,0} = v_{k,0}\ p'_{1,k,0}\ \ldots\ p'_{n,k,0} \end{array}\right)}$$

Dabei müssen die folgenden Funktionen $f_{1,1}, \ldots, f_{k,r_k}, f_{k+1,0}$ in denselben Block aufgenommen werden müssen:

- $f_{i,0} = \left(\begin{array}{llll} v_{i,0} & p'_{1,i,0} & \ldots & p'_{n,i,0} \\ v_{i,0} & y_1 & \ldots & y_n \end{array}\right.$ $\left.\begin{array}{l} = v_{i,1}\ b_{i,0}\ p'_{1,i,0}\ \ldots\ p'_{n,i,0} \\ = v_{i+1,0}\ y_1\ \ldots\ y_n \end{array}\right)$
  für alle $1 \leq i \leq k$
- $f_{i,j} = \left(\begin{array}{llllll} v_{i,j} & \texttt{True} & p'_{1,i,j} & \ldots & p'_{n,i,j} \\ v_{i,j} & y_0 & y_1 & \ldots & y_n \end{array}\right.$ $\left.\begin{array}{l} = v_{i,j+1}\ b_{i,j}\ p'_{1,i,j}\ \ldots\ p'_{n,i,j} \\ = v_{i+1,0}\ y_1\ \ldots\ y_n \end{array}\right)$
  für alle $1 \leq i \leq k$ und $1 < j < r_i$
- $f_{i,r_i} = \left(\begin{array}{llllll} v_{i,r_i} & \texttt{True} & p'_{1,i,r_i} & \ldots & p'_{n,i,r_i} \\ v_{i,r_i} & y_0 & y_1 & \ldots & y_n \end{array}\right.$ $\left.\begin{array}{l} = t_i\sigma_i\ where\ \underline{fs}_i\sigma_i \\ = v_{i+1,0}\ y_1\ \ldots\ y_n \end{array}\right)$
  für alle $1 \leq i \leq k$
- $f_{k+1,0} = \left(\begin{array}{llll} v_{k+1,0} & y_1 & \ldots & y_n \end{array}\right.$ $\left.\begin{array}{l} = \texttt{Program error} \end{array}\right)$

wobei

- $b_{i,j} := \begin{cases} y_{i,j} >= n & \text{falls } \underline{pats}_i|_{\pi_{i,j}} = (x+n), \\ y_{i,j} == \underline{pats}_i|_{\pi_{i,j}} & \text{falls } \underline{pats}_i|_{\pi_{i,j}} \in \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A} \end{cases}$

- $\sigma_i := \sigma_{i,0} \ldots \sigma_{i,r_i-1}$ mit $\sigma_{i,j} := \begin{cases} [x/(y_{i,j}-n)] & \text{falls } \underline{pats}_i|_{\pi_{i,j}} = (x+n), \\ id & \text{falls } \underline{pats}_i|_{\pi_{i,j}} \in \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A} \end{cases}$

- $(p'_{1,i,j}, \ldots, p'_{n,i,j}) := \text{varrep}(\pi_{i,j}, \underline{pats}_i[y_{i,0}]_{\pi_{i,0}} \ldots [y_{i,j}]_{\pi_{i,j}})$
  für alle $1 \leq i \leq r_i$ und
  $(p'_{1,i,r_i}, \ldots, p'_{n,i,r_i}) := \underline{pats}_i[y_{i,0}]_{\pi_{i,0}} \ldots [y_{i,r_{i-1}}]_{\pi_{i,r_{i-1}}}$
  mit

  - $\{\pi_{i,0}, \ldots, \pi_{i,r_i-1}\} := \{\pi \in Occ(\underline{pats}_i) \mid (\underline{pats}_i|_\pi) \in \mathsf{P_S}(\mathsf{D}, \mathsf{V})\}$,
  - $\underline{pats}_i := (p_{1,i}, \ldots, p_{n,i})$ und
  - $\pi_{i,0} <_{\text{lex}} \ldots <_{\text{lex}} \pi_{i,r_i-1}$ gilt und
  - $y_{i,0}, \ldots, y_{i,r_i-1}$ frische Variablen sind.

und

$t_i = \texttt{if}\ c_{i,1}\ \texttt{then}\ t_{i,1}\ \texttt{else}\ (\ldots \texttt{if}\ c_{i,m_i}\ \texttt{then}\ t_{i,m_i}\ \texttt{else}\ (v_{i+1,0}\ p'_{1,i,r_i}\ \ldots\ p'_{n,i,r_i}) \ldots)$

gilt.

Beispiel 3.5 *Durch die Bedingungsreduktion wird die folgende Funktion*[1]

```
detect :: List a → Char → Int → Bool → Int
detect Nil       'b'  (x+1)  True  =  x
detect (Cons a as)  x    y      z   |  length as > 3 = 0
```

durch diese neue Funktion ersetzt:

```
detect Nil x1 z1 z2 = detectA0  Nil x1 z1 z2
detect as  x  y  z  = detectB0  as  x  y  z

detectA0 Nil x1 z1 z2 = detectA1 (x1 == 'b') Nil x1 z1 z2
detectA0 y1  y2 y3 y4 = detectB0             y1  y2 y3 y4

detectA1 True Nil x1 x2 z1 = detectA2 (x2 >= 1) Nil x1 x2 z1
detectA1 y0   y1  y2 y3 y4 = detectB0           y1  y2 y3 y4

detectA2 True Nil x1 x2 True = x2 - 1
detectA2 y0   y1  y2 y3 y4   = detectB0  y1 y2 y3 y4

detectB0 (Cons a as) x y z = if (length as > 3) then 0
                             else detectC0 (Cons a as) x y z
detectB0 y1          y2 y3 y4 = detectC0 y1 y2 y3 y4

detectC0 y1 y2 y3 y4 = error
```

[1] Üblicherweise sind Funktionen die soviele sonderfälle aufeimal beinhalten sehr groß, so daß hier eine weniger sinnvolle Funktion gewählt wurde, welche aber die wesentlichen Aspekte der Bedingungsreduktion beleuchtet.

Satz 3.7 *Die Bedingungsreduktion erhält die Semantik.*

Beweisidee:

Da sich aus der Semantikregel (g) *aus [J+98, Seiten 31-33] die Auswertungsreihenfolge eines Konstruktorpatterns ablesen läßt und es genau der lexikographischen Reihenfolge der Position in einem Pattern entspricht, sind die Unterbrechungen einer Auswertung an den Stellen* $\pi_{i,0}, \ldots, \pi_{i,r_i-1}$ *exakt die, an der die Auswertung der Spezial-Patterns folgen würde. Die Funktionen* $f_{i,0}, \ldots, f_{i,r_i}$ *erzwingen so per Definition, daß die Bedigungen* $b_{i,1}, \ldots, b_{i,r_i-1}$ *zum richtigen Zeitpunkt ausgewertet werden. Die Bedingung einzelner Patterns aus* $\mathbb{Z} \cup \mathbb{Q} \cup \mathbb{A}$ *entspricht der aus Semantikregel* (h) *und die Bedingung eines* $(x+n)$*-Patterns entspricht der aus Semantikregel* (r)*. Außerdem werden die Substitutionen* $\sigma_{i,1}, \ldots, \sigma_{i,r_i-1}$*, ebenfalls wie in Semantikregel* (r)*, festgelegt.*

3.8 `newtype`-Reduktion

Die `newtype`-Deklaration dient dazu, einen namentlich neuen Typ auf der Basis eines schon vorhandenen Typs, einzuführen. Dieser neue Typ ist strukturell identisch mit dem Basistyp, aber wird im Typsystem als völlig neuer und fremder Typ verstanden. Syntaktisch wird zwischen diesen beiden Typen mit einem Kapselkonstruktor, der den neuen Typ kapselt, unterschieden. Sei beispielsweise folgende `newtype`-Deklaration in einem Haskellprogramm angegeben:

```
newtype NewBool = NB Bool
```

Diese `newtype`-Deklaration führt den neuen Typ `NewBool` auf der Basis von Typ `Bool` ein. Die Kapselung übernimmt der Kapselkonstruktor `NB`. Syntaktisch unterscheiden sich Kapselkonstruktoren und normale Konstruktoren nicht, aber das Verhalten während des Matchings ist unterschiedlich. Für einen Kapselkonstruktor K hat der Wert $K\ \bot$ die gleiche Semantik wie $\bot$. Insbesondere wenn das Pattern $K\ x$ gegen $\bot$ gematcht wird, wird x an den Wert $\bot$ gebunden und scheitert nicht, wie es für einen normalen Konstruktor der Fall wäre.

Die `newtype`-Reduktion ersetzt die Kapselkonstruktoren durch frische normale Konstruktoren und paßt die Patterns so an, daß diese semantisch den Patterns aus dem Kapselkonstruktor entsprechen. Das Pattern $(K_1 \ldots (K_n\ x) \ldots)$ mit Kapselkonstruktoren $K_1, \ldots, K_n$ und Variable x ist unabweisbar. Dem entsprechend wird es zu $\sim(K_1 \ldots (K_n\ x) \ldots)$ umgeschrieben und die $\sim p$-Reduktion darauf angewendet. Für Patterns der Form $(K_1 \ldots (K_n(c\ p_1\ \ldots\ p_m)) \ldots)$ ist keine Anpassung nötig, wenn c ein normaler Konstruktor ist, da der gekapselte Wert sowieso ausgewertet werden muß, wenn das Pattern $(c\ p_1\ \ldots\ p_m)$ gegen diesen

gematcht wird. Falls die Patterns $p_1, \ldots, p_m$ wieder Kapselkonstruktoren enthalten müssen diese ebenfalls eliminiert werden. Formal ist die `newtype`-Reduktion wie folgt definiert:

$$\frac{(K_1 \ldots (K_n\ x) \ldots)}{\sim(K'_1 \ldots (K'_n\ x) \ldots)}$$

und

$$\frac{(K_1 \ldots (K_n(c\ p_1\ \ldots\ p_m)) \ldots)}{(K'_1 \ldots (K'_n(c\ p_1\ \ldots\ p_m)) \ldots)}$$

Dabei sind $K'_1, \ldots, K'_n$ die neuen normalen Konstruktoren zu den Kapselkonstruktoren $K_1, \ldots, K_n$, während $c \in \underline{\mathsf{Cons}}_{\mathsf{D}}$ ein normaler Konstruktor ist und $p_1, \ldots, p_n$ Patterns aus $\mathsf{P}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ sind.

Satz 3.8 *Die* `newtype`*-Reduktion ist semantikerhaltend.*

Beweis:

Seien im folgenden $u, t, e \in \mathsf{H}(\mathsf{D}, \mathsf{V})$, $x', x'' \in \mathsf{V}_{\mathsf{L}}$ *und* $0 \leq i \leq n$.

- *Fall 1:*

  $\texttt{case}\ (K_1 \ldots (K_n\ u) \ldots)\ \texttt{of}\ \{(K_1 \ldots (K_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}$

  $\overset{(k)}{\equiv}\ \texttt{case}\ (K_2 \ldots (K_n\ u) \ldots)\ \texttt{of}\ \{(K_2 \ldots (K_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}$

  $\overset{(k)}{\equiv} \ldots \overset{(k)}{\equiv}\ \texttt{case}\ K_n\ u\ \texttt{of}\ \{K_n\ (c\ p_1 \ldots\ p_m)\ \to\ t;\ \_\ \to\ e\}$

  $\overset{(k)}{\equiv}\ \texttt{case}\ u\ \texttt{of}\ \{c\ p_1 \ldots\ p_m\ \to\ t;\ \_\ \to\ e\}$

  $\overset{(ß)}{\equiv}\ (\lambda\ x'\ \to\ \texttt{case}\ x'\ \texttt{of}\ \{c\ p_1 \ldots\ p_m\ \to\ t;\ \_\ \to\ e\})\ u$

  $\overset{(j)}{\equiv}\ \texttt{case}\ u\ \texttt{of}\ \{x'\ \to\ \texttt{case}\ x'\ \texttt{of}\ \{c\ p_1 \ldots\ p_m\ \to\ t;\ \_\ \to\ e\}\}$

  $\overset{(q)}{\equiv}\ \texttt{case}\ K'_n\ u\ \texttt{of}\ \{K'_n\ x\ \to\ \texttt{case}\ x'\ \texttt{of}\ \{c\ p_1 \ldots\ p_m\ \to\ t;\ \_\ \to\ e\}\}$

  $\overset{(g)}{\equiv}\ \texttt{case}\ K'_n\ u\ \texttt{of}\ \{K'_n\ (c\ p_1 \ldots\ p_m)\ \to\ t;\ \_\ \to\ e\}$

  $\overset{(q)}{\equiv} \ldots \overset{(g)}{\equiv} \ldots$

  $\overset{(q)}{\equiv}\ \texttt{case}\ (K'_1 \ldots (K'_n\ u) \ldots)\ \texttt{of}\ \{$
  $K'_1\ x'\ \to\ \texttt{case}\ x'\ \texttt{of}\ \{(K'_2 \ldots (K'_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}\}$

  $\overset{(g)}{\equiv}\ \texttt{case}\ (K'_1 \ldots (K'_n\ u) \ldots)\ \texttt{of}\ \{(K'_1 \ldots (K'_n\ (c\ p_1 \ldots\ p_m)) \ldots) \to\ t;\ \_\ \to\ e\}$

- *Fall 2:*

$$
\begin{aligned}
&\texttt{case}\ (K_1 \ldots (K_i\ \bot) \ldots)\ \texttt{of}\ \{(K_1 \ldots (K_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}\\
&\overset{(k)}{\equiv} \ldots \overset{(k)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{(K_{i+1} \ldots (K_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}\\
&\overset{(l)}{\equiv} \ldots \overset{(l)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{K_n\ (c\ p_1 \ldots\ p_m)\ \to\ t;\ \_\ \to\ e\}\\
&\overset{(l)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{c\ p_1 \ldots\ p_m\ \to\ t;\ \_\ \to\ e\}\\
&\equiv \bot\\
&\equiv \texttt{case}\ \bot\ \texttt{of}\ \{K'_n\ (c\ p_1 \ldots\ p_m)\ \to\ t;\ \_\ \to\ e\}\\
&\equiv \ldots \equiv \texttt{case}\ \bot\ \texttt{of}\ \{(K'_{i+1} \ldots (K'_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}\\
&\overset{(q)(g)}{\equiv} \ldots \overset{(q)(g)}{\equiv} \ldots\\
&\overset{(q)(g)}{\equiv} \texttt{case}\ (K_1 \ldots (K_i\ \bot) \ldots)\ \texttt{of}\ \{(K'_1 \ldots (K'_n\ (c\ p_1 \ldots\ p_m)) \ldots)\ \to\ t;\ \_\ \to\ e\}
\end{aligned}
$$

- *Fall 3:*

$$
\begin{aligned}
&\texttt{case}\ (K_1 \ldots (K_n\ u) \ldots)\ \texttt{of}\ \{(K_1 \ldots (K_n\ x) \ldots)\ \to\ t;\ \_\ \to\ e\}\\
&\overset{(k)}{\equiv} \ldots \overset{(k)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{x\ \to\ t;\ \_\ \to\ e\}\\
&\overset{(i)}{\equiv} \texttt{case}\ u\ \texttt{of}\ \{x\ \to\ t\}\\
&\overset{(j)}{\equiv} (\lambda\ x\ \to\ t)\ u\\
&\equiv (\lambda\ x'\ \to\ t[x/x'])\ u\\
&\overset{(ß)}{\equiv} (\lambda\ x'\ \to\ t[x/x'])\ ((\lambda\ x''\to\ x''))\ u)\\
&\overset{(j)}{\equiv} (\lambda\ x'\ \to\ t[x/x'])\ \texttt{case}\ u\ \texttt{of}\ \{x''\to\ x''\}\\
&\overset{(q)}{\equiv} (\lambda\ x'\ \to\ t[x/x'])\ \texttt{case}\ K'_n\ u\ \texttt{of}\ \{K'_n\ x\ \to\ x\}\\
&\overset{(q)}{\equiv} \ldots \overset{(q)}{\equiv} (\lambda\ x'\ \to\ t[x/x'])\\
&\texttt{case}\ (K'_1 \ldots (K'_n\ u) \ldots)\ \texttt{of}\ \{(K'_1 \ldots (K'_n\ x) \ldots)\ \to\ x\}\\
&\overset{(d)}{\equiv} \texttt{case}\ (K'_1 \ldots (K'_n\ u) \ldots)\ \texttt{of}\ \{\sim(K'_1 \ldots (K'_n\ x) \ldots) \to\ \ t;\ \_\ \to\ e\}
\end{aligned}
$$

- *Fall 4:*

$$
\begin{aligned}
&\texttt{case}\ (K_1 \ldots (K_i\ \bot) \ldots)\ \texttt{of}\ \{(K_1 \ldots (K_n\ x) \ldots)\ \rightarrow\ t;\ \_\ \rightarrow\ e\} \\
&\overset{(k)}{\equiv} \ldots \overset{(k)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{(K_{i+1} \ldots (K_n\ x) \ldots)\ \rightarrow\ t;\ \_\ \rightarrow\ e\} \\
&\overset{(l)}{\equiv} \ldots \overset{(l)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{K_n\ x\ \rightarrow\ t;\ \_\ \rightarrow\ e\} \\
&\overset{(l)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{x\ \rightarrow\ t;\ \_\ \rightarrow\ e\} \\
&\overset{(i)}{\equiv} \texttt{case}\ \bot\ \texttt{of}\ \{x\ \rightarrow\ t\} \\
&\overset{(j)}{\equiv} (\lambda\, x\ \rightarrow\ t)\ \bot \\
&\equiv (\lambda\, x'\ \rightarrow\ t[x/x'])\ \bot \\
&\overset{(ß)}{\equiv} (\lambda\, x'\ \rightarrow\ t[x/x'])\ ((\lambda\, x''\rightarrow\ x''))\ \bot) \\
&\overset{(j)}{\equiv} (\lambda\, x'\ \rightarrow\ t[x/x'])\ \texttt{case}\ \bot\ \texttt{of}\ \{x''\rightarrow\ x'')\} \\
&\equiv (\lambda\, x'\ \rightarrow\ t[x/x'])\ \texttt{case}\ \bot\ \texttt{of}\ \{K'_n\ x\ \rightarrow\ x\} \\
&\equiv \ldots \equiv (\lambda\, x'\ \rightarrow\ t[x/x'])\ \texttt{case}\ \bot\ \texttt{of}\ \{(K'_i \ldots (K'_n\ x) \ldots)\ \rightarrow\ x\} \\
&\overset{(q)}{\equiv} \ldots \overset{(q)}{\equiv} (\lambda\, x'\ \rightarrow\ t[x/x']) \\
&\texttt{case}\ (K'_1 \ldots (K'_n\ \bot) \ldots)\ \texttt{of}\ \{(K'_1 \ldots (K'_n\ x) \ldots)\ \rightarrow\ x\} \\
&\overset{(d)}{\equiv} \texttt{case}\ (K'_1 \ldots (K'_i\ \bot) \ldots)\ \texttt{of}\ \{\sim(K'_1 \ldots (K'_n\ x) \ldots) \rightarrow\ \ t;\ \_\ \rightarrow\ e\}
\end{aligned}
$$

□

3.9 `let`-Reduktion

Die `let`-Reduktion verschiebt alle Funktionen, die in einem `let`-Konstrukt lokal definiert werden, in den direkt übergeordneten Block. Dabei verlieren lokale Variablen, die außerhalb durch ein λ gebunden werden, ihren Bezug. Das wird vermieden, indem die Funktionen weitere Parameter erhalten, um diesen Bezug wieder herzustellen. Beispielsweise sei folgender Term in einem Haskellprogramm enthalten:

$$\texttt{let}\ \{z\ =\ 3\}\ \texttt{in}\ \lambda\, x\ \rightarrow\ \texttt{let}\ \{r\ y\ =\ y\ *\ x\}\ \texttt{in}\ r\ 5\ +\ z$$

Ein einfaches Verschieben der Funktion $r\ y\ =\ y\ *\ x$ in den übergeordneten `Let`-Block würde zu folgendem fehlerhaften Term führen:

$$\texttt{let}\ \{z\ =\ 3;\ r\ y\ =\ y\ *\ x\}\ \texttt{in}\ \lambda\, x\ \rightarrow\ r\ 5\ +\ z$$

Die Variable x in der Funktion r ist völlig ungebunden und könnte nicht mehr ausgewertet werden und der Term $r\ 5\ +\ z$ erhält dadurch eine abweichende Semantik. Dieses Problem kann vermieden werden, indem die Funktion r um einen Parameter wie folgt erweitert wird:

$$\texttt{let}\ \{z\ =\ 3;\ r\ x\ y\ =\ y\ *\ x\}\ \texttt{in}\ \lambda\, x\ \rightarrow\ r\ x\ 5\ +\ z$$

Nun hat die Variable x innerhalb der Funktion r wieder eine klare Semantik und der Term $r\ x\ 5\ +\ z$ liefert wieder das erwartete Ergebnis. Nach diesem Prinzip arbeitet die `let`-Reduktion, welche wie folgt definiert ist:

$$\frac{\texttt{let}\ \{f_1; \ldots; f_n\}\ \texttt{in}\ t}{t\sigma}$$

wobei

- $\sigma = [v_1/(v_1'\ x_1\ \ldots\ x_j), \ldots, v_l/(v_l'\ x_1\ \ldots\ x_j)]$
- $\{x_1, \ldots, x_j\} = \mathsf{V}_{\mathsf{free}}(f_1) \cup \cdots \cup \mathsf{V}_{\mathsf{free}}(f_l) \setminus \mathsf{V}_{\mathsf{F}}$

gilt und die folgenden Funktion $f_1', \ldots, f_n'$ in denselben Block aufgenommen werden müssen:

$$f_i' = \left(\begin{array}{ll} v_i' :: s_i' & \\ v_i'\ x_1\ \ldots\ x_j\ p_{i,1,1}\ \ldots\ p_{i,k_i,1} & = t_i\sigma \\ & \vdots \\ v_i'\ x_1\ \ldots\ x_j\ p_{i,1,m_i}\ \ldots\ p_{i,k_i,m_i} & = t_i\sigma \end{array} \right)$$

für jede Funktion f_i der Form

$$f_i = \left(\begin{array}{ll} v_i :: s_i & \\ v_i\ p_{i,1,1}\ \ldots\ p_{i,k_i,1} & = t_i \\ & \vdots \\ v_i\ p_{i,1,m_i}\ \ldots\ p_{i,k_i,m_i} & = t_i \end{array} \right)$$

Satz 3.9 *Die* `let`*-Reduktion erhält die Semantik.*

Beweis:
Die `let`*-Reduktion entspricht einem Spezialfall des Lemmas 3.4 (lL)und ist damit schon gültig.* □

3.10 Literalreduktion

Die Literalreduktion überführt die Zahlenliterale aus $\mathbb{Z}$ in ihre Peano-Darstellung unter Verwendung der Konstruktoren `Succ`, `Pred` und `Zero`, des Typs `Int`, während Zeichenliterale aus $\mathbb{A}$ in die mit dem Konstruktor `Char` umschlossene Peano-Darstellung ihrer Position im ASCII-Code übersetzt werden. Außerdem sind die Konstruktoren von `Int` nur innerhalb der Prelude zu sehen (sie werden von der Prelude nicht exportiert), so daß ein Haskellprogramm, welches die Prelude importiert, nicht auf diese zugreifen kann und daher diese ebenfalls definieren und mit eigener Semantik verwenden kann. Für den Typ `Float` ist bis jetzt noch keine Behandlung vorhanden. Die offensichtlichen Darstellungen durch Mantisse und Exponent oder durch einen Bruch (Zähler und Nenner) sind für die spätere Terminierungsanalyse nicht hilfreich und wurden deshalb verworfen, denn die Komplexität der primitiven Funktionen

$$\texttt{primPlusFloat}, \texttt{primMinusFloat}, \texttt{primMulFloat}, \texttt{primDivFloat}, \ldots$$

ermöglicht dabei keine günstigen Abschätzungen. Bis jetzt wurden noch keine sinnvollen Alternativen gefunden, so daß die primitiven Funktionen für den Datentyp `Float` erstmal durch $\bot$ definiert sind. Die Literalreduktion ist formal wie folgt definiert:

$$\frac{n}{\texttt{fromInt}\ (\underbrace{\texttt{Succ}\ \ldots\ (\texttt{Succ}}_{n\text{-mal}}\ \texttt{Zero})\ldots)}$$

$$\frac{-n}{\texttt{fromInt}\ (\underbrace{\texttt{Pred}\ \ldots\ (\texttt{Pred}}_{n\text{-mal}}\ \texttt{Zero})\ldots)}$$

$$\frac{c}{\texttt{Char}\ (\underbrace{\texttt{Succ}\ \ldots\ (\texttt{Succ}}_{m\text{-mal}}\ \texttt{Zero})\ldots)}$$

wobei $n \in \mathbb{N}$, $c \in \mathbb{A}$ und $m = \mathrm{ord}(c)$ gilt. Der Funktionsaufruf von `fromInt` dient dazu, die Peano-Darstellung, die den Ergebnistyp `Int` hat, in beliebige Typen, die

zu den Klassen `Integral` oder `Num` gehören, zu überführen. Jeder dieser Typen hat eine spezielle Instanz der Funktion `fromInt` definiert.

Satz 3.10 (Die Literalreduktion ist semantikerhaltend für $\mathbb{Z}$ und $\mathbb{A}$)
Die Implementierung der Funktionen (siehe Anhang A)

$$\texttt{primPlusInt}, \texttt{primMinusInt}, \texttt{primMulInt}, \texttt{primDivInt}, \ldots$$

mit den Konstruktoren für die Peano-Darstellung stellt einen Isomorphismus zu den ganzen Zahlen mit den Funktionen $+, -, *, /, \ldots$ dar, daher ist die Literalreduktion semantikerhaltend.

Der Beweis, daß unsere Implementierung ein Isomorphismus zu den ganzen Zahlen ist, wird wegen seiner Größe und seiner strukturellen Einfachheit weggelassen.

3.11 Reduktionsstrategie

Die in diesem Kapitel vorgestellten Reduktionen können nicht ganz in beliebiger Reihenfolge angewendet werden, da deren Vorbedingungen jeweils erfüllt sein müssen. In bezug auf die Terminierungsanalyse scheint die Reihenfolge, wie sie im Algorithmus Reduce abgearbeitet wird, günstig zu sein. (Bis jetzt konnte die Qualtität von Reduce nocht nicht ermittelt werden, da die Evaluierung für die gesamte Terminierungsanalyse noch aussteht.) Der Algorithmus Reduce reduziert ein Haskellprogramm, welches den Typ `Float` nicht benutzt, auf ein äquivalentes Haskellprogramm, dessen Funktionen nur Basisterme, Basispatterns und nur bedingten Regeln der Form $|\texttt{True} = t$ enhalten:

Input : Haskellprogramm HP
Output : Haskellprogramm HP
1 λ-Reduktion;
2 `case`-Reduktion;
3 `if`-Reduktion;
4 $\sim p$-Reduktion;
5 _-Reduktion;
6 $x@p$-Reduktion;
7 Bedingungsreduktion;
8 `if`-Reduktion;
9 `newtype`-Reduktion;
10 $\sim p$-Reduktion;
11 `let`-Reduktion;
12 Literalreduktion;
13 **return** HP;

Algorithmus 1 : Reduce

Die Idee des Algorithmus ist es, jede Reduktion solange anzuwenden, bis diese alle Programmkonstrukte, auf denen sie arbeiten kann, eliminiert hat. Die zweite Anwendung der $\sim p$-Reduktion reduziert die $\sim p$-Patterns, die möglicherweise durch die vorausgehende `newtype`-Reduktion entstanden sind. Ähnlich verhält es sich mit der zweiten Anwendung der `if`-Reduktion, denn diese reduziert die `if`-Terme, die durch die Bedingungsreduktion wieder entstehen. Da alle übrigen Reduktionen keine Programmkonstrukte außer Funktionen einführen, sind nach Schritt 3 keine λ-Abstraktionen, `case`-Terme oder `if`-Terme mehr vorhanden. Nach Schritt 6 sind die $\sim p$, `_` oder $x@p$-Patterns beseitigt und nach Schritt 10 gibt es keine Spezial- oder `newtype`-Patterns mehr. Nach Schritt 11 und 12 fallen `let`- und Literalterme weg.

Nach Anwendung dieses Algorithmus kann ein Haskellprogramm nur noch bedingungsfreie Funktionen aus Basispatterns und Basistermen enthalten. Wenn in den folgenden Kapiteln von einem Haskellprogramm HP gesprochen wird, ist es immer durch den Algorithmus Reduce vereinfacht worden.

Die einzelnen Reduktionen wurden innerhalb des AProVE-Projekts ([GTSKF04]) als Prozessoren, wie sie von Martin Mertens in seiner Diplomarbeit mit dem Thema „Modularity, Strategies and Proof Management in Automated Program Verification“ [Mer05] beschrieben sind, implementiert. Ein Strategie-Programm, das diese Prozessoren entsprechend dem Algorithmus Reduce abarbeitet, ist ebenfalls erstellt worden. Syntax und Semantik von Strategie-Programmen sind ebenfalls in [Mer05] beschrieben.

Kapitel 4

Startterm-Analyse

Der übliche Terminierungsbegriff, der bei anderen funktionalen Sprachen oder gar den Termersetzungssystemen in der Art definiert ist, daß jede in einem Programm vorgefundene Funktion mit ihren Unterfunktionen für alle Eingaben zusammen terminiert, ist in Haskell nicht sinnvoll. Viele Funktionen, die allein in der sogenannten Prelude definiert sind, terminieren nicht. Diese werden aber in Programmen verwendet, die an sich schon als terminierend betrachtet werden. Dieser scheinbare Widerspruch ist ein Folge der Auswertungsstrategie von Haskell, welche für bestimmte Terme nicht fordert, bis zu deren Normalformen auszuwerten, denn Haskell benutzt eine sogenannte „lazy“ Auswertungsstrategie. Ob also eine Auswertung eines Funktionsaufrufs zum Stillstand kommt, hängt unter anderem davon ab, wie viele Informationen der Aufrufer benötigt und wie viele Auswertungsschritte dafür ausgeführt werden müssen. Andererseits ist es genauso von Bedeutung, welche Parameter die Funktion übergeben bekommt. Es ist üblich, daß in Programmen Funktionen definiert werden, welche partiell terminieren, während alle anderen Funktionen solche Funktionen nur für bestimmte Parametersätze aufrufen. Durch den Aufrufer wird letztlich bestimmt, welche Teile einer Funktion wirklich relevant werden.
Der sogenannte Aufrufer oder auch Einsprungspunkt in ein Haskellprogramm ist in vielen Haskell-Implementationen der Term, der an die Funktionsvariable `main` gebunden ist. Das heißt, ein Haskellprogramm terminiert, wenn die Auswertung des an `main` gebundenen Terms terminiert. Wir wollen uns hier aber nicht auf diese starre Definition beschränken, sondern lassen beliebige Grundterme als Einsprungspunkte zu. Üblicherweise ist die Terminierung einer Funktion für genau einen Parametersatz, welches einem Grundterm als Einsprungspunkt entspricht, weniger von Bedeutung. Viel mehr ist die Aussage, ob eine Funktion für beliebig komplexe Eingaben terminiert, von Interesse. Das läßt sich am besten durch Variablen in dem Term, der als Einsprungspunkt dient, beschreiben. Eine partielle Terminierungsanalyse schließt diese Idee mit ein, da es keine Verpflichtung gibt, innerhalb des Terms, der als Einsprungspunkt dient, alle Parameter einer Funktion mit Variablen zu belegen.

Beispielsweise sei zu dem Haskellprogramm aus Beispiel 4.1 der Term `double` x als Einsprungspunkt gewählt.

Beispiel 4.1

```
data  Nat      =  Succ Nat | Zero
data  List  a  =  Cons a (List a) | Nil

take  _          Zero      =  Nil
take  Nil        _         =  Nil
take  (Cons y ys)  (Succ n)  =  Cons y (take ys n)

double  Zero      = Zero
double  (Succ x)  = Succ (Succ (double x))

infinity  =  Succ infinity
from x  =  Cons x (from (Succ x))

isZero  Zero      = True
isZero  (Succ x)  = False
```

Die Funktion `double` isoliert betrachtet, terminiert unter der Annahme, daß in die Variable x ein Term, welcher eine Normalform besitzt, eingesetzt wurde. Wenn für x der Term `infinity` angenommen wird, terminiert die Auswertung nicht mehr. Dennoch würden wir gerne annehmen, daß die Funktion `double` terminiert, da nicht die Funktion direkt dafür veranwortlich ist, daß sie auf der Eingabe `infinity` nicht terminiert. Es liegt nämlich daran, daß schon die Auswertung von `infinity` nicht terminiert. Durch die Forderung, daß für Variablen nur terminierende und endliche Terme eingesetzt werden, erreichen wir dieses Ziel. Die Funktion `infinity` kann auch als Parameter für die Funktion `isZero` dienen, so daß der Term `isZero infinity` zu `False` ausgewertet werden kann. Die Funktion `infinity` terminiert also innerhalb des Terms `isZero infinity`, das liegt daran, daß die Funktion `isZero` von ihrem Parameter nur die Information braucht, ob der oberste Konstruktor `Zero` oder `Succ` ist und genau diese kann `infinity` noch liefern ohne, dabei in eine unendliche Auswertung zu geraten. Da aber der Kontext, in dem eine Funktion später angewendet werden soll, nicht bekannt ist, kann nicht behauptet werden, daß `infinity` allgemein terminiert. Es muß für unseren Terminierungsbegriff also noch gefordert werden, daß der Term, der als Einsprungspunkt dient, vollständig ausgewertet wird, so daß er die maximale Forderung nach Informationen des Aufrufers erfüllen kann. Die Forderung nach einer vollständigen Auswertung eines Terms wird auch hyper termination genannt. Die beiden vorgestellten Forderungen werden zusammengefaßt und präzisiert durch den Begriff der NF-Terminierung eines Terms, welcher in der Definition 4.5 eingeführt wird. Als Grundlage dafür werden vorher die normale Auswertungsfunktion von Haskell und deren Erweiterung auf Terme mit freien Variablen erklärt.

Definition 4.1 (Auswertungsfunktion) *Die Auswertungsfunktion*

$$\text{evaluate}_{\mathsf{HP}} : \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V}) \longrightarrow \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V})$$

führt genau einen Auswertungsschritt anhand der Haskellauswertungsstrategie unter der Verwendung des Haskellprogramms HP *aus. Sie liefert den neuen Haskellterm zurück, der durch genau eine Regelanwendung auf den aktuellen Redex*[1] *innerhalb der Eingabe entsteht. Für den Fehlerfall, also falls keine Regel matcht, wird die spezielle Funktionsvariable* $\underline{\text{error}} \in \mathsf{V_F}$ *ausgegeben.* $\underline{\text{error}}$ *hat das Typschema* $\emptyset \Rightarrow a$ *und ist so für jeden Typ vorhanden. Für einen Term* $C[\underline{\text{error}}]$*, dessen Redex* $\underline{\text{error}}$ *ist, liefert* $\text{evaluate}_{\mathsf{HP}}$ *ebenfalls* $\underline{\text{error}}$ *zurück.*

Definition 4.2 (Auswertungsrelation) *Die Auswertungsrelation* $\rightarrow_{\mathsf{HP}}$ *für ein Haskellprogramm* HP *ist folgendermaßen definiert:*

$$\rightarrow_{\mathsf{HP}} := \{(t_1, t_2) \in \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V}) \times \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V}) \mid \text{evaluate}_{\mathsf{HP}}(t_1) = t_2\}$$

Statt $(t_1, t_2) \in \rightarrow_{\mathsf{HP}}$ *wird die übliche Notation* $t_1 \rightarrow_{\mathsf{HP}} t_2$ *verwendet. Außerdem wird mit* $t{\downarrow_{\mathsf{HP}}}$ *die Normalform des Terms* $t \in \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V})$ *bezüglich der Auswertungsrelation* $\rightarrow_{\mathsf{HP}}$ *bezeichnet, sofern diese existiert.*

Definition 4.3 (Redex) *Wenn* $t = t[h\ t_1\ \ldots\ t_n]_\pi \rightarrow_{\mathsf{HP}} t[h'\ t'_1\ \ldots\ t'_m]_\pi$ *gilt und eine Regel der an* $h \in \mathsf{V_F}$ *gebundenen Funktion angewendet wurde, so bezeichnen wir den Term* $t|_\pi$ *als Redex von* t*.*

Für einen Redex $h\ t_1\ \ldots\ t_n$ mit $h \in \mathsf{V_F}$ ist zu beachten, daß es dabei gleichgültig ist, welche Stelligkeit die angewendete Regel r hat, es ist also $arity(r) < n$ möglich.

Zu dem Term `take` (`Cons Zero Nil`) (`double` (`Succ Zero`)) und dem Haskellprogramm aus Beispiel 4.1 ergibt sich beispielsweise folgende Auswertung:

Beispiel 4.2

```
take (Cons Zero Nil) (double (Succ Zero))
 →HP take (Cons Zero Nil) (Succ (Succ (double Zero)))        (1)
 →HP Cons Zero (take Nil (Succ (double Zero)))               (2)
 →HP Cons Zero Nil                                           (3)
```

Durch Auswerten des Redex (`double` (`Succ Zero`) entsteht in Schritt (1) des Beispiels 4.2 der Term `Succ` (`Succ` (`double Zero`)), da die erste Regel von `take` nur angewendet werden kann, wenn der zweite Parameter von `take` auf `Zero` matcht.

[1] reducible expression

Um das zu prüfen, muß der Redex `double` (`Succ Zero`) soweit ausgewertet werden, bis eine Entscheidung möglich wird. Der Term `Succ` (`Succ` (`double Zero`)) ermöglicht diese Entscheidung und wird deswegen erst einmal nicht weiter ausgewertet. Im Schritt (2) wird nun wieder überprüft, welche `take`-Regel angewendet werden kann. Die dritte Regel matcht auf die übergebenen Parameter. Für Schritt (3) wird dann die zweite Regel von `take` benutzt. Der Term `double Zero` wurde nicht weiter ausgewertet, weil sein Wert für den umschließenden Kontext nicht weiter relevant war. Der Term `Cons Zero Nil` kann nicht weiter ausgewertet werden und ist damit eine Normalform bezüglich $\rightarrow_{\mathsf{HP}}$.

Definition 4.4 (Erweiterte Auswertungsfunktion) *Die erweiterte Auswertungsfunktion*

$$\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}} : \mathsf{H_B}(\mathsf{D},\mathsf{V}) \longrightarrow \mathsf{H_B}(\mathsf{D},\mathsf{V})$$

ist auf Grundtermen genauso definiert wie $\text{evaluate}_{\mathsf{HP}}$*. Für den Fall, daß der Redex die Form* $x\ t_1\ \ldots\ t_n$ *mit* $x \in \mathsf{V_L}$ *und* $t_i \in \mathsf{H_B}(\mathsf{D},\mathsf{V})$ *hat, wird* $\underline{\text{error}}\ x\ t_1\ \ldots\ t_n$ *zurückgeliefert. Ansonsten werden freie lokale Variablen wie normale Terme behandelt. Falls für eine Member-Variable* v *einer Klasse die konkrete Instanz nicht ermittelt werden kann, scheitert* $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}$ *mit* $\underline{\text{error}}\ v$*. Außerdem ist die Auswertungsrelation* $\rightarrow^{\mathsf{EXT}}_{\mathsf{HP}}$ *analog zu* $\rightarrow_{\mathsf{HP}}$ *definiert.*

Mit Hilfe der erweiterten Auswertungsfunktion können nun auch Terme, die lokale Variablen beinhalten, ausgewertet werden, wobei ein Fehler zurückgegeben wird, wenn bei einem normalen Auswertungsschritt der Wert einer lokalen Variablen benötigt wird. Der letzte Fall von $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}$ muß aufgenommen werden, weil die Eingabeterme möglicherweise nicht mehr ausreichend instantiierte Typen haben, um die konkrete Instanz einer Member-Variablen zu ermitteln. Die freien lokalen Variablen dürfen jeden passenden Typ annehmen, so auch den allgemeinsten, der an ihren Positionen erlaubt ist. Beispielsweise soll der Term $(==)\ x\ y$ ausgewertet werden. Voll annotiert würde der Term folgendermaßen aussehen:

$$((==)_{|a\to a\to \texttt{Bool}|}\ x_{|a|}\ y_{|a|})_{|\texttt{Bool}|}$$

Der Redex wäre an Position ϵ. Nun müßte die richtige Instanz für $(==)$ ausgewählt werden. Der Typ dieses Terms ermöglicht diese Entscheidung aber nicht, so daß $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}$ mit $\underline{\text{error}}\ (==)$ abbricht. Im Abschnitt 4.1 werden wir sehen, wie mit der erweiterten Auswertungsfunktion Narrowing-Bäume und Narrowing-Graphen aufgebaut werden können.

Definition 4.5 (NF-Terminierung) *Die Menge der NF-terminierenden Grundterme* $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}} \subseteq \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D},\mathsf{V})$ *ist die kleinste Menge, für die gilt:*

$$\begin{aligned}
&t_{|\tau|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}\ \textit{gdw.}\\
&(t_{|\tau|}\ \textit{typkorrekt})()\\
&\wedge\ (t_{|\tau|}{\downarrow_{\mathsf{HP}}}\ \ \textit{existiert})()\\
&\wedge\ \big((\tau = \tau' \to \tau'') \Rightarrow (\forall {t'}_{|\tau'|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}\colon (t_{|\tau'\to\tau''|}\ {t'}_{|\tau'|})_{|\tau''|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}})\big)
\end{aligned}$$

Die Menge der NF-terminierenden Terme $\mathsf{NF}_{\mathsf{HP}} \subseteq \mathsf{H}_{\mathsf{B}}(\mathsf{D},\mathsf{V})$ *ist definiert wie folgt:*

$$t \in \mathsf{NF}_{\mathsf{HP}} \textit{ gdw. } \big((\forall x \in \mathsf{V}(t)\text{: } x\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}) \wedge t\sigma \textit{ typkorrekt} \Rightarrow t\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}\big)$$
$$\textit{für alle Substitutionen } \sigma \textit{ gilt}$$

Wenn ein Term typkorrekt ist, sind alle Instanzen der Klassen, die er für seine Auswertung benötigt in dem Haskellprogramm vorhanden. Das heißt jede Typannotation von Member-Variablen, die innerhalb seiner Auswertung auftreten, sind soweit konkretisiert, daß immer *eine für sie passende Instanz einer Klasse aus dem Haskellprogramm gewählt werden kann. Ansonsten hätte der Typchecker diesen Term als nicht typkorrekt erkannt.*
Zusätzlich wird eine Substitution σ *eine* $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*-Substitution genannt, wenn für diese* $\forall x \in \mathsf{V}_{\mathsf{L}}\text{: } x\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt.*

Mit Hilfe der NF-Terminierung können wir den intuitiven Begriff der Terminierung eines Haskellprogramms mit Einsprungspunkt (oder Startterm) wie folgt definieren:

Definition 4.6 (Startterm-Terminierung eines Haskellprogramms) *Ein Startterm* $t \in \mathsf{H}_{\mathsf{B}}(\mathsf{D},\mathsf{V})$ *zu einem Haskellprogramm* HP *terminiert genau dann, wenn er ein Element der Menge* $\mathsf{NF}_{\mathsf{HP}}$ *ist.*

Ebenso wie die Auswertungsfunktion, muß der Haskell-Typchecker auf Terme erweitert werden, in denen freie Variablen auftreten. Die Startterme können so vorher auf ihre Typkorrektheit geprüft werden.

Definition 4.7 (Erweiterte Typchecker) *Der erweiterte Typchecker* $\text{typeChecker}_{\mathsf{HP}} : \mathsf{H}_{\mathsf{B}}(\mathsf{D},\mathsf{V}) \mapsto (\text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \times \mathsf{H}_{\mathsf{B}}(\mathsf{D},\mathsf{V}))$ *bildet jeden Term auf seine impliziten Klassenbedingungen und den strukturell identischen mit dem allgemeinsten Typ bezüglich des Haskellprogramms* HP *annotierten Term ab. Die Annotationen des Eingabeterms werden dabei vollständig ignoriert. Anders als der normale Typchecker, kann der erweiterte Typchecker auch mit freien Variablen innerhalb eines Terms umgehen. Diese bekommen den allgemeinsten Typ den sie an der Position, an der sie im Eingabeterm stehen, annehmen können. Falls ein Term nicht typkorrekt ist, wird vom erweiterten Typchecker* $(\emptyset, \underline{\text{error}})$ *zurückgegeben.*

Zum Beispiel gibt der kontextfreie Typchecker zu dem Term

$$(\texttt{equal}_{|\texttt{List Nat}\to\texttt{List Nat}\to\texttt{Bool}|}\ \texttt{x}_{|\texttt{List Nat}|})_{|\texttt{List Nat}\to\texttt{Bool}|}$$

folgendes Ergebnis:

$$\begin{aligned}&\text{typeChecker}_{\mathsf{HP}}((\texttt{equal}_{|\texttt{List Nat}\to\texttt{List Nat}\to\texttt{Bool}|}\ \texttt{x}_{|\texttt{List Nat}|})_{|\texttt{List Nat}\to\texttt{Bool}|})\\ &= (\{\texttt{Equal}\ a\}, (\texttt{equal}_{|a\to a\to\texttt{Bool}|} x_{|a|})_{|a\to\texttt{Bool}|})\end{aligned}$$

4.1 Narrowing-Graph

Nachdem im letzten Abschnitt die Startterm-Terminierung in der Definition 4.6 beschrieben worden ist, stellt sich die Frage, wie nun genau ein Startterm auf seine NF-Terminierung hin überpüft werden kann. Die Idee, einfach alle Funktionen, die der Startterm benutzt, auf Terminierung zu überprüfen, führt hier, wie bereits zu Kapitelanfang erwähnt, nicht weiter, weil allein der einfache NF-terminierende Term `take` (`from` i) n für das Haskellprogramm aus Beispiel 4.1 diesen Ansatz als unbrauchbar entlarvt, da schon die Funktion `from` nicht terminiert.

Ein anderer Ansatz ist es, den Startterm schon einmal ein paar Schritte auszuwerten und dabei genauer zu analysieren, wie dieser sich dabei verhält. Natürlich kann ein Startterm nicht mit der normalen Auswertungsfunktion von Haskell ausgewertet werden, wenn dieser noch nicht instantiierte Variablen enthält. Die erweiterte Auswertungsfunktion ist dazu aber in der Lage und liefert, falls bei einem Auswertungsschritt der Wert einer Variablen $x_{|\tau|}$ benötigt wird, error $x_{|\tau|}$ zurück. In so einem Fall kann nun $x_{|\tau|}$ probehalber einmal mit jedem Konstruktor, der für den Typ τ zur Verfügung steht, instantiiert werden. Wenn dabei einer der Konstruktoren Parameter benötigt, werden dort frische Variablen eingesetzt. Jetzt kann wieder für jede dieser Instanzen ein Auswertungsschritt probiert werden, möglicherweise muß noch einmal eine solche Fallunterscheidung getroffen werden und so weiter. Der Typ τ kann nicht der Funktionstyp oder der allgemeine Typ a sein, weil zu diesen keine Patterns existieren und so deren Werte nicht für das Patternmatching und damit auch nicht für einen Auswertungsschritt benötigt werden. Durch diesen Vorgang, der auch als Narrowing bezeichnet wird, entsteht ein Narrowing-Baum, an dessen Wurzel unser Startterm steht.
Ein Anfangsstück eines solchen Baums für den Startterm `take` (`from` i) n zu dem Haskellprogramm aus Beispiel 4.1 ist in Abbildung 4.1 zu sehen. Auf Knoten `a` wurde eine Fallunterscheidung angewendet und für die Terme der Knoten `b`, `c` und `e` wurde einmal die Auswertungsfunktion angewendet. Der Knoten `d` ist eine Normalform und braucht nicht weiter betrachtet zu werden, während für Knoten `f` wiederum eine Fallunterscheidung angewendet werden müßte. Es ist zu erkennen, daß solch ein Baum das Auswertungsverhalten eines Startterms perfekt widerspiegelt. Außerdem NF-terminiert ein Term eines Knotens genau dann, wenn alle seine Kinder NF-terminieren. Wenn wir es nun schaffen, den Narrowing-Baum vollständig aufzubauen, so NF-terminiert jedes Blatt, da deren Kinder trivial NF-terminieren, und so ergibt sich, daß alle Väter NF-terminieren und so auch der Startterm. Der Nachteil dieser Narrowing-Bäume ist jedoch, daß diese für die meisten Startterme unendlich sind und es nicht klar ist, bis zu welcher Tiefe diese dann aufgebaut werden sollen. Die Idee, diese Bäume bis zu einer bestimmten Tiefe aufzubauen, bringt uns nur wieder zu der ursprünglichen Fragestellung zurück, denn dann muß die NF-Terminierung der Terme an diesen Blättern nach-

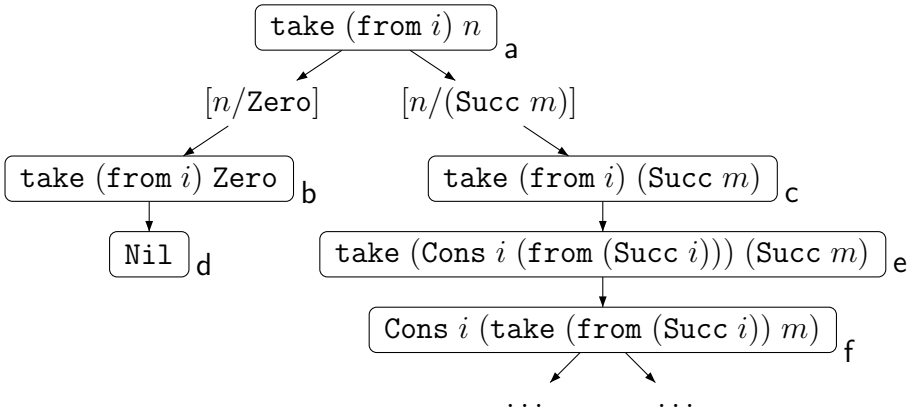

Abbildung 4.1: Narrowing-Baum zu Beispiel 4.1 und Startterm take (from i) n

gewiesen werden. Von einer NF-Terminierung dieser Blatterme kann nicht einfach ausgegangen werden, da diese im vollständigen Narrowing-Baum möglicherweise doch Kinder haben. Die mögliche Unendlichkeit der Narrowing-Bäume ist deren grundsätzliches Problem. Dieser Nachteil kann durch die Verallgemeinerung der Narrowing-Bäume auf Narrowing-Graphen umgangen werden. Narrowing-Graphen sind ebenfalls eine Erweiterung der Termination Tableaux, wie sie von Sven Eric Panitz und Manfred Schmidt-Schauß in [PSS97] vorgestellt werden. Ein ähnlicher Ansatz mit Derivation Trees wurde von Olivier Fissore, Isabelle Gnaedig und Hélène Kirchner in [FGK02] beschreiben sind. Der Narrowing-Graph berücksichtigt im Gegensatz zu den Tableaux die Typen und die zugehörigen Klassenbedingungen der Terme. Außerdem können in einem Narrowing-Graph beliebig ineinander verschränkte Zyklen vorkommen, während diese in den Termination Tableaux zu Problemen führen, wie in [Pan96] auf Seite 21 beschrieben ist.

Definition 4.8 (Narrowing-Graph) *Ein Narrowing-Graph ist der 7-Tupel*

$$\mathsf{NG} := \langle \mathsf{Q}, \mathsf{M}, \text{case}, \text{subterm}, \text{eval}, \text{varexp}, \text{generalisation} \rangle$$

wenn

- $\mathsf{Q} \subseteq \mathsf{CC}(\mathsf{D}, \mathsf{V}) \times \mathsf{H}_\mathsf{B}(\mathsf{D}, \mathsf{V})$,
- $\mathsf{M} : \mathsf{Q} \longrightarrow \{\underline{\text{case}}, \underline{\text{eval}}, \underline{\text{varexp}}, \underline{\text{parsplit}}, \underline{\text{instance}}\}$,
- $\text{case} \subseteq \mathsf{Q} \times \mathsf{SUBS}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q}$,
- $\text{subterm} \subseteq \mathsf{Q} \times (\mathsf{V}_\mathsf{L} \cup \mathbb{N}) \times \mathsf{Q}$,
- $\text{eval} : \mathsf{Q} \longrightarrow \mathsf{Q}$,
- $\text{varexp} : \mathsf{Q} \longrightarrow \mathsf{Q}$ *und*

- generalisation : $\mathsf{Q} \longrightarrow \mathsf{Q}$

gilt. Die Funktion M *weist jedem Knoten eine Markierung zu, während die Mengen* case*(Fallunterscheidungskanten) und* subterm*(Teiltermkanten) beschriftete Kanten repräsentieren. Mit den partiellen Funktionen* eval *(Auswertungskanten),* varexp*(Variablenexpansionskanten) und* generalisation*(Generalisierungskanten) werden ebenfalls weitere Arten von Kanten dargestellt. Die Bedeutung der einzelnen Kantenarten wird später bei den einzelnen Transformationen, die den Narrowing-Graph aufbauen, vorgestellt. Außerdem wird ein Knoten* $(\underline{cs}, u) \in \mathsf{Q}$ *genau dann als NF-terminierend bezeichnet, wenn sein Term* u *NF-terminiert (kurz:* $u \in \mathsf{NF_{HP}}$*).*

Das Anfangsstück des Narrowing-Baums aus Beispiel 5.2 kann als Narrowing-Graph NG_{ex} wie folgt dargestellt werden:

$$\mathsf{NG} := \langle \mathsf{Q}_{ex}, \mathsf{M}_{ex}, \text{case}_{ex}, \emptyset, \text{eval}, \bot, \bot \rangle$$

wobei

- $\mathsf{M}_{ex} = \bot[\mathsf{a} \mapsto \underline{\text{case}}]$
  $[\mathsf{b} \mapsto \underline{\text{eval}}]$
  $[\mathsf{c} \mapsto \underline{\text{eval}}]$
  $[\mathsf{d} \mapsto \underline{\text{parsplit}}]$
  $[\mathsf{e} \mapsto \underline{\text{eval}}]$
  $[\mathsf{f} \mapsto \underline{\text{parsplit}}]$,
- $\text{case}_{ex} = \{(\mathsf{a}, [n/\texttt{Zero}], \mathsf{b}),$
  $(\mathsf{a}, [n/(\texttt{Succ}\ m)], \mathsf{c})\}$ und
- $\text{eval}_{ex} = \bot[\mathsf{b} \mapsto \mathsf{d}][\mathsf{c} \mapsto \mathsf{e}][\mathsf{e} \mapsto \mathsf{f}]$

gilt. Der Operator $[a \mapsto b]$ ersetzt in einer Abbildung f das Bild c von a (also $c = f(a)$) durch das Bild b, so daß $f[a \mapsto b](a) = b$ gilt. Die Knoten c und f bekommen die Markierung $\underline{\text{parsplit}}$, um anzudeuten, daß auf diese eine Parameteraufteilung, wie sie später noch eingeführt wird, angewendet wurde. Da auf den Knoten a eine Fallunterscheidung angwendet wurde, wird dieser mit der Markierung $\underline{\text{case}}$ versehen. Außerdem wurden zwei Fallunterscheidungskanten zu den Nachfolgern b und c gezogen, welche mit den Substitutionen $[n/\texttt{Zero}]$ und $[n/(\texttt{Succ}\ m)]$ beschriftet sind. Da an den Knoten b, c und e jeweils ein Auswertung stattfand, sind diese mit $\underline{\text{eval}}$ markiert. Sie sind jeweils durch eine Auswertungskante mit ihren Nachfolgern verbunden.

Im folgenden wird ein Verfahren vorgestellt, welches den Narrowing-Graph, ähnlich wie einen Narrowing-Baum, schrittweise durch Transformationen aufbaut. Als Ausgangspunkt dafür dient ein Narrowing-Graph in Startkonfiguration, welche für jeden Startterm existiert.

Definition 4.9 (Startkonfiguration) *Für einen Startterm u ist der Narrowing-Graph NG in Startkonfiguration, wenn*

$$\mathsf{NG} = \langle\{\{(\underline{cs}', u')\}, \bot, \emptyset, \emptyset, \bot, \bot, \bot\rangle$$

und $(\underline{cs}', u') = \text{typeChecker}_{\mathsf{HP}}(u)$ gilt. Der Knoten $(\underline{cs}', u')$ einer Startkonfiguration wird auch Startknoten genannt. In einer Startkonfiguration gibt es also keine Kanten und Markierungen und die Knotenmenge enthält nur den Startknoten.

In den folgenden Abschnitten werden die Transformationen, die für den Aufbau eines Narrowing-Graphen benötigt werden, vorgestellt. Diese Transformationen betrachten einen Knoten und erzeugen zu diesem eventuell neue Nachfolgerknoten. Die Idee dabei ist, wie im Narrowing-Baum, daß ein Knoten dann NF-terminiert, wenn alle seine Nachfolger NF-terminieren und so das Problem der NF-Terminierung eines Terms auf die Terme in den Nachfolgerknoten verteilt wird. Die Transformationen sind, wie wir noch sehen werden, so definiert, daß die Terme in den Nachfolgern „einfacher“ werden. Das Ziel ist es nun, auf jeden Knoten des Narrowing-Graphen eine Transformation anzuwenden, wobei jede Transformation den Knoten, auf den sie angewendet wird, mit ihrem Kürzel aus der Menge $\{\underline{\text{case}}, \underline{\text{eval}}, \underline{\text{varexp}}, \underline{\text{parsplit}}, \underline{\text{instance}}\}$ markiert, um später eine einfachere Kontrolle ihrer Anwendung zu ermöglichen. Bei Anwendung der einzelnen Transformationen entstehen möglicherweise wieder neue Knoten, auf die wiederum Transformationen angewendet werden müssen und so weiter. Unter den Transformationen gibt es aber die Instantiierung, welche schon vorhandene Knoten als Nachfolger für den aktuellen wählen, und so dem unendlichen Aufbauprozess des Narrowing-Graphen entgegenwirken kann. Dieser Schritt ist so in einem Narrowing-Baum unmöglich, und das ist der entscheidende Vorteil der Narrowing-Graphen. Dazu muß aber die Reihenfolge, in der die Transformationen angewendet werden, entsprechend gewählt werden, denn die Transformationen bilden zusammen kein terminierendes Transformationssystem. Die Beschreibung der richtigen Transformationsfolge wird in das spätere Kapitel 6 verschoben, weil zu deren Auswahl bestimmte Aspekte einfließen werden, welche erst im Rückblick als sinnvoll erkannt werden können. Bei richtiger Wahl der Transformationsfolge, erhalten wir für jedes Haskellprogramm und dessen zugehörigem Startterm nach endlich vielen Transformationsschritten immer einen Narrowing-Graph der im folgenden beschriebenen geschlossenen Form, wie uns das Abschlußlemma 4.10 auf Seite 83 bestätigen wird.

Definition 4.10 (Geschlossene Narrowing-Graphen) *Ein Narrowing-Graph*

$$\mathsf{NG} := \langle \mathsf{Q}, \mathsf{M}, \text{case}, \text{subterm}, \text{eval}, \text{varexp}, \text{generalisation}\rangle$$

wird genau dann als geschlossen bezeichnet, wenn $\text{Dom}(\mathsf{M}) = Q$ gilt und die Menge Q endlich ist. Die geschlossenenen Narrowing-Graph entsprechen den Preclosed Termination Tableaux, wie sie in [PSS97] definiert sind.

Der geschlossene Narrowing-Graph besitzt die gewünschten Eigenschaften der Endlichkeit und zeigt, daß auf jeden Knoten eine Transformation angewendet wurde. Später wird im Kapitel 5 gezeigt, wie mit Hilfe eines geschlossenen Narrowing-Graphen die NF-Terminierung eines Startterms nachgewiesen werden kann. Aber nun kommen wir erst einmal zu den einzelnen Transformationen zurück, welche den Narrowing-Graph schrittweise erstellen und in die geschlossene Form überführen. Zu jeder Transformation wird ihre Korrektheit in einem Lemma nachgewiesen, wobei die nachgewiesene Eigenschaft oftmals stärker ist, als die Zurückführung der NF-Terminierung eines Knotens auf die NF-Terminierung seiner Nachfolger. Hierbei bleibt die Erweiterungs-Transformation außen vor, da diese als einzige keine neuen Kanten erstellen kann und so keinen Zurückführungsnachweis braucht.

Um die Transformationen definieren zu können, wird die Funktion truncate zur Kürzung von Klassenbedingungen benötigt. Denn die Klassenbedingungen eines Knotens können in dessen Nachfolgerknoten ihren Bezug verlieren, da der Term eines Nachfolgers möglicherweise nicht mehr die Typvariablen besitzt, die in einer Klassenbedingung vom Ausgangsknoten auftreten, so daß diese Klassenbedingungen nutzlos wird.

Definition 4.11 (Kürzen von Klassenbedingungen) *Die Funktion*

$$\text{truncate} :: \text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \times \mathsf{H}_\mathsf{B}(\mathsf{D},\mathsf{V}) \longrightarrow \text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V}))$$

kürzt eine Menge von Klassenbedingungen auf solche, die von einem gegebenen Term gedeckt werden. Dabei ist ein Klassenbedingung durch einen Term gedeckt, wenn sie nur Typvariablen enthält, die in den Typannotationen des Terms vorkommen. Die Funktionen truncate *ist folgendermaßen definiert:*

$$\text{truncate}(\underline{cs}, t) := \{(c\ \tau) \in \underline{cs} \mid \mathsf{V}(\tau) \subseteq \bigcup_{u_{|\tau'|} \trianglelefteq t} \mathsf{V}(\tau')\}$$

Beispielsweise ergibt sich für die Menge $\{\texttt{Equal}\ a, \texttt{Equal}\ b\}$ von Klassenbedingungen nach einer Kürzung auf den Term $\texttt{Cons}_{|a \to \texttt{List}\ a \to \texttt{List}\ a|}\ x_{|a|}\ \texttt{Nil}_{|\texttt{List}\ a|}$ folgende Menge von Klassenbedingungen:

$$\text{truncate}(\{\texttt{Equal}\ a, \texttt{Equal}\ b\}, \texttt{Cons}_{|a \to \texttt{List}\ a \to \texttt{List}\ a|}\ x_{|a|}\ \texttt{Nil}_{|\texttt{List}\ a|}) = \{\texttt{Equal}\ a\}$$

Die Klassenbedingung $\texttt{Equal}\ b$ ist nicht gedeckt, weil die Typvariable b nicht in den Typannotationen des Terms $\texttt{Cons}_{|a \to \texttt{List}\ a \to \texttt{List}\ a|}\ x_{|a|}\ \texttt{Nil}_{|\texttt{List}\ a|}$ vorkommt.

4.1.1 Auswertung

Durch die Auswertung erhält man zu einem Knoten $(\underline{cs}, u) \in \mathsf{Q}$ einen weiteren Knoten $(\underline{cs}', u')$, dessen Term u' durch einen Auswertungsschritt aus dem Term u hervorgegangen ist. Der Term u NF-terminiert nämlich genau dann, wenn auch u' NF-terminiert. Formal läßt sich die Auswertung wie folgt definieren:

$$\frac{\langle \mathsf{Q} \uplus \{q\}, \mathsf{M}[q \mapsto \bot], \ldots, \mathrm{eval}, \ldots \rangle}{\langle \mathsf{Q} \uplus \{q, q'\}, \mathsf{M}[q \mapsto \underline{\mathrm{eval}}], \ldots, \mathrm{eval}[q \mapsto q'], \ldots \rangle}$$

wobei

- $q = (\underline{cs}, v\ u_1\ \ldots\ u_n)$,
- $q' = (\mathrm{truncate}(\underline{cs}, u'), u')$,
- $u' = \mathrm{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(v\ u_1\ \ldots\ u_n)$,
- $u', u_1, \ldots, u_n \in \mathsf{H_B}(\mathsf{D}, \mathsf{V})$,
- u' nicht der Form $\underline{\mathrm{error}}\ t$ für einen Term $t \in \mathsf{H_B}(\mathsf{D}, \mathsf{V})$ ist und
- $v \in \mathsf{V_F}$

gilt. Die Klassenbedingungen $\underline{cs}$ werden auf $\mathrm{truncate}(\underline{cs}, u')$ gekürzt, weil durch den Auswertungsschritt möglicherweise alle Teilterme, die einige Klassenbedingungen aus $\underline{cs}$ decken würden, wegfallen. Die Funktion truncate beseitigt alle ungedeckten Klassenbedingungen. Das Ergebnis von $\mathrm{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(v\ u_1\ \ldots\ u_n)$ soll nicht $\underline{\mathrm{error}}\ t$ sein. Es soll also keinen Fehler der erweiterten Auswertungsfunktion sein, während alle Ergebnisse der normalen Auswertungsfunktion erlaubt sind und damit auch $\underline{\mathrm{error}}$. Außerdem darf der Term des Knotens q nur mit einer Funktionsvariablen beginnen, damit die Auswertung nicht auch angewendet werden kann, wenn eigentlich eine Parameteraufteilung nötig ist.

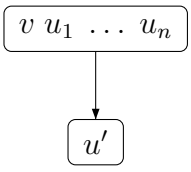

Abbildung 4.2: Auswertung

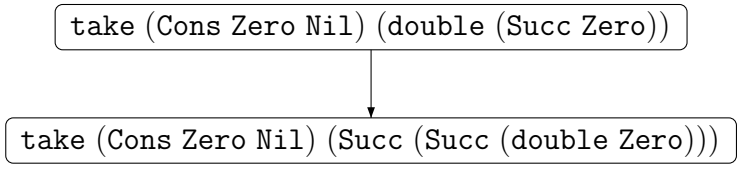

Abbildung 4.3: Beispiel einer Auswertung

In Abbildung 4.2 sehen wir den allgemeinen Teilgraph ab Knoten $(\underline{cs}, v\ u_1\ \ldots\ u_n)$ der durch eine Auswertung entsteht. Ein Beispiel einer Auswertung für den Knoten $(\emptyset, \texttt{take}\ (\texttt{Cons Zero Nil})\ (\texttt{double}\ (\texttt{Succ Zero})))$, dessen Term zu dem Haskellprogramm aus Beispiel 4.1 gehört ist in Abbildung 4.3 zusehen. Dabei wird der Term

$$\texttt{take}\ (\texttt{Cons Zero Nil})\ (\texttt{double}\ (\texttt{Succ Zero}))$$

zu dem Term

$$\texttt{take}\ (\texttt{Cons Zero Nil})\ (\texttt{Succ}\ (\texttt{Succ}\ (\texttt{double Zero})))$$

des Nachfolgerknotens ausgwertet. Außerdem ist zu eval ist die Auswertungskante zwischen den beiden Knoten hinzugekommen.

Lemma 4.1 (Korrektheit der Auswertung)
$(u \to_{\mathsf{HP}}^{\mathsf{EXT}} u') \Rightarrow (u'\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \Rightarrow u\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}})$

Beweis:
Wir zeigen die Aussage per Induktion über die Typgröße. Sei nun $u'\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$*, so existiert* $u'\sigma{\downarrow_{\mathsf{HP}}}$*. Wegen* $u\sigma \in \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V})$ *und* $u \to_{\mathsf{HP}}^{\mathsf{EXT}} u'$*, gilt* $u\sigma \to_{\mathsf{HP}} u'\sigma$*. Da* $\to_{\mathsf{HP}}$ *eindeutig ist, folgt so* $u'\sigma{\downarrow_{\mathsf{HP}}} = u\sigma{\downarrow_{\mathsf{HP}}}$*. Wenn* $u\sigma$ *nicht von einem Funktionstyp ist, sind wir laut Definition 4.5 fertig. Ansonsten sei der Term* $u'\sigma\ t$ *für einen Term* $t \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$ *typkorrekt. So gilt wegen* $u'\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$ *auch* $u'\sigma\ t \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$*. Da* $\to_{\mathsf{HP}}$ *den Typ eines Terms nicht modifiziert, sind* $u\sigma$ *und* $u'\sigma$ *vom selben Typ. Zusätzlich gilt* $u'\sigma\ t = (u\ t)\sigma$ *und* $u'\sigma\ t = (u\ t)\sigma$*, weil* $t \in \mathsf{H}_{\mathsf{B}}^{\mathsf{G}}(\mathsf{D},\mathsf{V})$*. Weil* $\to_{\mathsf{HP}}^{\mathsf{EXT}}$ *den Redex soweit links und dort soweit außen wie möglich sucht, folgt, daß der Term* $u\ t$ *immer erst zu* $u'\ t$ *ausgewertet wird, und so folgt* $(u\ t) \to_{\mathsf{HP}}^{\mathsf{EXT}} (u'\ t)\sigma$*. Mit* $(u'\ t)\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$*, und weil der Typ von* $(u'\ t)\sigma$ *kleiner als der von* $u'\sigma$ *ist, folgt per Induktionshypothese, daß* $(u\ t)\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$*. Zusammen ergibt sich:*

$$u\sigma{\downarrow_{\mathsf{HP}}}\ \textit{existiert}\ \wedge (u\ t)\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$$

also $u\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}$*.* □

4.1.2 Parameteraufteilung

Die Parameteraufteilung erzeugt zu einem Knoten $(\underline{cs}, h\ u_1\ \ldots\ u_n) \in \mathsf{Q}$ mehrere Knoten, die mit den Parametern vom Kopf h beschriftet sind. Da h aus $\underline{\mathsf{Cons}_\mathsf{D}} \cup \mathsf{V}_\mathsf{L}$ ist, sind für das Terminierungsverhalten des Terms $h\ u_1\ \ldots\ u_n$ nur die Parameter $u_1, \ldots, u_n$ von Bedeutung. Die Parameteraufteilung ist die einzige Möglichkeit, wie Blätter in einem Narrowing-Graphen entstehen können, dieser Fall tritt ein, wenn h keine Parameter besitzt. Formal läßt sich die Parameteraufteilung folgendermaßen definieren:

$$\frac{\langle \mathsf{Q} \uplus \{q\}, \mathsf{M}[q \mapsto \bot], \ldots, \text{subterm}, \ldots \rangle}{\langle \mathsf{Q} \uplus \{q, q_1, \ldots, q_n\}, \mathsf{M}[q \mapsto \underline{\text{parsplit}}], \ldots, \text{subterm}', \ldots \rangle}$$

wobei

- $\text{subterm}' = \text{subterm} \uplus \{(q, 1, q_1), \ldots, (q, n, q_n)\}$,
- $q = (\underline{cs}, h\ u_1\ \ldots\ u_n)$,
- $h \in \underline{\mathsf{Cons}_\mathsf{D}} \cup \mathsf{V}_\mathsf{L}$,
- $q_i = (\text{truncate}(\underline{cs}, u_i), u_i)$ und
- $\neg\exists \delta \in \mathsf{SUB}(\mathsf{D}, \mathsf{V}), q'' \in \mathsf{Q}$: $(q'', \delta, q) \in \text{case}$

gilt. Klassenbedingungen werden wie bei der Auswertung durch die Funktion truncate reduziert. Die Bedingung $\neg\exists \delta \in \mathsf{SUB}(\mathsf{D}, \mathsf{V}), q'' \in \mathsf{Q}$: $(q'', \delta, q) \in \text{case}$ garantiert, daß die Parameteraufteilung nicht auf Knoten angewendet werden kann, die direkt durch eine Fallunterscheidung entstanden sind. Mit dieser Bedingung wird erreicht, daß in jedem Narrowing-Baum des Narrowing-Graphen nach einer Fallunterscheidung mindestens eine Auswertung angwendet wird. Diese Eigenschaft der Narrowing-Graphen wird später in der Terminierungsanalyse (siehe Kapitel 5) benötigt, um sinnvolle Regeln für Dependency-Pair-Probleme ablesen zu können.

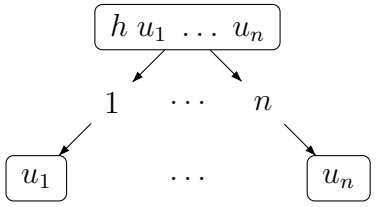

Abbildung 4.4: Parameteraufteilung

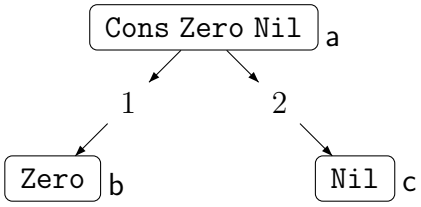

Abbildung 4.5: Beispiel einer Parameteraufteilung

Abbildung 4.4 zeigt den allgemeinen Teilgraph, der ab Knoten $(\underline{cs}, h\ u_1\ \ldots\ u_n)$ bei einer Parameteraufteilung entsteht. Ein Beispiel für eine Parameteraufteilung des Knotens $(\emptyset, \texttt{Cons Zero Nil})$ ist in Abbildung 4.5 gegeben. Dabei sind die Terme der Nachfolgerknoten jeweils die Parameter `Zero` und `Nil` des Konstruktors `Cons`. Außerdem sind die beiden Teiltermkanten $(\mathsf{a}, 1, \mathsf{b})$ und $(\mathsf{a}, 2, \mathsf{c})$ zu subterm hinzugekommen.

Lemma 4.2 (Korrektheit der Parameteraufteilung)

$$(h \in \underline{\mathsf{Cons}}_\mathsf{D} \cup \mathsf{V}_\mathsf{L},\ h\sigma, u_1\sigma, \ldots, u_n\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}) \Rightarrow (h\ u_1\ \ldots\ u_n)\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$$

Beweis:
Seien $h \in \underline{\mathsf{Cons}}_\mathsf{D} \cup \mathsf{V}_\mathsf{L}$, $h\sigma, u_1\sigma, \ldots, u_n\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *und sei* σ *eine typkorrekte Substitution. Wegen* $h\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *folgt, daß* $h\sigma\ t_1 \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *für alle typkorrekten* $t_1 \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$. *Daraus folgt nun wiederum, daß* $h\sigma\ t_1\ t_2 \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *für alle typkorrekten* $t_1, t_2 \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *usw. bis* $h\sigma\ t_1\ \ldots\ t_n \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *für alle typkorrekten* $t_1, \ldots, t_n \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$. *Wegen* $u_1\sigma, \ldots, u_n\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *gilt so* $h\sigma\ u_1\sigma\ \ldots\ u_n\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$ *oder äquivalent dazu* $(h\ u_1\ \ldots\ u_n)\sigma \in \mathsf{NF}^\mathsf{G}_\mathsf{HP}$. □
.

4.1.3 Fallunterscheidung einer Variablen

Sobald der Redex eines Terms $C\big[x_{|d\ \tau_1\ \ldots\ \tau_m|}\big]$ die Variable $x_{|d\ \tau_1\ \ldots\ \tau_m|} \in \mathsf{V}_\mathsf{L}$ ist, kann die Auswertungsfunktion keinen weiteren Schritt gehen, weil dafür der konkrete Wert von $x_{|d\ \tau_1\ \ldots\ \tau_m|}$ nötig ist. Für die Terminierungsanalyse stellt das kein Hindernis dar, denn die NF-Terminierung von $C\big[x_{|d\ \tau_1\ \ldots\ \tau_m|}\big]$ ist gegeben, wenn für jede $\mathsf{NF}^\mathsf{G}_\mathsf{HP}$-Substitution σ der Term $C\big[x_{|d\ \tau_1\ \ldots\ \tau_m|}\big]\sigma$ NF-terminiert, wenn dieser typkorrekt ist. Da hier die $\mathsf{NF}^\mathsf{G}_\mathsf{HP}$-Substitution σ typkorrekt ist, kann diese für die Variable $x_{|d\ \tau_1\ \ldots\ \tau_m|}$ nur NF-terminierende Terme vom Typ $d\ \tau_1\ \ldots\ \tau_m$ einsetzen. NF-terminierende Terme haben genau eine Normalform und wegen $d \in \underline{\mathsf{TyCons}}_\mathsf{D} \setminus \{\rightarrow\}$ haben diese Normalformen immer einen der Konstruktoren des Typkonstruktors d als Kopf. Also ist es eine äquivalente Forderung, daß die Terme

$$C\big[x_{|d\ \tau_1\ \ldots\ \tau_m|}\big]\delta_1, \ldots, C\big[x_{|d\ \tau_1\ \ldots\ \tau_m|}\big]\delta_n$$

für $\delta_i = [x_{|d\ \tau_1\ \ldots\ \tau_m|}/(c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|}]$ NF-terminieren, wenn $c_1, \ldots, c_n$ die Konstruktoren von d sind, $n_i = arity(c_i)$ ist und $x_{i,j} \in \mathsf{V_L}$ frische Variablen sind. Für Variablen vom Funktionstyp oder dem allgemeinen Typ a gibt es keine Patterns, so daß für diese nie eine Fallunterscheidung benötigt wird, da sie eine Auswertung nicht beim Patternmatching blockieren können.
Konkret läßt sich die Fallunterscheidung einer Variablen so definieren:

$$\frac{\langle \mathsf{Q} \uplus \{q\}, \mathsf{M}[q \mapsto \bot], \text{case}, \ldots \rangle}{\langle \mathsf{Q} \uplus \{q, q_1, \ldots, q_n\}, \mathsf{M}[q \mapsto \underline{\text{case}}], \text{case} \uplus \{(q, \sigma_1, q_1), \ldots, (q, \sigma_n, q_n)\}, \ldots \rangle}$$

wobei folgende Bedingungen erfüllt sein müssen:

- $q = (\underline{cs}, u)$,
- $u = v\ u_1\ \ldots\ u_n$,
- $v \in \mathsf{V_F}$,
- $u_1, \ldots, u_n \in \mathsf{H_B(D, V)}$,
- $\underline{\text{error}}\ x_{|d\ \tau_1\ \ldots\ \tau_m|} = \text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u)$,
- $x_{|d\ \tau_1\ \ldots\ \tau_m|} \in \mathsf{V_L}$,
- $d \in \underline{\mathsf{TyCons_D}} \setminus \{\rightarrow\}$,
- $m = \text{arity}(d)$,
- $\{c_1, \ldots, c_n\} = \text{constr}_{\mathsf{D}}(d)$,
- $n_i = \text{arity}(c_i)$,
- $x_{i,j} \in \mathsf{V_L}$ frische Variablen,
- $\delta_i = [x_{|d\ \tau_1\ \ldots\ \tau_m|}/(c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|}]$ und
- $q_i = (\underline{cs}, u\delta_i)$

Die Fallunterscheidung für Variablen darf nur auf solche Knoten angewendet werden, deren Terme mit einer Funktionsvariablen beginnen, denn sonst wäre die Parameteraufteilung anwendbar. Durch $\underline{\text{error}}\ x_{|d\ \tau_1\ \ldots\ \tau_m|} = \text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u)$ wird gerade erklärt, daß die Variable $x_{|d\ \tau_1\ \ldots\ \tau_m|}$ innerhalb des Terms u ausgewertet werden muß, falls dieser selbst ausgewertet wird.

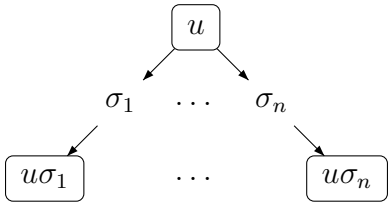

Abbildung 4.6: Fallunterscheidung einer Variablen

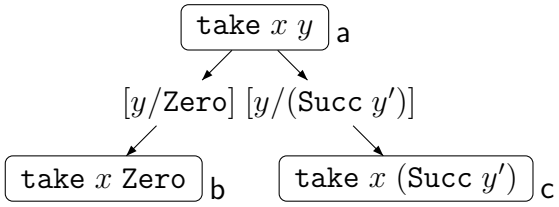

Abbildung 4.7: Beispiel einer Fallunterscheidung einer Variablen

In Abbildung 4.6 ist der allgemeine Teilgraph zu sehen, der durch die Fallunterscheidung ab Knoten $(\underline{cs}, u)$ entsteht. Ein Beispiel einer Fallunterscheidung für die Variable y des Knotens $(\emptyset, \texttt{take}\ x\ y)$ ist in Abbildung 4.1.3 zu sehen, wobei der Term zu dem Haskellprogramm aus Beispiel 4.1 gehört. Dabei wird die Variable y jeweils durch die Terme `Zero` und $(\texttt{Succ}\ y')$ ersetzt, denn diese decken alle Konstruktoren ab, die der Typ `Nat` besitzt. Es entstehen die beiden Nachfolgerknoten $(\emptyset, \texttt{take}\ x\ \texttt{Zero})$ und $(\emptyset, \texttt{take}\ x\ (\texttt{Succ}\ y'))$, zu denen jeweils die Fallunterscheidungskanten $(a, [y/\texttt{Zero}], b) \in \text{case}$ und $(a, [y/(\texttt{Succ}\ y')], c) \in \text{case}$ gezogen werden.

Lemma 4.3 (Korrektheit der Fallunterscheidung einer Variablen)
$u\delta_1, \ldots, u\delta_n \in \mathsf{NF}_{\mathsf{HP}} \Rightarrow u \in \mathsf{NF}_{\mathsf{HP}}$

Beweis:
Sei $u\delta_1, \ldots, u\delta_n \in \mathsf{NF}_{\mathsf{HP}}$. *Annahme, es ist* $u \notin \mathsf{NF}_{\mathsf{HP}}$, *so gibt es eine* $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-*Substitution* σ *mit* $x_{|d\,\tau_1\,\ldots\,\tau_m|}\sigma = t_{|d\,\tau_1\,\ldots\,\tau_m|}$ *für die* $u\sigma$ *typkorrekt ist und eine unendliche Auswertung startet. Wegen* $t_{|d\,\tau_1\,\ldots\,\tau_m|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *existiert die Normalform* $t_{|d\,\tau_1\,\ldots\,\tau_m|}{\downarrow_{\mathsf{HP}}}$. *Durch* $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u) = \underline{\text{error}}\ x_{|d\,\tau_1\,\ldots\,\tau_m|}$ *wird ausgedrückt, daß die Variable*
$x_{|d\,\tau_1\,\ldots\,\tau_m|}$ *der nächste Redex von* u *währe, aber diese nicht ausgewertet werden konnte. Deswegen ist* $t_{|d\,\tau_1\,\ldots\,\tau_m|}$ *der Redex von* $u\sigma$.
Wenn nun $t_{|d\,\tau_1\,\ldots\,\tau_m|}{\downarrow_{\mathsf{HP}}} = \underline{\text{error}}$ *gilt, würde* $u\sigma$ *unweigerlich zu* $\underline{\text{error}}$ *ausgewertet werden und würde so terminieren. Also ist* $t_{|d\,\tau_1\,\ldots\,\tau_m|}{\downarrow_{\mathsf{HP}}} \neq \underline{\text{error}}$ *und es existiert eine* $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-*Substitution* μ *mit* $(c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\,\tau_1\,\ldots\,\tau_m|}\mu = t_{|d\,\tau_1\,\ldots\,\tau_m|}{\downarrow_{\mathsf{HP}}}$, *weil der*

Typkonstruktor d mindestens den Konstruktor c_i besitzt, da $d \in \underline{\mathsf{TyCons}_\mathsf{D}} \backslash \{\rightarrow\}$ ist. Also gibt es den ersten Term $t'_{|d\ \tau_1\ \ldots\ \tau_m|}$ in der Auswertungsfolge von $t_{|d\ \tau_1\ \ldots\ \tau_m|}$, für den ebenfalls μ' mit $(c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|}\mu' = t'_{|d\ \tau_1\ \ldots\ \tau_m|}$ existiert. Der Teilterm $t_{|d\ \tau_1\ \ldots\ \tau_m|}$ innerhalb von $u\sigma$ muß mindestens bis zu $t'_{|d\ \tau_1\ \ldots\ \tau_m|}$ ausgewertet werden, damit entschieden werden kann, welche Regel als nächste genommen werden muß, ansonsten würde $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u) = \underline{\text{error}}\ x_{|d\ \tau_1\ \ldots\ \tau_m|}$ *nicht gelten. Also wird $u\sigma$ in einigen Schritten zu*

$$\begin{aligned}
&u[x_{|d\ \tau_1\ \ldots\ \tau_m|}/t'_{|d\ \tau_1\ \ldots\ \tau_m|}]\sigma \\
&= u[x_{|d\ \tau_1\ \ldots\ \tau_m|}/((c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|}\mu)]\sigma \\
&= u[x_{|d\ \tau_1\ \ldots\ \tau_m|}/((c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|})]\mu\sigma \\
&= u\delta_i\mu\sigma
\end{aligned}$$

ausgewertet. Da $u\sigma$ eine unendliche Auswertung startet, muß der Term $u\delta_i\mu\sigma$ diese fortführen. Das widerspricht der Annahme $u\delta_i \in \mathsf{NF}_{\mathsf{HP}}$ und so folgt, daß $u\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ oder äquivalent dazu $u \in \mathsf{NF}_{\mathsf{HP}}$. □

4.1.4 Fallunterscheidung einer Typvariablen

Ähnlich wie in der Fallunterscheidung einer Variablen blockiert die Auswertungsfunktion eines Terms $t \in \mathsf{H}_\mathsf{B}(\mathsf{D},\mathsf{V})$ mit dem Redex $v_{|\rho|}\ t_1\ \ldots\ t_m$ an Stelle π für eine Instanz $v_{|\rho|} \in \mathsf{V}_\mathsf{F}$ der Member-Variablen $v_{|\tau|}$, wenn die Typannotation ρ nicht konkret genug ist, um eine in dem Haskellprogramm angegebenen Instanz $v_{|\tau\mu_i|}$ der Member-Variablen $v_{|\tau|}$ zu wählen. Der Term t NF-terminiert laut Definition 4.5 nur, wenn jede typkorrekte Konkretisierung $t\sigma$ durch die $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-Substitution σ NF-terminiert. Es ist dabei zu beachten, daß hier σ auch auf Typannotationen wirkt. Das heißt, die nun genügend konkrete Instanz $v_{|\rho\sigma|}$ der Member-Variablen $v_{|\tau|}$ innerhalb des Terms $t\sigma$ muß spezieller sein als eine der im Haskellprogramm angegebenen Instanzen $v_{|\tau\mu_1|}, \ldots, v_{|\tau\mu_n|}$ der Member-Variablen $v_{|\tau|}$, wobei $\mu_1, \ldots, \mu_n$ Substitutionen auf Typvariablen sind. Es ist daher eine äquivalente Forderung, daß die Terme

$$t[v_{|\rho\delta_1|}\ t_1\ \ldots\ t_m]_\pi, \ldots, t[v_{|\rho\delta_n|}\ t_1\ \ldots\ t_m]_\pi$$

NF-terminieren, wenn δ_i der MGU für die Typen $\tau\mu_i$ und ρ für jedes i ist. Die Fallunterscheidung einer Typvariablen erzeugt nun die Knoten für alle Terme der Form $t[v_{|\rho\delta_i|}\ t_1\ \ldots\ t_m]_\pi$ und ist definiert wie folgt:

$$\frac{\langle \mathsf{Q} \uplus \{q\}, \mathsf{M}[q \mapsto \bot], \text{case}, \ldots \rangle}{\langle \mathsf{Q} \uplus \{q, q_1, \ldots, q_n\}, \mathsf{M}[q \mapsto \underline{\text{case}}], \text{case} \uplus \{(q, \sigma_1, q_1), \ldots, (q, \sigma_n, q_n)\}, \ldots \rangle}$$

wobei die folgenden Bedingung erfüllt sein müssen:

- $q = (\underline{cs}, u)$,
- $u = w\ u_1\ \ldots\ u_n$,
- $w \in \mathsf{V_F}$,
- $u_1, \ldots, u_n \in \mathsf{H_B}(\mathsf{D}, \mathsf{V})$,
- $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u) = \underline{\text{error}}\ v$,
- $v \in \mathsf{V_F}$,
- $\{\delta_1, \ldots, \delta_n\} = \text{filter}_{\mathsf{HP}}(\text{instances}_{\mathsf{HP}}(v), \underline{cs})$ und
- $q_i = (\text{reduce}_{\mathsf{HP}}(\underline{cs}\delta_i), u\delta_i)$

Die Fallunterscheidung für Typvariablen darf nur auf solche Knoten angewendet werden, deren Terme mit einer Funktionsvariablen beginnen, denn sonst wäre die Parameteraufteilung anwendbar. Durch $\underline{\text{error}}\ v = \text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u)$ wird gefordert, daß der Typ der Member-Variablen v nicht ausreicht, um eine im Haskellprogramm gegebene Instanz zu wählen, damit diese Fallunterscheidung nur zur Anwendung kommt, wenn sie wirklich benötigt wird.

Die verschiedenen Instanzen werden mit der Funktion

$$\text{instances}_{\mathsf{HP}} :: \mathsf{V_F} \longrightarrow \mathsf{SUBS}(\mathsf{D}, \mathsf{V})$$

erzeugt, welche folgendermaßen definiert ist:

$$\begin{aligned}\text{instances}_{\mathsf{HP}}(v_{|\rho|}) := \{\delta \in \mathsf{SUBS}(\mathsf{D}, \mathsf{V}) \mid\ & (\underline{cs}, c\ a, \underline{fs}) \in \mathsf{C_{HP}},\\ & (v_{|\tau|}, s, \underline{rs}) \in \underline{fs},\\ & (\underline{cs}', c\ \tau', \underline{fs}') \in \mathsf{I_{HP}},\\ & \delta \text{ ist MGU von } \tau[a/\tau'] \text{ und } \rho\}\end{aligned}$$

Die Klasse $(\underline{cs}, c\ a, \underline{fs})$ der Member-Variablen $v_{|\rho|}$ zeichnet sich dadurch aus, daß in ihrer Funktionliste $\underline{fs}$ die Funktion $(v_{|\tau|}, s, \underline{rs})$ existiert, an die die Member-Variable $v_{|\tau|}$ gebunden ist. Für alle Instanzen $(\underline{cs}', c\ \tau', \underline{fs}')$ wird nun geprüft, ob deren Member-Variablen $v_{|\tau[a/\tau']|}$ auf $v_{|\rho|}$ passen, also ob der MGU δ von $\tau[a/\tau']$ und ρ existiert. Für den Fall, daß δ existiert, ist $v_{|\rho\delta|}$ spezieller als $v_{|\tau[a/\tau']|}$. Das heißt, δ ist die gesuchte Substitution, um den Typ ρ von $v_{|\rho|}$ so zu konkretisieren, daß die Auswertungsfunktion nur die an $v_{|\tau[a/\tau']|}$ gebundene Funktion aus der Funktionsliste $\underline{fs}'$ der Instanz $(\underline{cs}', c\ \tau', \underline{fs}')$ wählen kann und damit wieder eindeutig wird.

Möglicherweise erfüllen einige Instanzen nicht die Klassenbedingungen $\underline{cs}$ des ursprünglichen Knotens q und müssen daher ausgelassen werden. Die Funktion $\text{filter}_{\mathsf{HP}} :: \text{Pot}(\mathsf{SUBS}(\mathsf{D},\mathsf{V})) \times \mathsf{CC}(\mathsf{D},\mathsf{V}) \longrightarrow \mathsf{SUBS}(\mathsf{D},\mathsf{V})$ übernimmt diese Aufgabe folgendermaßen:

$$\text{filter}_{\mathsf{HP}}(M, \underline{cs}) := \{\delta \in M \mid \forall (c\ \tau) \in \text{reduce}_{\mathsf{HP}}(\underline{cs}\delta)\text{: } (\tau = h\ \tau_1\ \ldots\ \tau_n \wedge h \in \mathsf{V_T})\}$$

Jetzt wird erst die Funktion $\text{reduce}_{\mathsf{HP}}$ eingeführt und anschließend wird eine Erklärung der Bedingung $\forall (c\ \tau) \in \text{reduce}_{\mathsf{HP}}(\underline{cs}\delta)\text{: } (\tau = h\ \tau_1\ \ldots\ \tau_n \wedge h \in \mathsf{V_T})$ aus der Funktion $\text{filter}_{\mathsf{HP}}$ gegeben.

Definition 4.12 (Klassenbedingungsreduktion) *Die Funktion* $\text{reduceStep}_{\mathsf{I}} :: \text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \longrightarrow \text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V}))$ *eliminiert eine durch eine Instanz aus der Menge* $\mathsf{I} \subseteq \mathsf{I}(\mathsf{D},\mathsf{V})$ *von Instanzen erfüllbare Klassenbedingung und gibt die restlichen Klassenbedingungen zurück.*

$$\text{reduceStep}_{\mathsf{I}}(\{c\ \tau\} \uplus \underline{cs}) := \{c_1\ a_{i_1}\sigma, \ldots, c_n\ a_{i_n}\sigma\} \uplus \underline{cs}$$

wenn

- $(\{c_1\ a_{i_1}, \ldots, c_n\ a_{i_n}\}, c\ (c'\ a_1\ \ldots\ a_m), \underline{fs}) \in \mathsf{I}$
- σ *ist Matcher von* $c'\ a_1\ \ldots\ a_m$ *und* τ

Es wird geprüft, ob die Klassenbedingung $c\ \tau$ *einer gegeben Instanz*

$$(\{c_1\ a_{i_1}, \ldots, c_n\ a_{i_n}\}, c\ (c'\ a_1\ \ldots\ a_m), \underline{fs})$$

aus I *entspricht. In diesem Fall wird sie durch die mit dem Matcher* σ *verfeinerten Klassenbedingungen* $c_1\ a_{i_1}\sigma, \ldots, c_n\ a_{i_n}\sigma$ *ersetzt.*

Mit $\text{reduce}_{\mathsf{I}}(\underline{cs})$ *wird die Normalform der Menge* $\underline{cs}$ *bezüglich der Funktion* $\text{reduceStep}_{\mathsf{I}}$ *bezeichnet. Diese Normalform ist eindeutig, weil die Instanzen einer Klasse in einem Haskellprogramm sich nicht überlappen dürfen, wie im Haskell-98-Report [J+98] auf Seite 47 verlangt wird. So ergeben die einzelnen Instanzen ein terminierendes und konfluentes abstraktes Ersetzungssystem auf Klassenbedingungen. An dieser Stelle sei auf Typing Haskell in Haskell [Jon99] verwiesen, worin eine tiefergehende Erklärung des Haskell-Typsystems zu finden ist. In dieser Arbeit wird darauf nicht weiter eingegangen, da sonst ihr Rahmen gesprengt wird.*

Zusätzlich ist die Funktion $\text{reduce}_{\mathsf{HP}} :: \text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V})) \longrightarrow \text{Pot}(\mathsf{CC}(\mathsf{D},\mathsf{V}))$ *für das Haskellprogramm* HP*, welches die Instanzen aus der Menge* I *besitzt, wie folgt definiert:*

$$\text{reduce}_{\mathsf{HP}}(\underline{cs}) := \text{reduce}_{\mathsf{I}}(\underline{cs})$$

Zum Beispiel wird die Menge

$$\{\texttt{Equal}\ (\texttt{List}\ a), \texttt{Equal}\ \texttt{Nat}, \texttt{Equal}\ \texttt{b}\}$$

von Klassenbedingungen durch die Instanzen aus Beispiel 2.1 zu der Normalform $\{\texttt{Equal}\ a, \texttt{Equal}\ b\}$ reduziert. Dabei werden folgende Reduktionsschritte vorgenommen:

$$\begin{aligned}&\text{reduceStep}_{\mathsf{I}}(\{\texttt{Equal}\ (\texttt{List}\ a), \texttt{Equal}\ \texttt{Nat}, \texttt{Equal}\ b\})\\ &= \{\texttt{Equal}\ a, \texttt{Equal}\ \texttt{Nat}, \texttt{Equal}\ b\}\\ &\text{reduceStep}_{\mathsf{I}}(\{\texttt{Equal}\ a, \texttt{Equal}\ \texttt{Nat}, \texttt{Equal}\ b\})\\ &= \{\texttt{Equal}\ a, \texttt{Equal}\ b\}\end{aligned}$$

Dabei werden die beiden Schritte durch die folgenden Instanzen ermöglicht:

- $(\{\texttt{Equal}\ a\}, \texttt{Equal}\ (\texttt{List}\ a), \underline{fs}_2) \in \mathsf{I}$
- $(\emptyset, \texttt{Equal}\ \texttt{Nat}, \underline{fs}_1) \in \mathsf{I}$

Die Bedingung $\forall(c\ \tau) \in \text{reduce}_{\mathsf{HP}}(\underline{cs}\delta)$: $(\tau = h\ \tau_1\ \ldots\ \tau_n \wedge h \in \mathsf{V_T})$ der Funktion $\text{filter}_{\mathsf{HP}}$ sagt aus, daß der Typ τ jeder Klassenbedingung $(c\ \tau)$ aus der Normalform $\text{reduce}_{\mathsf{HP}}(\underline{cs}\delta)$ als Kopf eine Typvariable enthält. Wenn es stattdessen ein Typkonstruktor ist, konnte bei der Reduktion der Menge $\underline{cs}\delta$ von Klassenbedingungen mindestens eine Klassenbedingung nicht durch eine Instanz aus dem Haskellprogramm eliminiert werden. Eine Instanz einer Klasse für diesen Typkonstruktor existiert also nicht, so daß δ die Typvariable so instantiiert haben muß, daß diese Klassenbedingung nicht erfüllt werden kann, auch dann nicht, wenn weitere Verfeinerungen auf $\underline{cs}\delta$ angewendet werden.

Sei zum Beispiel $\underline{cs} = \{\texttt{Equal}\ a, \texttt{A}\ a\}$, wobei die Klasse `Equal` wie im Haskellprogramm aus Beispiel 2.1 definiert ist und die andere Klasse `A` keine Instanzen besitzt. Für $\delta = [a/(\texttt{List}\ a)]$ ergibt sich so diese Menge

$$\underline{cs}\delta = \{\texttt{Equal}\ (\texttt{List}\ a), \texttt{A}\ (\texttt{List}\ a)\}$$

von Klassenbedingungen. Die Menge $\underline{cs}\delta$ hat die folgende Normalform:

$$\text{reduce}_{\mathsf{HP}}(\{\texttt{Equal}\ (\texttt{List}\ a), \texttt{A}\ (\texttt{List}\ a)\}) = \{\texttt{Equal}\ a), \texttt{A}\ (\texttt{List}\ a)\}$$

Diese Normalform wird nach einem Reduktionsschritt durch die Instanz

$$(\{\texttt{Equal}\ a\}, \texttt{Equal}\ (\texttt{List}\ a), \underline{fs}_2) \in \mathsf{I}$$

erreicht, weil für Klasse `A` keine Instanzen für Reduktionen bereit stehen. Würde es eine Instanz der Klasse `A` für den Typ `List a` geben, müßte diese hier anwendbar sein und die Klassenbedingung `A` (`List` a) könnte reduziert werden. Denn die ersten beiden Bedingungen für Instanzen der Form

$$(\{c_1\ a_{i_1}, \ldots, c_n\ a_{i_n}\}, c\ (c'\ a_1\ \ldots\ a_m), \underline{fs}) \in \mathsf{I}(\mathsf{D}, \mathsf{V})$$

sind $a_1, \ldots, a_m \in \mathsf{V_T}$ und $i_1, \ldots, i_n \in \{1, \ldots, m\}$, wie es bereits in der Definition 2.21 der Instanzen auf Seite 22 gefordert wird. Diese Forderung ist ebenfalls im Haskell-98-Report [J+98] auf Seite 46 zu finden.

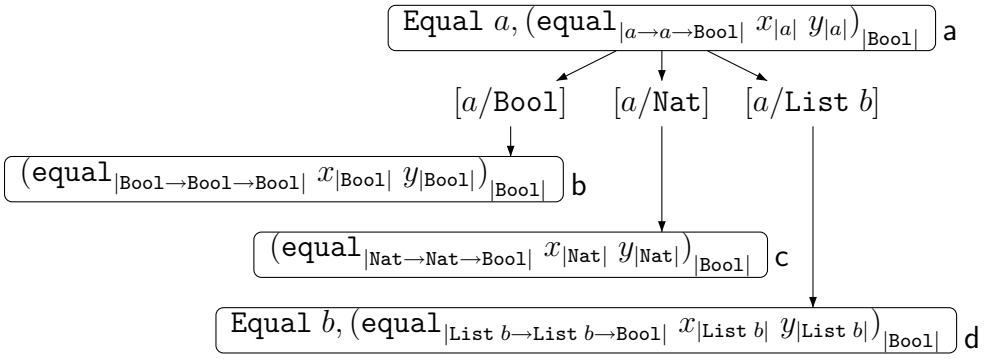

Abbildung 4.8: Beispiel einer Fallunterscheidung einer Typvariablen

In der Abbildung 4.8 ist eine Fallunterscheidung für die Typvariable a des Knotens (`Equal` a, (`equal`$_{|a\to a\to \texttt{Bool}|}$ $x_{|a|}$ $y_{|a|}$)$_{|\texttt{Bool}|}$) gezeigt, wobei der Term zu dem Haskellprogramm aus Beispiel 2.1 gehört. Die Typvariable a wurde jeweils durch die Typen `Bool`, `List` b und `Nat` instantiiert. Die jeweiligen Klassenbedingungen `Equal Bool`, `Equal` (`List` b) und `Equal Nat` werden durch die im Haskellprogramm in Beispiel 2.1 angegebenen Instanzen eliminiert, so daß alle Mengen der Klassenbedingungen der Nachfolgerknoten leer sind. Außerdem sind die drei Fallunterscheidungskanten $(\mathsf{a}, [a/\texttt{Bool}], \mathsf{b}) \in$ case, $(\mathsf{a}, [a/\texttt{Nat}], \mathsf{c}) \in$ case und $(\mathsf{a}, [a/\texttt{List}\ b], \mathsf{d}) \in$ case entstanden.

Lemma 4.4 (Korrektheit der Fallunterscheidung einer Typvariablen)
$u\delta_1, \ldots, u\delta_n \in \mathsf{NF_{HP}} \Rightarrow u \in \mathsf{NF_{HP}}$

Beweis:
Sei $u\delta_1, \ldots, u\delta_n \in \mathsf{NF_{HP}}$ *und sei* σ *eine* $\mathsf{NF^G_{HP}}$*-Substitution, so gilt* $u\sigma \in \mathsf{H^G_B}(\mathsf{D}, \mathsf{V})$. *In dem Grundterm* $u\sigma$ *ist die Instanz der Member-Variablen* v *eindeutig und muß eine Spezialisierung von einem der Fälle* $v\delta_1, \ldots, v\delta_n$ *sein. Das heißt also, es gibt ein* $i \in \{1, \ldots, n\}$ *und eine Substitution* μ *mit* $u\delta_i\mu = u\sigma$. *So folgt aus* $u\delta_i \in \mathsf{NF_{HP}}$, *daß* $u\sigma \in \mathsf{NF^G_{HP}}$ *oder äquivalent dazu* $u \in \mathsf{NF_{HP}}$. □

4.1.5 Variablenexpansion

Die Variablenexpansion appliziert die frische lokale Variable $x_{|\tau|} \in \mathsf{V_L}$ an den Term $u_{|\tau\to\tau'|}$ eines Knotens, da aus der Definition 4.5 folgt, daß der Term $u_{|\tau\to\tau'|}$ genau dann NF-terminiert, wenn der Term $(u_{|\tau\to\tau'|}\ x_{|\tau|})_{|\tau'|}$ NF-terminiert.

$$\frac{\langle \mathsf{Q} \uplus \{q\}, \mathsf{M}[q \mapsto \bot], \ldots, \text{varexp}, \ldots \rangle}{\langle \mathsf{Q} \uplus \{q, q'\}, \mathsf{M}[q \mapsto \underline{\text{varexp}}], \ldots, \text{varexp}[q \mapsto q'], \ldots \rangle}$$

wobei

- $q = (\underline{cs}, u_{|\tau\to\tau'|})$,
- $x_{|\tau|} \in \mathsf{V_L}$ eine frische Variable ist,
- $q' = (\underline{cs}, (u_{|\tau\to\tau'|}\ x_{|\tau|})_{|\tau'|})$ und
- (u ist eine Normalform $\vee$ $\neg\exists \delta \in \mathsf{SUB}(\mathsf{D}, \mathsf{V}), q'' \in \mathsf{Q}$: $(q'', \delta, q) \in$ case)

gilt. Die Variablenexpansion kann nur auf Knoten angewendet werden, wenn diese nicht direkt durch eine Fallunterscheidung entstanden sind, außer deren Terme sind in Normalform, wie die letzte der oben gelisteten Bedingungen fordert. Es wird so erreicht, daß direkt auf eine Fallunterscheidung eine Auswertung folgt. Mit dieser Bedingung wird erreicht, daß in jedem Narrowing-Baum des Narrowing-Graphen nach einer Fallunterscheidung mindestens eine Auswertung angwendet wird. Diese Eigenschaft der Narrowing-Graphen wird später in der Terminierungsanalyse (siehe Kapitel 5) benötigt, um sinnvolle Regeln für Dependency-Pair-Probleme ablesen zu können. Die Ausnahme für die Normalform ist nötig, falls eine Fallunterscheidung für die Typvariable a auf den Knoten, der den Term $u_{|a|}$ vom Typ a hat, angewendet wird. Eine der Fallbetrachtungs-Substitutionen könnte von der Form $[a/(\tau \to \tau')]$ sein, so daß die zugehörige Konkretisierung von $u_{|a|}$ der Term $u_{|\tau\to\tau'|}$ wäre. Wenn dieser nun eine Normalform ist, kann keine Transformation außer der Variablenexpansion angewendet werden, wodurch vielleicht wieder Regeln anwendbar werden.

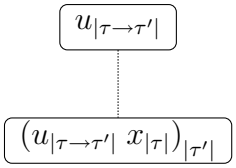

Abbildung 4.9: Variablenexpansion

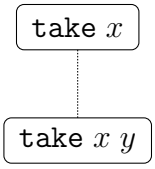

Abbildung 4.10: Beispiel einer Variablenexpansion

In Abbildung 4.9 wird der allgemeine Teilgraph ab Knoten $(\underline{cs}, u_{|\tau\to\tau'|})$ gezeigt, wie er bei einer Variablenexpansion entsteht. Ein Beispiel einer Variablenexpansion ist in Abbildung 4.10 für den Knoten $(\emptyset, \texttt{take}\ x)$ zu sehen, wobei sein Term zu dem Haskellprogramm aus Beispiel 4.1 gehört. Der Typ von $\texttt{take}\ x$ ist $\texttt{Nat} \to \texttt{List}\ a$ weil take vom Typ $\texttt{List}\ a \to \texttt{Nat} \to \texttt{List}\ a$ ist. Die Variablenexpansion appliziert nun die frische Variable y vom Typ Nat an den Term $\texttt{take}\ x$, so daß sich der Nachfolgerknoten mit dem Term $\texttt{take}\ x\ y$ vom Typ $\texttt{List}\ a$ ergibt.

Lemma 4.5 (Korrektheit der Variablenexpansion)
$(u_{|\tau\to\tau'|}\ x_{|\tau|})_{|\tau'|}\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}} \Rightarrow u_{|\tau\to\tau'|}\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$

Beweis:
Aus der Definition 4.5 folgt, daß die Normalform

$$(u_{|\tau\to\tau'|}\ x_{|\tau|})_{|\tau'|}\sigma{\downarrow_{\mathsf{HP}}}$$

existiert. So existiert zu dem Teilterm $u_{|\tau\to\tau'|}\sigma$ die Normalform $u_{|\tau\to\tau'|}\sigma{\downarrow_{\mathsf{HP}}}$, weil die Auswertungsrelation $\to_{\mathsf{HP}}$ den Redex soweit außen links wie möglich sucht. Zusammen mit $(u_{|\tau\to\tau'|}\ x_{|\tau|})_{|\tau'|}\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ folgt so $u_{|\tau\to\tau'|}\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$. □

4.1.6 Instantiierung

Die Instantiierung reduziert die NF-Terminierung eines Terms $u\mu$ auf die NF-Terminierung des generelleren Terms u. Die Substitution μ ist aber im allgemeinen Fall keine $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-Substitution und daher müssen die Terme im Bildbereich von μ auch auf NF-Terminierung überprüft werden. Die Instantiierung ist folgendermaßen definiert:

$$\frac{\langle \mathsf{Q} \uplus \{q, q'\}, \mathsf{M}[q \mapsto \bot], \ldots, \text{subterm}, \ldots, \text{generalisation}\rangle}{\langle \mathsf{Q} \uplus \{q, q', q_1, \ldots, q_n\}, \mathsf{M}[q \mapsto \underline{\text{instance}}], \ldots, \text{subterm}', \ldots, \text{generalisation}'\rangle}$$

wobei

- $\text{subterm}' = \text{subterm} \uplus \{(q, x_1, q_1), \ldots, (q, x_n, q_n)\}$,
- $\text{generalisation}' = \text{generalisation}[q \mapsto q']$,
- $q' = (\underline{cs}', u)$,
- $q = (\underline{cs}, u\mu)$,
- $\mu = [x_1/u_1, \ldots, x_n/u_n]$,
- $q_i = (\text{truncate}(\underline{cs}, u_i), u_i)$,
- $\mathsf{M}(q') \in \{\underline{\text{case}}, \underline{\text{eval}}\} \ \vee \ (u = x\ y \ \wedge \ x \ \in \mathsf{V_L} \ \wedge \ y \in \mathsf{V_L})$,
- $q \neq q'$ und
- $\neg \exists x \in \mathsf{V_L}, q'' \in \mathsf{Q}\text{: } (q'', x, q) \in \text{case}$

gilt. Die Instantiierung darf nur auf einen Knoten angewendet werden, wenn dieser nicht direkt durch eine Fallunterscheidung entstanden ist und die Generalisierungskante zu einem Knoten gezogen werden soll, auf dem entweder eine Fallunterscheidung oder eine Auswertung angwendet wird ($\mathsf{M}(q') \in \{\underline{\text{case}}, \underline{\text{eval}}\}$). Eine Generalisierungskante soll auch nicht wieder auf denselben Knoten zurückführen ($q \neq q'$). Später, in der Terminierungsanalyse (siehe Kapitel 5), werden Regeln ab Knoten, in die Generalisierungskanten zeigen, für Dependency-Pair-Probleme abgelesen. Damit diese Regeln echte Auswertungsschritte enthalten, werden diese Bedingungen gefordert. Ein Ausnahme ist eine Instantiierung mit dem Knoten des Terms $x\ y$, dessen Sonderstellung wird ebenfalls im Kapitel 6 erklärt.

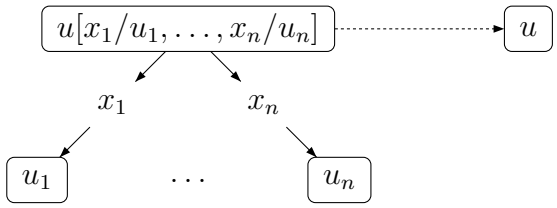

Abbildung 4.11: Instantiierung

Der allgemeinen Teilgraphen ab Knoten $(\emptyset, u[x_1/u_1, \ldots, x_n/u_n])$, wie er bei einer Instantiierung entsteht, zeigt die Abbildung 4.11. Ein Beispiel einer Instantiierung ist in Abbildung 4.11 für den Knoten

$$(\emptyset, \texttt{take}\ (\texttt{double}\ (\texttt{Succ}\ \texttt{Zero}))\ (\texttt{Cons}\ \texttt{Zero}\ \texttt{Nil}))$$

zu sehen, wobei sein Term zu dem Haskellprogramm aus Beispiel 4.1 gehört.

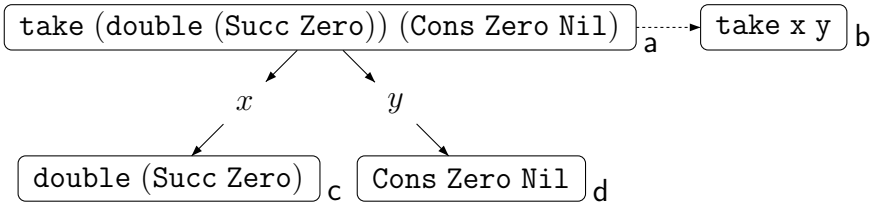

Abbildung 4.12: Beispiel einer Instantiierung

Der Term `take x y` des Knotens `b` matcht den des Knotens `a`, so daß eine Generalisierungskante von `a` nach `b` gezogen werden kann. Zu den Knoten `c` und `d` der Teilterme `double (Succ Zero)` und `Cons Zero Nil` werden die zwei Teiltermkanten $(\mathsf{a}, x, \mathsf{c}) \in$ subterm und $(\mathsf{a}, y, \mathsf{d}) \in$ subterm gezogen.

Lemma 4.6 (Korrektheit der Instantiierung) *Wenn σ eine $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-Substitution ist und $u \in \mathsf{NF}_{\mathsf{HP}}$, $u_1\sigma, \ldots, u_n\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ gilt, folgt $u\mu\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$.*

Beweis:
Sei σ eine $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-Substitution und $u \in \mathsf{NF}_{\mathsf{HP}}$ und seien $u_1\sigma, \ldots, u_n\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$. Wegen $u_1\sigma, \ldots, u_n\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ folgt, daß $\mu\sigma$ eine $\mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$-Substitution ist und damit auch $u\mu\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$, wegen $u \in \mathsf{NF}_{\mathsf{HP}}$. □

4.1.7 Erweiterung

Die Erweiterung des Narrowing-Graphen um einen beliebigen Knoten, stellt eine Generalisierung des ursprünglichen Problems dar und ist daher unkritisch. Sie dient dazu, neue Möglichkeiten für Instantiierungen zu erzeugen. Es gibt Startterme, wie wir im Kapitel 6 noch sehen werden, zu denen keine geschlossenen Narrowing-Graphen erzeugt werden können, wenn nicht die Instantiierung neue Ziele in Form von neuen Knoten geboten bekommt.

Ein neuer Knoten dient als Wurzel für einen neuen Narrowing-Graph, der in den aktuellen Narrowing-Graph eingebettet wird. Er erzeugt damit eine neue Menge von Knoten, zu denen durch Instantiierungen neue Generalisierungskanten gezogen werden können. Natürlich muß auch der neue Narrowing-Graph geschlossen werden, wenn der aktuelle geschlossen werden soll, da er in diesen eingebettet ist. Typischerweise wird eine Erweiterung um Knoten vorgenommen, deren Terme Generalisierungen von Termen sind, die bereits im Narrowing-Graph vorhandenen sind. Es wird also ein Term eines Knoten aus dem Narrowing-Graph genommen, in diesem werden einige Teilterme durch Variablen ersetzt und der neue Term wird in einen neuen Knoten gekapselt und in den Narrowing-Graph durch die Erweiterung eingebracht. Zu diesem neuen Knoten und seinen Nachfolgern

können jetzt von anderen Knoten aus Generalisierungskanten gezogen werden. Im Kapitel 6 wird das Kriterium der n-fachen Schachtelung vorgestellt, welches klärt, wie die Terme für eine Erweiterung durch eine Generalisierung von schon vorhandenen Termen gefunden werden können.

Die Erweiterung ist definiert durch:

$$\frac{\langle \mathsf{Q}, \mathsf{M}, \ldots \rangle}{\langle \mathsf{Q} \uplus \{q\}, \mathsf{M}, \ldots \rangle}$$

wobei $q \in \mathsf{CC}(\mathsf{D}, \mathsf{V}) \times \mathsf{H_B}(\mathsf{D}, \mathsf{V})$ gilt.

4.2 Eigenschaften von Narrowing-Graphen

In diesem Abschnitt werden grundlegende Eigenschaften der Narrowing-Graphen, auf welche wir später in der Terminierungsanalyse zurückgreifen werden, gefolgert.

Lemma 4.7 (Anwendungslemma) *Auf jeden Knoten in einem geschlossenen Narrowing-Graph* NG *wurde genau eine Transformation angewendet, wenn* NG *aus einer Startkonfiguration hervorgegangen ist.*

Beweis:
Sei der Narrowing-Graph

$$\mathsf{NG} := \langle \mathsf{Q} \uplus \{q\}, \mathsf{M}, \mathrm{case}, \mathrm{subterm}, \mathrm{eval}, \mathrm{varexp}, \mathrm{generalisation} \rangle$$

geschlossen und aus einer Startkonfiguration hervorgegangen. Annahme, auf Knoten q wäre keine Transformation angewendet worden, so folgt aus den Definitionen der Transformationen, daß $q \notin \mathrm{Dom}(\mathsf{M})$. Das ist aber ein Widerspruch zu der Geschlossenheit von NG *mit* $\mathrm{Dom}(\mathsf{M}) = Q$. □

Lemma 4.8 (Zykellemma) *Jeder Zyklus innerhalb eines Narrowing-Graphen* NG *enthält mindestens eine Generalisierungskante.*

Beweis:
Jede Narrowing-Graphtransformation außer der Instantiierung fügt neue Knoten hinzu und verbindet den Ausgangsknoten mit diesen neuen Knoten einmalig. Dadurch können also keine Zyklen entstehen. Nur die Instantiierung legt eine Generalisierungskante zu einem schon vorhandenen Knoten an. □

Lemma 4.9 (Generalisierungskantenlemma) *Eine unendliche Folge von aufeinander folgenden Knoten innerhalb eines geschlossenen Narrowing-Graphen erreicht mindestens eine Generalisierungskante.*

Beweis:
Die unendliche Folge muß in einen Zyklus laufen, weil der Narrowing-Graph endlich ist. Aus dem Zykellemma 4.8 folgt, daß in jedem Zyklus mindestens eine Generalisierungskante liegt. Da die unendliche Folge den Zyklus nicht verlassen kann, wird mindestens eine Generalisierungskante erreicht. □

Lemma 4.10 (Abschlußlemma) *Für jeden Narrowing-Graph in Startkonfiguration gibt es eine Transformationsfolge, die diesen schließt.*

Beweis:
Zu jeder Funktion $(v, \forall a_1, \ldots, a_n.\underline{cs} \Rightarrow \tau, \underline{rs}) \in \mathsf{F_{HP}}$ *eines Haskellprogramms* HP *wird eine Erweiterung um den Knoten* $(\underline{cs}, v\ x_1\ \ldots\ x_n)$ *angewendet, wobei alle* $x_i \in \mathsf{V_L}$ *frische Variablen sind. Zusätzlich wird um den Knoten* $(\emptyset, x\ y)$ *erweitert. Zu jedem dieser Knoten können mit endlicher Anzahl von Fallunterscheidungen und Auswertungen Pfade zu Knoten der rechten Seiten der Regeln aufgebaut werden. Für alle Knoten, auf die bis jetzt keine Transformation angewendet wurden(also inklusive des Startknotens), werden die nun folgenden Schritte ausgeführt. Wenn der Typ des Terms eines solchen Knotens einem Funktionstyp entspricht, wird so lange eine Variablenexpansion angewendet, bis dieser keinem Funktionstyp mehr entspricht. Der Konstruktorkontext eines solchen Knotens wird durch die Parameteraufteilung entfernt. Danach steht in jedem Fall ein Funktionsaufruf auf äußerster Position, welcher durch eine Instantiierung eliminiert wird. Das ist möglich, wenn die Funktion mit entsprechender Anzahl von Parametern ausgestattet ist, da für jede Funktion der Knoten* $(\underline{cs}, v\ x_1\ \ldots\ x_n)$ *existiert. Falls zu wenige Parameter existieren, kann durch eine Anzahl von Variablenexpansionen die passende Anzahl wieder erreicht werden. Wenn der Knoten zu viele Parameter hat, wird eine Instantiierung mit dem Knoten* $(\emptyset, x\ y)$ *vorgenommen und der letzte der überzähligen Parameter fällt weg.*

Für alle Teiltermnachfolger, die bei Instantiierung oder Parameteraufteilung entstehen, wiederholen wir diese Vorgehensweise wie beschrieben. Da das Haskellprogramm HP *endlich ist, gibt es zusammen mit dem Startterm nur endlich viele Teilterme der rechten Seiten von Regeln und für jeden dieser Terme wird entweder eine Parameteraufteilung, eine Variablenexpansion oder eine Instantiierung angewendet.* □

Kapitel 5

Terminierungsanalyse

Wie im Abschnitt 4.1 beschrieben, repräsentiert ein geschlossener Narrowing-Graph das Auswertungsverhalten eines Startterms. Sein innerer Aufbau garantiert, daß aus der NF-Terminierung der Nachfolger eines Knotens die NF-Terminierung dieses Knotens folgt. Wenn der Narrowing-Graph azyklisch ist, so ist jeder Pfad, ausgehend vom Startterm, endlich. Da auf jeden Knoten eine Transformation angewendet wurde, gilt für Knoten ohne Nachfolger, daß diese NF-terminieren und so NF-terminiert der Startterm ebenfalls. Ein Narrowing-Graph ist im allgemeinen nicht zykelfrei. Betrachten wir dazu den Zykel $\{\mathsf{a}, \mathsf{c}, \mathsf{f}, \mathsf{h}\}$ innerhalb des Narrowing-Graphen aus Abbildung 5.1 für das Haskellprogramm aus Beispiel 5.1 genauer.

Beispiel 5.1

```
data Nat = Succ Nat | Zero

minus (Succ x) (Succ y) = minus x y
minus Zero     y        = Zero
minus x        Zero     = x
```

Da der Startknoten $(\emptyset, \texttt{minus}\ m\ n)$ in diesem Zykel enthalten ist, wird die NF-Terminierung des Startterms auf seine eigene NF-Terminierung zurückgeführt. Diese Tautologie ist nicht sehr hilfreich. Für konkrete Instanzen eines Startterms zeigt sich ein anderes Bild von Zurückführungen der NF-Terminierung. Im Gegensatz zu der ursprünglichen Tautologie stellt sich für konkrete Instanzen des Startterms heraus, daß die NF-Terminierung einer Instanz von Knoten a auf die NF-Terminierung einer *anderen* Instanz von Knoten a zurückgeführt wird.

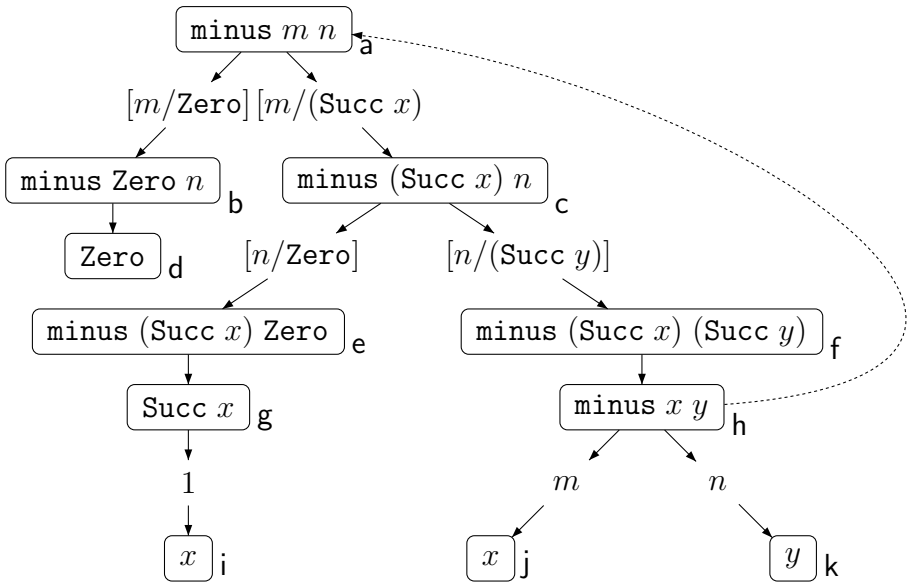

Abbildung 5.1: Ein zyklischer Narrowing-Graph zu Beispiel 5.1 für den Startterm `minus` m n

Beispielsweise ist für die konkrete Instanz `minus` (`Succ Zero`) (`Succ Zero`) des Startterms `minus` m n der Verlauf der Zurückführung in der Tabelle 5 zu sehen. Es zeigt sich, daß die Instanzen von Knoten a wirklich unterschiedlich sind, und

	Term	Substitution	Instanz
a	`minus` m n	$[m/(\texttt{Succ Zero}),$ $n/(\texttt{Succ Zero})]$	`minus` (`Succ Zero`) (`Succ Zero`)
c	`minus` (`Succ` x) n	$[x/\texttt{Zero}, n/(\texttt{Succ Zero})]$	`minus` (`Succ Zero`) (`Succ Zero`)
f	`minus` (`Succ` x) (`Succ` y)	$[x/\texttt{Zero}, y/\texttt{Zero}]$	`minus` (`Succ Zero`) (`Succ Zero`)
h	`minus` x y	$[x/\texttt{Zero}, y/\texttt{Zero}]$	`minus Zero Zero`
a	`minus` m n	$[m/\texttt{Zero}, n/\texttt{Zero}]$	`minus Zero Zero`
b	`minus Zero` n	$[n/\texttt{Zero}]$	`minus Zero Zero`
d	`Zero`	[]	`Zero`
j	`Zero`	$[x/\texttt{Zero}]$	`Zero`
k	`Zero`	$[y/\texttt{Zero}]$	`Zero`

Tabelle 5.1: Zurückführung der NF-Terminierung für die Instanz `minus` (`Succ Zero`) (`Succ Zero`)

zum Schluss wird sogar der Zykel verlassen. Beim Knoten h ist zu beachten, daß auch noch die beiden Knoten j und k weiter verfolgt werden müssen, da sich bei einer Instantiierung die NF-Terminierung nicht, wie bei der Fallunterscheidung für konkrete Instanzen, aus nur einem Nachfolger ableiten läßt. Es zeigt sich, daß von allen Instanzen des Startterms `minus` m n nach endlich vielen Zurückführungsschritten immer Knoten aus der Menge {d, i, j und k} erreicht werden. Wenn es

jetzt für einen Startterm gelingt, nachzuweisen, daß von jeder seiner Instanzen die Zurückführungspfade an Knoten ohne Nachfolger enden, ist die NF-Terminierung dieses Startterms gezeigt. Die Endlichkeitsnachweise von diesen Pfaden lassen sich gerade hier gut auf die Terminierung von Termersetzungssystemen zurückführen, da die Zurückführungsschritte der Pfade starke Ähnlichkeiten mit Termersetzungsschritten aufweisen. Außerdem gibt es für Termersetzungssysteme schon einige erfolgreiche Terminierungsanalyse-Tools und so können die Erfahrungen in diesem Bereich mit genutzt werden. Die unendlichen Zurückführungspfade für nicht NF-terminierende Startterme, die hier intuitiv erklärt wurden, werden später in der Definitionen 5.9 der $\overline{\mathsf{NF}}$-Nachfolgerfunktion und 5.10 der $\overline{\mathsf{NF}}$-Ketten im Abschnitt 5.3 formal gefaßt.

Heute ist es im allgemeinen üblich, daß Terminierungsanalyse-Tools für Termersetzungssysteme Derivate der Dependency-Pair-Methode, wie sie in [AG00] beschrieben worden ist, als Basistechniken verwenden. Ein wichtiger Schritt der Dependency-Pair-Methode ist die Erstellung von Dependency-Graphen, wie sie in Definition 15 von [AG00] beschrieben sind. Laut [GTSK05] werden aus diesen Graphen Dependency-Pair-Probleme zu jeder darin vorkommenden maximalen starken Zusammenhangskomponente abgelesen. Anschließend wird mit weiteren Methoden nachgewiesen, daß diese Dependency-Pair-Probleme endlich sind, woraus die Terminierung des ursprünglichen Termersetzungssystems gefolgert werden kann. Da diese Dependency-Graphen starke Ähnlichkeiten mit den Narrowing-Graphen aufweisen, werden wir hier direkt Dependency-Pair-Probleme aus den maximalen starken Zusammenhangskomponenten des Narrowing-Graphen ablesen und den Zwischenschritt über Termersetzungssysteme auslassen. Denn die Dependency-Pair-Probleme können auch unabhängig von einem Termersetzungssystem auf ihre Endlichkeit überpüft werden, wie es in [GTSK05] beschrieben ist. Diese Idee hat außerdem die Vorteile, daß Haskell-spezifische Strukturen, die innerhalb des Narrowing-Graphen beachtet worden sind, bis hinein in die Bildung der Dependency-Pairs wirken können. Zum Beispiel ist in den Aufbau der Narrowing-Graphen die exakte Haskellauswertungsstrategie eingeflossen, welche im Rahmen von Termersetzungssystemen nur sehr schwer zu simulieren ist, so daß sich ein erheblicher Vorteil des direkten Ablesens gegenüber dem Umweg über Termersetzungssysteme ergibt. Außerdem werden wir sehen, daß die Ablesestrategie für Regeln aus dem Narrowing-Graph nicht ohne weiteres auf den higher-order Fall erweiterbar ist, während die Ablesestrategie für Dependency-Pair-Probleme auch viele higher-order Fälle mit abdecken kann.

5.1 DP-Probleme

In diesem Abschnitt werden die Grundlagen für Termersetzungssysteme und Dependency-Pair-Probleme vorgestellt, um später das Ablesen genau beschreiben zu können.

Definition 5.1 (Term) *Ein Term t ist ein Element der Menge $\mathcal{T}(\Sigma, \mathcal{V})$, wobei*

- *Σ eine Menge von Symbolen und*
- *$\mathcal{V}$ eine Menge von Variablen ist.*

Zusätzlich hat jedes Symbol s aus der Menge Σ eine Stelligkeit $arity(s) \in \mathbb{N}$.
Die Menge $\mathcal{T}(\Sigma, \mathcal{V})$ ist die kleinste Menge, für die gilt:

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ *gdw.* $f \in \Sigma$, $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ *und* $arity(f) = n$

Folgende Bezeichnungen werden zusätzlich benutzt:

- $\mathcal{V}(t)$ *ist die Menge der Variablen, die im Term t enthalten sind.*
- $Occ(t)$ *ist die Menge der gültigen Stellen des Terms t und ist folgendermaßen definiert:*

  $$Occ(f(t_1, \ldots, t_n)) := \{i\pi \in \mathbb{N}^* \mid i \in \{1, \ldots, n\}, \pi \in Occ(t_i)\} \cup \{\epsilon\}$$
  $$Occ(x) := \{\epsilon\}$$

  wobei $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ und $x \in \mathcal{V}$ gilt.
- $t|_\pi$ *ist der Term an Stelle π innerhalb des Terms t*
- $t[r]_\pi$ *ist der Term t, in dem der Term an der Stelle π durch den Term r ersetzt wurde.*
- $s \trianglelefteq t$ *gilt gdw. s ein Teilterm von t ist:*

  $$s \trianglelefteq t \text{ gdw. } \exists \pi \in Occ(t)\colon t|_\pi = s$$
- $s \triangleleft t$ *gilt gdw.* $s \trianglelefteq t$ *und* $s \neq t$ *gilt*
- *Substitutionen werden meist σ oder μ genannt.*

Definition 5.2 (Regel) *Das Paar (l, r) ist eine Regel, wenn*

- $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$,
- $\mathcal{V}(l) \supseteq \mathcal{V}(r)$ *und*
- $l \notin \mathcal{V}$ *ist.*

Im folgenden wird die übliche Notation $l \to r$ benutzt.

Definition 5.3 (Termersetzungsschritt) *$s \to_{l\to r} t$ ist ein Termersetzungsschritt, wenn*

- $l \to r$ *eine Regel ist und*
- $s|_\pi = l\sigma$ *und dabei*
- $t = s[r\sigma]_\pi$ *gilt.*

Der Teilterm $l\sigma$ von s wird als Redex bezeichnet.

Definition 5.4 (Termersetzungssystem) *Ein Termersetzungssystem $\mathcal{R}$ ist eine endliche Menge von Regeln. Es induziert dabei die Termersetzungsrelation $\to_\mathcal{R}$. Für diese gilt:*

$$(s,t) \in \to_\mathcal{R} \text{ gdw. } s \to_{l\to r} t \text{ und } l \to r \in \mathcal{R}$$

Statt $(s,t) \in \to_\mathcal{R}$ wird die Notation $s \to_\mathcal{R} t$ benutzt. Die Stelle π, an der ein Ersetzungsschritt angewendet wird, kann durch die Notation $s \to_{\mathcal{R},\pi} t$ explizit angegeben werden. Die Menge der Termersetzungssysteme wird mit TES *bezeichnet.*

Die folgenden Definitionen der $(\mathcal{P}, \mathcal{R})$-Ketten und DP-Probleme sind Spezialisierungen der Definitionen 6 und 9 aus [GTSK05].

Definition 5.5 ($(\mathcal{P}, \mathcal{R})$-Kette) *Seien $\mathcal{P}$ und $\mathcal{R}$ Termersetzungssysteme und*

$$\begin{aligned} s_1 &\to t_1 \\ s_2 &\to t_2 \\ s_3 &\to t_3 \\ &\vdots \end{aligned}$$

ein Folge von Regeln aus $\mathcal{P}$, wobei o.B.d.A angenommen werden kann, daß zwei verschiedene Regeln dieser Folge keine gemeinsamen Variablen besitzen. Diese Folge ist eine $(\mathcal{P}, \mathcal{R})$-Kette gdw. es eine Substition σ gibt, so daß

$$t_i\sigma \to^*_\mathcal{R} s_{i+1}\sigma$$

für alle $i > 0$ gilt.

Nun werden die DP-Probleme eingeführt, welche aus den Narrowing-Graphen abgelesen werden sollen, wie es am Kapitelanfang bereits erwähnt wurde. Im weiteren Verlauf dieses Kapitels wird genau erklärt, wie diese DP-Probleme abgelesen werden.

Definition 5.6 (DP-Problem) *Ein Dependency-Pair-Problem (kurz: DP-Problem) ist ein Paar $(\mathcal{P}, \mathcal{R})$ aus den Termersetzungssystemen $\mathcal{P}$ und $\mathcal{R}$. Ein DP-Problem $(\mathcal{P}, \mathcal{R})$ ist endlich, wenn zu diesem keine unendliche $(\mathcal{P}, \mathcal{R})$-Kette existiert. Die Menge der DP-Probleme wird mit* DP *bezeichnet.*

Beispielsweise ist das DP-Problem $(\mathcal{P}_{\text{Toyama}}, \mathcal{R}_{\text{Toyama}})$ mit

$$\begin{aligned}\mathcal{P}_{\text{Toyama}} &:= \{f(0,1,x) \to f(x,x,x)\}\\ \mathcal{R}_{\text{Toyama}} &:= \{g(x,y) \to x,\\ &\qquad\ g(x,y) \to y\}\end{aligned}$$

nicht endlich, wie diese unendliche $(\mathcal{P}_{\text{Toyama}}, \mathcal{R}_{\text{Toyama}})$-Kette

$$\begin{aligned}f(0,1,x_1) &\to f(x_1,x_1,x_1)\\ f(0,1,x_2) &\to f(x_2,x_2,x_2)\\ f(0,1,x_3) &\to f(x_3,x_3,x_3)\\ &\vdots\end{aligned}$$

mit der Substitution $\sigma = [x_1/g(0,1), x_2/g(0,1), x_3/g(0,1), \ldots]$ belegt. Für die einzelnen Kettenglieder gilt hier:

$$\begin{aligned}f(x_i,x_i,x_i)\sigma &= f(g(0,1),g(0,1),g(0,1))\\ &\to_{\mathcal{R}} f(0,g(0,1),g(0,1))\\ &\to_{\mathcal{R}} f(0,1,g(0,1))\\ &= f(0,1,x_{i+1})\sigma\end{aligned}$$

Da DP-Probleme, wie oben definiert, aus den Termersetzungssystemen $\mathcal{P}$ und $\mathcal{R}$ bestehen und es laut [GTSK05] für die Endlichkeitsnachweise der DP-Probleme günstig ist, wenn die obersten Funktionssymbole einer linken Seite der Regeln aus $\mathcal{P}$ nicht innerhalb der Regeln aus $\mathcal{R}$ vorkommen, nutzen wir das später bei der Erzeugung der DP-Probleme durch Anwendung der $\sharp$-Transformation aus.

Definition 5.7 ($\sharp$-Transformation) *Die $\sharp$-Transformation benennt ein Funktionssymbol f an Stelle ϵ innerhalb eines Terms in ein zugehöriges neues Tupelsymbol $f^\sharp$ um. Sie ist definiert wie folgt:*

$$[f(t_1,\ldots,t_n)]^\sharp := f^\sharp(t_1,\ldots,t_n)$$

Zum Beispiel ergeben sich für die Terme

- $f(0, 1, x)$ und
- $f(0, 1, g(0, 1))$

nach Anwendung der $\sharp$-Transformation die folgenden Terme:

- $[\text{f(0,1,x)}]^\sharp = f^\sharp(0, 1, x)$
- $[\text{f(0,1,g(0,1))}]^\sharp = f^\sharp(0, 1, g(0, 1))$

Wie zu Kapitelanfang erklärt, sollen Termersetzungssysteme aus den Narrowing-Graphen abgelesen werden. Dabei müssen Haskell-Basisterme, die innerhalb der Narrowing-Graphen auftreten, als Terme von Termersetzungssystemen dargestellt werden. Es müssen also higher-order Terme aus Haskell auf first-order Terme der Termersetzungssysteme abgebildet werden. Diese Aufgabe übernimmt die APP-Transformation.

Definition 5.8 (APP-Transformation) *Die* APP*-Transformation überführt einen Haskell-Basisterm t aus $\mathsf{H_B}(\mathsf{D}, \mathsf{V})$ in den Term $[\![t]\!]_{\mathsf{APP}}$ aus*

$$\mathcal{T}(\{app\} \cup \{c_d \mid d \in \underline{\mathsf{Cons}}_\mathsf{D} \cup \mathsf{V_F}\}, \mathsf{V_L})$$

wobei $\forall d \in \underline{\mathsf{Cons}}_\mathsf{D} \cup \mathsf{V_F}$: $arity(c_d) = 0$ und $arity(app) = 2$ gilt. Sie ist definiert wie folgt:

$$\begin{aligned} [\![d]\!]_{\mathsf{APP}} &:= c_d \\ [\![x]\!]_{\mathsf{APP}} &:= x \\ [\![(t_1\ t_2)]\!]_{\mathsf{APP}} &:= app([\![t_1]\!]_{\mathsf{APP}}, [\![t_2]\!]_{\mathsf{APP}}) \end{aligned}$$

wobei

- $d \in \underline{\mathsf{Cons}}_\mathsf{D} \cup \mathsf{V_F}$,
- $x \in \mathsf{V_L}$ *und*
- $t_1, t_2 \in \mathsf{H_B}(\mathsf{D}, \mathsf{V})$

gilt. Für Terme der Form $app(x, y)$ wird auch häufig die Kurzschreibweise $x'y$ verwendet und statt c_d wird auch einfach nur d geschrieben.

Für die Terme

- `minus` (`Succ Zero`) y,
- `minus` (`Succ` x) und
- `minus`

ergibt eine Anwendung der APP-Transformation die folgenden Ergebnisse:

- $[\![\texttt{minus (Succ Zero)}\ y]\!]_{\mathsf{APP}} = app(app(c_{\texttt{minus}}, app(c_{\texttt{Succ}}, c_{\texttt{Zero}})), y)$,
- $[\![\texttt{minus (Succ}\ x)]\!]_{\mathsf{APP}} = app(c_{\texttt{minus}}, app(c_{\texttt{Succ}}, x))$ und
- $[\![\texttt{minus}]\!]_{\mathsf{APP}} = c_{\texttt{minus}}$.

wobei diese in Kurzschreibweise so aussehen:

- $[\![\texttt{minus (Succ Zero)}\ y]\!]_{\mathsf{APP}} = \texttt{minus}'(\texttt{Succ}'\texttt{Zero})'y$
- $[\![\texttt{minus (Succ}\ x)]\!]_{\mathsf{APP}} = \texttt{minus}'(\texttt{Succ}'x)$ und
- $[\![\texttt{minus}]\!]_{\mathsf{APP}} = \texttt{minus}$.

5.2 Ablesen von DP-Problemen

Es sollen Dependency-Pairs zu jeder maximalen starken Zusammenhangskomponente eines Narrowing-Graphen abgelesen werden. Wie zu Kapitelbeginn bereits erwähnt, soll dadurch der Zwischenschritt über Termersetzungssysteme übersprungen werden.

Um zu verdeutlichen, warum die Dependency-Pairs auf die im folgenden vorgestellte Weise abgelesen werden, wird hier erst einmal beschrieben, wie ein Termersetzungssystem aus dem Narrowing-Graph abgelesen werden könnte, welches, wenn es terminiert, uns bestätigt, daß der Startterm NF-terminiert. Später übernehmen wir diese Idee zur Bildung von Dependency-Pairs, ohne das Termersetzungssystem explizit zu erzeugen. Dazu wird die Funktion

$$\text{term}_{\mathsf{NG}} :: Q \times \text{Pot}(Q) \longrightarrow \mathsf{H_B}(\mathsf{D}, \mathsf{V}) \times \text{Pot}(Q)$$

benötigt, welche die Grundlage für das Ablesen von Regeln und Dependency-Pairs aus einem Narrowing-Graph bildet. Die Funktion $\text{term}_{\mathsf{NG}}$ liefert für einen Knoten $(\underline{cs}, u)$ den Term zurück, zu dem u ausgewertet werden kann, ohne dabei an freie Variablen gebundene Werte auswerten zu müssen. Die zweite Komponente vom Ergebnis ist eine Knotenmenge, die Knoten enthält, an denen $\text{term}_{\mathsf{NG}}$ die Verfolgung der Pfade im Narrowing-Graph beendet hat. Der zweite Parameter ist eine Filtermenge für Knoten, an denen $\text{term}_{\mathsf{NG}}$ stoppen soll. Wenn an einem Knoten durch diese Filtermenge gestoppt wird, wird zu diesem eine frische Variable vom selben Typ seines Term und eine leere Knotenmenge zurückgeliefert. Mit Hilfe dieser Filtermenge wird beim Ablesen von Dependency-Pairs gesteuert, zu welchen Knoten Regeln abgelesen werden müssen. Die genaue Erklärung der Filtermenge wird auf später verschoben, wo wir definieren werden, wie Dependency-Pairs abgelesen werden und wie die Menge $\text{freeapps}_{\mathsf{NG}}$ gebildet wird, welche als Filtermenge der Funktion $\text{term}_{\mathsf{NG}}$ zum Einsatz kommt. Hier wird erst einmal angenommen, daß diese Menge leer ist.

Konkret ist die Funktion $\text{term}_{\mathsf{NG}}$ wie folgt definiert:

$$\text{term}_{\mathsf{NG}}(q, Q) := \begin{cases} \text{term}_{\mathsf{NG}}(\text{eval}(q), Q), & \text{für } \mathsf{M}(q) = \underline{\text{eval}} \\ (c\ t_1\ \ldots\ t_n, \cup_{i=1}^{n} R_i), & \text{für } \mathsf{M}(q) = \underline{\text{parsplit}}, \\ & (q, i, q_i) \in \text{subterm}, \\ & (t_i, R_i) = \text{term}_{\mathsf{NG}}(q_i, Q), \\ & q = (\underline{cs}, c\ u_1\ \ldots\ u_n), \\ & c \in \underline{\mathsf{Cons}_{\mathsf{D}}} \\ (u[x_1/t_1, \ldots, x_n/t_n], \cup_{i=1}^{n} R_i), & \text{für } \mathsf{M}(q) = \underline{\text{instance}}, \\ & (q, x_i, q_i) \in \text{subterm}, \\ & (t_i, R_i) = \text{term}_{\mathsf{NG}}(q_i, Q), \\ & (\underline{cs}, u) = \text{generalisation}(q), \\ & q \notin Q \\ (u, \{q\}), & \text{für } \mathsf{M}(q) = \underline{\text{case}}, \\ & q = (\underline{cs}, u), \\ & q \notin Q \\ (u, \emptyset), & \text{für } \mathsf{M}(q) = \underline{\text{varexp}}, \\ & q = (\underline{cs}, u) \\ (v\ t_1\ \ldots\ t_n, \cup_{i=1}^{n} R_i), & \text{für } \mathsf{M}(q) = \underline{\text{parsplit}}, \\ & (q, i, q_i) \in \text{subterm}, \\ & (t_i, R_i) = \text{term}_{\mathsf{NG}}(q_i, Q), \\ & q = (\underline{cs}, v\ u_1\ \ldots\ u_n), \\ & v \in \mathsf{V_L}, \\ & q \notin Q \\ (x_{|\tau|}, \emptyset), & \text{für } \mathsf{M}(q) = \underline{\text{parsplit}}, \\ & q = (\underline{cs}, (v\ u_1\ \ldots\ u_n)_{|\tau|}), \\ & v \in \mathsf{V_L}, \\ & x_{|\tau|} \in \mathsf{V_L} \text{ frische Variable} \\ & q \in Q \\ (x_{|\tau|}, \emptyset), & \text{für } \mathsf{M}(q) \in \{\underline{\text{case}}, \underline{\text{instance}}\}, \\ & q = (\underline{cs}, u_{|\tau|}), \\ & x_{|\tau|} \in \mathsf{V_L} \text{ frische Variable}, \\ & q \in Q \end{cases}$$

Die Arbeitsweise der Funktion $\text{term}_{\mathsf{NG}}$ wird nun an dem Narrowing-Graph aus Abbildung 5.2 zu dem Startterm `f` x (`dec` n) des Haskellprogramms aus Beipspiel 5.2 verdeutlicht Nicht alle Funktionen aus dem Beispiel 5.2 sind im Narrowing-Graph aus Abbildung 5.2 verwendet worden, diese werden später in anderen Beispielen von Narrowing-Graphen noch Verwendung finden.

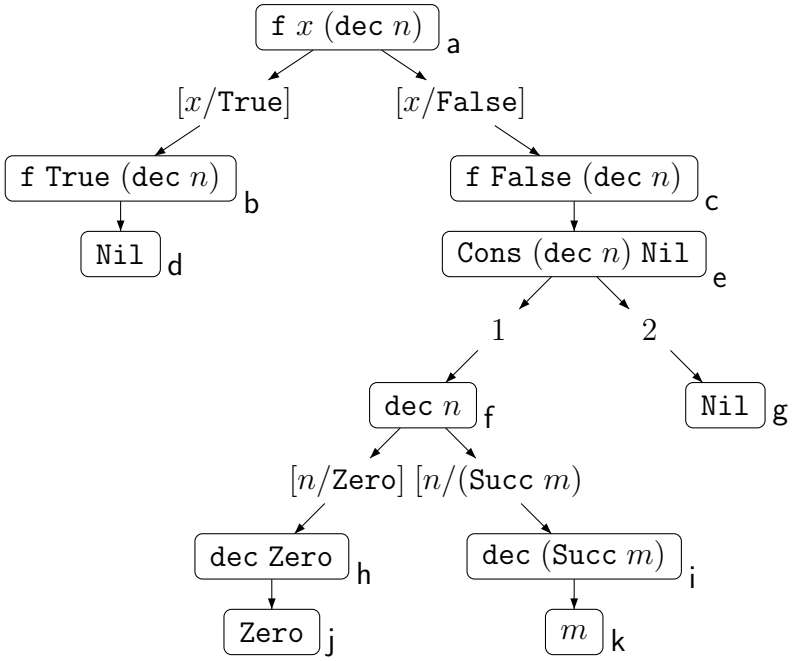

Abbildung 5.2: Ein Narrowing-Graph zu Beispiel 5.2 für Startterm `f` x (`dec` n)

Beispiel 5.2[1]

```
data Nat      = Succ Nat | Zero
data List a   = Cons a (List a) | Nil

f True   y = Nil
f False  y = Cons y Nil

dec Zero     = Zero
dec (Succ m) = m

minus x Zero     = x
minus x (Succ y) = minus (dec x) y

div x        Zero     = Zero
div Zero     (Succ m) = Zero
div (Succ y) (Succ m) = Succ (div (minus y m) (Succ m))

g Zero     y = y
g (Succ n) y = applyTo Zero (g n)

applyTo Zero x = x Zero
```

[1]Die Funktion `g` berechnet `Zero`, wenn der erste Parameter der Form `Succ` y ist.

Beispielsweise ergeben die Aufrufe von $\text{term}_{\mathsf{NG}}$ für jeweils einen der Knoten a, b und c folgende Ergebnisse:

$$\text{term}_{\mathsf{NG}}(\mathsf{a}, \emptyset) = (\mathtt{f}\ x\ (\mathtt{dec}\ n), \{\mathsf{a}\})$$

$$\text{term}_{\mathsf{NG}}(\mathsf{b}, \emptyset) = (\mathtt{Nil}, \emptyset)$$

$$\begin{aligned}\text{term}_{\mathsf{NG}}(\mathsf{c}, \emptyset) = \text{term}_{\mathsf{NG}}(\mathsf{e}, \emptyset) &= (\mathtt{Cons}\ t_1\ t_2, S_1 \cup S_2) = (\mathtt{Cons}\ (\mathtt{dec}\ n)\ \mathtt{Nil}, \{\mathsf{f}\}) \\ \text{wobei}\quad (t_1, S_1) &= \text{term}_{\mathsf{NG}}(\mathsf{f}, \emptyset) = (\mathtt{dec}\ n, \mathsf{f}) \\ (t_2, S_2) &= \text{term}_{\mathsf{NG}}(\mathsf{g}, \emptyset) = (\mathtt{Nil}, \emptyset)\end{aligned}$$

Die Funktion $\text{rules}_{\mathsf{NG}} : \mathsf{H_B}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q} \longrightarrow \text{TES}$ ist mit Hilfe der Funktion $\text{term}_{\mathsf{NG}}$ wie folgt definiert:

$$\text{rules}_{\mathsf{NG}}(l, q) := \begin{cases} \text{rules}_{\mathsf{NG}}(l, \text{eval}(q)), & \text{für } \mathsf{M}(q) = \underline{\text{eval}} \\ \{[\![l]\!]_{\mathsf{APP}} \to [\![u]\!]_{\mathsf{APP}}\}, & \text{für } \mathsf{M}(q) = \underline{\text{varexp}}, q = (\underline{cs}, u) \\ \bigcup\limits_{(q, \delta_i, q_i) \in \text{case}} \text{rules}_{\mathsf{NG}}(l\delta_i, q_i), & \text{für } \mathsf{M}(q) = \underline{\text{case}} \\ \{[\![l]\!]_{\mathsf{APP}} \to [\![t]\!]_{\mathsf{APP}}\} \cup \bigcup\limits_{(\underline{cs}, u) \in W} \text{rules}_{\mathsf{NG}}(u, (\underline{cs}, u)), & \\ \qquad \text{für } \mathsf{M}(q) \in \{\underline{\text{instance}}, \underline{\text{parsplit}}\}, & \\ \qquad (t, W) = \text{term}_{\mathsf{NG}}(q, \emptyset) & \end{cases}$$

Sie erzeugt eine Menge von Regeln, also ein Termersetzungssystem. Ihr erster Parameter ist dabei ein Akkumulator, welcher den Term enthält, der zu der linken Seite einer Regel wird, wenn $\text{rules}_{\mathsf{NG}}$ auf einen Knoten mit den Marken $\underline{\text{parsplit}}$, $\underline{\text{instance}}$ oder $\underline{\text{varexp}}$ trifft. Bei jeder Fallunterscheidung wird auf den Akkumulator die Substitution zugehörig zu dem Pfad, der von $\text{rules}_{\mathsf{NG}}$ weiter verfolgt wird, appliziert. Wenn ab dem Knoten $(\underline{cs}, u)$ Regeln abgelesen werden sollen, muß am Anfang der Akkumulator mit dem Term u gefüllt werden, ansonsten ergeben sich keine korrekten Regeln. Für den Knoten a des Narrowing-Graphen aus Abbildung 5.2 ergibt sich folgendes Termersetzungssystem:

$$\begin{aligned}\text{rules}_{\mathsf{NG}}(\mathtt{f}\ x\ (\mathtt{dec}\ n), \mathsf{a}) = \{ & \mathtt{f'True'}(\mathtt{dec'}n) && \to\ \mathtt{Nil}, \\ & \mathtt{f'False'}(\mathtt{dec'}n) && \to\ \mathtt{Cons'}(\mathtt{dec'}n)\mathtt{'Nil} \\ & \mathtt{dec'Zero} && \to\ \mathtt{Zero}, \\ & \mathtt{dec'}(\mathtt{Succ'}m) && \to\ m \qquad\qquad \}\end{aligned}$$

wobei folgende Zwischenergebnisse berechnet wurden:

$$\begin{aligned}\text{rules}_{\mathsf{NG}}(\mathtt{f}\ x\ (\mathtt{dec}\ n), \mathsf{a}) = &\ \text{rules}_{\mathsf{NG}}(\mathtt{f\ True}\ (\mathtt{dec}\ n), \mathsf{b}) \\ &\cup\ \text{rules}_{\mathsf{NG}}(\mathtt{f\ False}\ (\mathtt{dec}\ n), \mathsf{c})\end{aligned}$$

$$\begin{aligned}\text{rules}_{\mathsf{NG}}(\texttt{f True } (\texttt{dec } n), \mathsf{b}) &= \{\texttt{f'True'}(\texttt{dec'}n) \to \texttt{Nil}\} \\ \text{rules}_{\mathsf{NG}}(\texttt{f False } (\texttt{dec } n), \mathsf{c}) &= \{\texttt{f'False'}(\texttt{dec'}n) \to \texttt{Cons'}(\texttt{dec'}n)\texttt{'Nil}\} \\ &\quad \cup \text{rules}_{\mathsf{NG}}(\texttt{dec } n, \mathsf{f})\end{aligned}$$

$$\begin{aligned}\text{rules}_{\mathsf{NG}}(\texttt{dec } n, \mathsf{f}) = \{&\texttt{dec'Zero} &\to\ &\texttt{Zero}, \\ &\texttt{dec'}(\texttt{Succ'}m) &\to\ &m \quad \}\end{aligned}$$

Dieses Termersetzungssystem terminiert offensichtlich, gleichzeitig sind alle Auswertungen, die von Konstruktorgrundterm-Instanzen des Startterms $\texttt{f } x \ (\texttt{dec } n)$ ausgehen, mit diesem simulierbar (wobei mehrere Schritte zu einem zusammengefaßt werden) und da $\texttt{f } x \ (\texttt{dec } n)$ vom Typ $\texttt{List } a$ ist kann $\texttt{f } x \ (\texttt{dec } n)$ nicht auf andere Terme appliziert werden, so daß aus der Terminierung des Termersetzungssystemes sogar die NF-Terminierung von $\texttt{f } x \ (\texttt{dec } n)$ folgt.

Bis jetzt ist immer noch unklar, wie von Narrowing-Graphen, bei deren Erstellung die Instantiierung zur Anwendung kamen, Regeln abgelesen werden sollen. Die Funktion $\text{rules}_{\mathsf{NG}}$ geht nämlich nicht über Generalisierungskanten hinweg, und so entstehen Term auf rechten Seiten von Regeln, auf die keine weiteren Regeln passen, obwohl diese Terme keine Normalformen im Haskellprogramm sind. Betrachten wir hierzu den Narrowing-Graph aus Abbildung 5.3. In diesem gibt es den Knoten l, aus welchem eine Generalisierungskante zu Knoten f entspringt. Wenn wir nun die Regeln zu Knoten a ablesen, erhalten wir das folgende Termersetzungssystem:

$$\begin{aligned}\text{rules}_{\mathsf{NG}}(\texttt{f } x \ (\texttt{dec } n), \mathsf{a}) = \{&\texttt{f'True'}(\texttt{dec'}n) &\to\ &\texttt{Nil}, \\ &\texttt{f'False'}(\texttt{dec'}n) &\to\ &\texttt{Cons'}(\texttt{dec'}n)\texttt{'Nil}\}\end{aligned}$$

Dieses Termersetzungssystem reicht nun nicht aus, alle Auswertungen, die von Normalform-Instanzen des Startterms $\texttt{f } x \ (\texttt{dec } n)$ ausgehen, zu simulieren. Neben der Normalform Nil kann nur der Term $\texttt{Cons'}(\texttt{dec'n})\texttt{'Nil}$, welcher keine Normalform ist, erreicht werden. Es fehlen Regeln, die den letzten Schritt, je nachdem wie n instantiiert ist, übernehmen. Wenn aber die Regeln zu Knoten f

$$\begin{aligned}\text{rules}_{\mathsf{NG}}(\texttt{dec } n, \mathsf{f}) = \{&\texttt{dec'Zero} &\to\ &\texttt{Zero}, \\ &\texttt{dec'}(\texttt{Succ'}m) &\to\ &m \quad \}\end{aligned}$$

zusätzlich abgelesen werden, können die Normalformen, welche Instanzen von den Termen $\texttt{Cons'Zero'Nil}$ und $\texttt{Cons'}(\texttt{Succ'}n)\texttt{'Nil}$ sind, wieder erreicht werden. Generell läßt sich sagen, daß für alle Knoten, in die Generalisierungskanten führen, Regeln abgelesen und zu denen vom Startknoten hinzugenommen werden müssen.

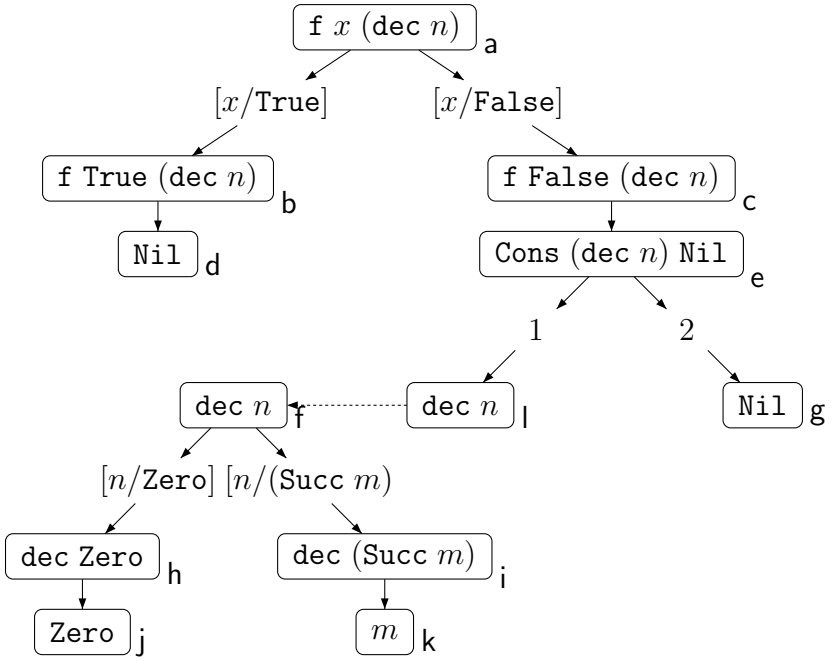

Abbildung 5.3: Ein Narrowing-Graph zu Beispiel 5.2 für Startterm f x (dec n)

In der Definition der Funktion $\text{rules}_{\mathsf{NG}}$ wurde absichtlich darauf verzichtet, daß über Generalisierungskanten hinweg Regeln abgelesen werden. Durch das Zykellemma 4.8 ist so sichergestellt, daß $\text{rules}_{\mathsf{NG}}$ und $\text{term}_{\mathsf{NG}}$ terminieren. Im Abschnitt 5.3 wird diese Eigenschaft häufiger für Induktionsbeweise über die Rekursionsstrukturen von den Funktionen $\text{rules}_{\mathsf{NG}}$ und $\text{term}_{\mathsf{NG}}$ gebraucht.

Die bis jetzt abgelesenen Regeln stellen für den first-order Fall von Starttermen sicher, daß diese NF-terminieren, wenn das zugehörige Termersetzungssystem terminiert, weil dabei NF-Terminierung und normale Terminierung äquivalent sind. Für higher-order Startterme ist nicht klar, wie Regeln abgelesen werden sollen, denn dabei treten im Narrowing-Graph möglicherweise Variablenexpansionen und Parameteraufteilung für freie Applikationen der Form $v\ t_1\ \ldots\ t_n$ mit $v \in \mathsf{V_L}$ auf. Es ist bis jetzt unklar, wie damit in einem Termersetzungssystem umgegangen werden soll. Seien zum Beispiel folgende Regeln vom Narrowing-Graphen abgelesen worden:

$$\begin{array}{rlcl} \mathcal{R} := & \mathtt{f}'x'\mathtt{True} & \to & \mathtt{f}'x'(x'\mathtt{False}) \\ & \mathtt{f}'x'\mathtt{False} & \to & \mathtt{False} \\ & \mathtt{g}'y & \to & y \end{array}$$

Die auftretende freie Applikation x False wurde einfach wie ein normaler Term behandelt. Nehmen wir weiter an, daß y im Narrowing-Graph vom Typ Bool

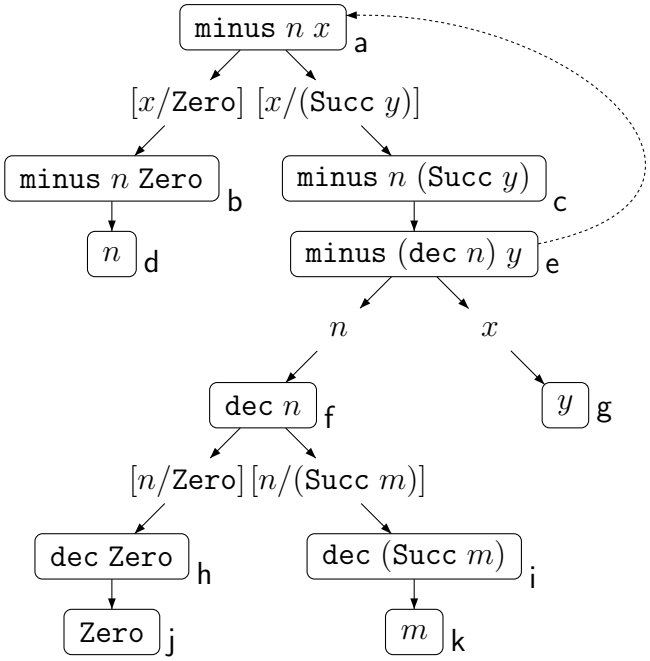

Abbildung 5.4: Ein zyklischer Narrowing-Graph zu Beispiel 5.2 für Startterm `minus` n x

ist und x vom Typ `Bool` $\rightarrow$ `Bool` ist. Jetzt zeigt sich, daß dieses Termersetzungssystem terminiert, wobei die Typen mit beachtet werden müssen und für x nur Funktion, die innerhalb des Termersetzungssystems $\mathcal{R}$ vorkommen, eingesetzt werden dürfen, denn so kann die Funktion `f` keinen unendlichen Lauf starten. Die Funktion `f` NF-terminiert aber nicht, weil für x eine Funktion eingesetzt werden kann, die immer `True` zurückliefert, denn diese kommt möglicherweise in dem ursprünglichen Haskellprogramm vor, wir wissen es aber nicht, denn in einem Narrowing-Graph werden nicht immer alle Funktionen eines Haskellprogramms explizit berücksichtigt, sondern vielfach nur implizit durch lokale Variablen. Es gibt aber, wie wir sehen werden, eine Möglichkeit für Narrowing-Graphen, die Variablenexpansion und freien Applikationen enthalten, DP-Probleme so abzulesen, daß aus deren Endlichkeitsnachweis die NF-Terminierung des Startterms folgt. Betrachten wir hierzu den zyklischen Narrowing-Graph in Abbildung 5.4. Zu diesem erhalten wir durch die Ablesestrategie das Termersetzungssystem $\mathcal{R}_{\texttt{minus}}$. Es entspricht genau der Regelmenge vom Knoten `a`:

$$\begin{aligned}\mathcal{R}_{\texttt{minus}} := \text{rules}_{\mathsf{NG}}(\texttt{minus}\ n\ x, \mathsf{a}) = \{&\texttt{minus}'n'\texttt{Zero} && \rightarrow && n,\\ &\texttt{minus}'n'(\texttt{Succ}'y) && \rightarrow && \texttt{minus}'(\texttt{dec}\ n)'y,\\ &\texttt{dec}'\texttt{Zero} && \rightarrow && \texttt{Zero},\\ &\texttt{dec}'(\texttt{Succ}'m) && \rightarrow && m \qquad\qquad \}\end{aligned}$$

Nach der Dependency-Pair-Methode aus [GTSK05] ergibt sich zu $\mathcal{R}$ die Menge $\mathcal{P}_{\texttt{minus}}$ von Dependency-Pairs folgendermaßen:

$$\begin{aligned}\mathcal{P}_{\texttt{minus}} := \{\ &\texttt{minus}'n^\sharp(\texttt{Succ}'y) \ \rightarrow\ \texttt{minus}'(\texttt{dec}\ n)^\sharp y,\\ &\texttt{minus}'n^\sharp(\texttt{Succ}'y) \ \rightarrow\ \texttt{minus}^\sharp(\texttt{dec}\ n)\ \ \}\end{aligned}$$

wobei das Symbol $^\sharp$ das Tupelsymbol zu $'$ (also zu *app*) ist. Es ist zu erkennen, daß das Dependency-Pair $\texttt{minus}'n^\sharp(\texttt{Succ}'y) \rightarrow \texttt{minus}'(\texttt{dec}\ n)^\sharp y$ das wirklich relevante zu dem Termersetzungssystem $\mathcal{R}_{\texttt{minus}}$ ist, da das andere nicht in der Lage ist, in Zyklen aufzutreten.

Mit Hilfe der Funktion dps_{NG} können diese relevanten Dependency-Pairs direkt aus Narrowing-Graphen zu den einzelnen maximalen starken Zusammenhangskomponenten abgelesen werden. Der erste Parameter für dps_{NG} wird dabei immer ein Knoten $(\underline{cs}, u) \in Z$ aus der maximalen starken Zusammenhangskomponente Z sein, diese wird dps_{NG} ebenfalls als zweiter Parameter übergeben.

$$\begin{aligned}&\text{dps}_{\mathsf{NG}} : \mathsf{Q} \times \text{Pot}(\mathsf{Q}) \longrightarrow \text{TES}\\ &\text{dps}_{\mathsf{NG}}((\underline{cs}, u), Z) := \{[[\![l]\!]_{\mathsf{APP}}]^\sharp \rightarrow [[\![r]\!]_{\mathsf{APP}}]^\sharp \mid (l, q, (r, R)) \in \text{calls}_{\mathsf{NG}}(u, (\underline{cs}, u), Q)\}\end{aligned}$$

Die Funktion dps_{NG} basiert auf der Funktion $\text{calls}_{\mathsf{NG}}$. Die Funktion $\text{calls}_{\mathsf{NG}}$ sammelt zum Eingabeknoten $(\underline{cs}, u)$ aus der maximalen starken Zusammenhangskomponente Z, die ihr als dritter Parameter ebenfalls übergeben wird, alle Nachfolgerknoten auf, aus denen jeweils Generalisierungskanten entspringen, wobei sie diese nicht überschreitet. Die Funktion $\text{calls}_{\mathsf{NG}}$ steigt also in den Narrowing-Baum ab, welcher Knoten $(\underline{cs}, u)$ als Wurzel besitzt. Die aufgesammelten Nachfolger entsprechen den Aufrufen, die bei einer Auswertung des Terms aus Knoten $(\underline{cs}, u)$ möglicherweise erreicht werden.

$$\text{calls}_{\mathsf{NG}} : \mathsf{H_B}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q} \times \text{Pot}(\mathsf{Q}) \longrightarrow \text{Pot}(\mathsf{H_B}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q} \times (\mathsf{H_B}(\mathsf{D}, \mathsf{V}) \times \text{Pot}(\mathsf{Q})))$$

$$\text{calls}_{\mathsf{NG}}(t, q, Z) := \begin{cases} \text{calls}_{\mathsf{NG}}(t, \text{eval}(q), Z), & \text{für } \mathsf{M}(q) = \underline{\text{eval}}\\ \text{calls}_{\mathsf{NG}}(t, \text{varexp}(q), Z), & \text{für } \mathsf{M}(q) = \underline{\text{varexp}}\\ \bigcup\limits_{(q,i,q') \in \text{subterm}} \text{calls}_{\mathsf{NG}}(t, q', Z), & \text{für } \mathsf{M}(q) = \underline{\text{parsplit}}\\ \bigcup\limits_{(q,\delta,q') \in \text{case}} \text{calls}_{\mathsf{NG}}(t\delta, q', Z), & \text{für } \mathsf{M}(q) = \underline{\text{case}}\\ \{(t, q, \text{term}_{\mathsf{NG}}(q, \text{freeapps}_{\mathsf{NG}} \setminus \{q\}))\} & \\ \quad \cup \bigcup\limits_{(q,i,q') \in \text{subterm}} \text{calls}_{\mathsf{NG}}(t, q', Z), & \text{für } \mathsf{M}(q) = \underline{\text{instance}},\\ & \quad \text{generalisation}(q) \in Z\\ \bigcup\limits_{(q,i,q') \in \text{subterm}} \text{calls}_{\mathsf{NG}}(t, q', Z), & \text{für } \mathsf{M}(q) = \underline{\text{instance}},\\ & \quad \text{generalisation}(q) \notin Z \end{cases}$$

Der erste Parameter von $\text{calls}_{\mathsf{NG}}$ ist ein Term, der als Akkumulator dient, ähnlich wie in der Funktion $\text{rules}_{\mathsf{NG}}$. Auf diesen Akkumulator werden die auf den Pfaden ab $(\underline{cs}, u)$ besuchten Substitutionen der Fallunterscheidungen appliziert. Wenn beim Abstieg in den Narrowing-Baum, den der Knoten $(\underline{cs}, u)$ aufspannt, eine Generalisierungskante erreicht wird, entspricht der Akkumulator genau der Instanz des Terms u von Knoten $(\underline{cs}, u)$, welche bei Auswertung den Funktionsaufruf erreicht, dem die Generalisierungskante entspricht. Durch den letzten Parameter, in dem sich die maximale starke Zusammenhangskomponente Z befindet, liefert die Funktion $\text{calls}_{\mathsf{NG}}$ nur Aufrufe zurück, die innerhalb von Z liegen, weil die anderen Aufrufe nicht benötigt werden. Der Aufruf von $\text{term}_{\mathsf{NG}}$ innerhalb von $\text{calls}_{\mathsf{NG}}$ bewirkt, daß eine soweit wie möglich ausgewertete rechte Seite eines Dependency-Pairs entsteht, und insbesondere werden die Knoten, zu denen Regeln für das gerade abgelesene Dependency-Pair gebildet werden müssen, in der letzten Ergebniskomponente von $\text{calls}_{\mathsf{NG}}$ zurückgeliefert. Die Funktion dps_{NG} verwirft die Menge R, die sie von $\text{calls}_{\mathsf{NG}}$ zurückgeliefert bekommt, während die Funktion $\text{neededRules}_{\mathsf{NG}}$, die später noch vorgestellt wird, genau diese letzte Ergebniskomponente von $\text{calls}_{\mathsf{NG}}$ benutzt, um für die Knoten darin Regeln abzulesen. Die Bedeutung der Menge $\text{freeapps}_{\mathsf{NG}}$ wird später erklärt. Sie wird Knoten enthalten, zu denen *keine* Regeln abgelesen werden dürfen, während sie hier erst einmal als leer angenommen wird.

Für den Narrowing-Graph aus Abbildung 5.4 wird durch die Funktion dps_{NG} für den Knoten a aus der maximalen starken Zusammenhangskomponente $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$ die folgende Menge von Dependency-Pairs berechnet:

$$\text{dps}_{\mathsf{NG}}(\mathsf{a}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) = \{\mathtt{minus}'n^\sharp(\mathtt{Succ}'y) \to \mathtt{minus}'(\mathtt{dec}\ n)^\sharp y\}$$

wobei die folgenden Zwischenergebnisse von $\text{calls}_{\mathsf{NG}}$ berechnet wurden:

$$\begin{aligned}
&\text{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ x, \mathsf{a}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&= \text{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ \mathtt{Zero}, \mathsf{b}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&\quad \cup\ \text{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{c}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\})
\end{aligned}$$

$$\begin{aligned}
&\text{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ \mathtt{Zero}, \mathsf{b}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&= \text{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ \mathtt{Zero}, \mathsf{d}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&= \emptyset
\end{aligned}$$

$$
\begin{aligned}
&\mathrm{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{c}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&= \mathrm{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{e}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&= \{(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{e}, \mathrm{term}_{\mathsf{NG}}(\mathsf{e}, \mathrm{freeapps}_{\mathsf{NG}} \setminus \{\mathsf{e}\}))\} \\
&\quad \cup \mathrm{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{f}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&\quad \cup \mathrm{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{g}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\
&= \{(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{e}, \mathrm{term}_{\mathsf{NG}}(\mathsf{e}, \emptyset))\} \\
&= \{(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{e}, \mathtt{minus}'(\mathtt{dec}\ n)'y)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{f}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) &= \emptyset \\
\mathrm{calls}_{\mathsf{NG}}(\mathtt{minus}\ n\ (\mathtt{Succ}\ y), \mathsf{g}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) &= \emptyset
\end{aligned}
$$

Es erscheint so, als würde $\mathrm{calls}_{\mathsf{NG}}$ zu viele Dependency-Pairs ablesen, da diese Funktion im Gegensatz zu $\mathrm{rules}_{\mathsf{NG}}$ über Variablenexpansionskanten hinweg geht. Betrachten wir dazu den Narrowing-Graph in Abbildung 5.5, der eine Variablenexpansion für Knoten f enthält.

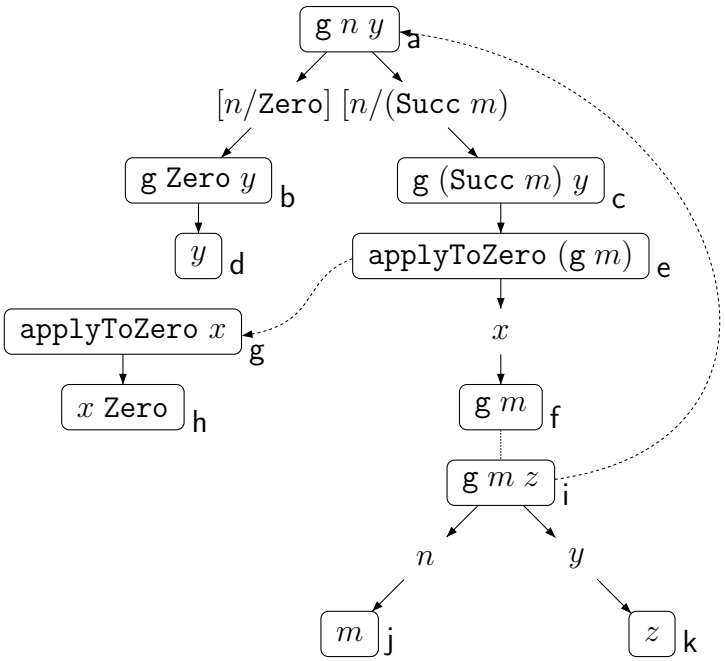

Abbildung 5.5: Ein zyklischer Narrowing-Graph zu Beispiel 5.2 für Startterm g n y

Weil der Teilgraph ab Knoten g keine Zyklen enthält, werden zu diesem keine Dependency-Pairs erzeugt. Der Zyklus ab Knoten a enthält eine Variablenexpansionskante. Im Term von Knoten e ist unklar, wie die Funktion g m innerhalb von `applyToZero` verwendet wird, oder ob diese überhaupt verwendet wird. Da diese Frage unentscheidbar ist, gehen wir in diesen Situationen immer davon aus,

daß eine übergebene Funktion verwendet wird. Um jegliche Verwendung von $\mathsf{g}\ m$ zu simulieren, hat die Variablenexpansion bereits die frische Variable z an den Term $\mathsf{g}\ m$ appliziert. Es kann sein, daß eine Instanz des Terms $\mathsf{g}\ n\ y$ aus Knoten a eine Auswertung einer Instanz des Terms $\mathsf{g}\ m\ z$ aus Knoten i anstößt. Aus diesem Grund muß das Dependency-Pair ausgehend von Knoten a nach Knoten i gebildet werden. Deswegen ergibt sich die Menge

$$\mathcal{P}_{\mathsf{g}} := \mathrm{dps}_{\mathsf{NG}}(\mathsf{a}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}, \mathsf{f}, \mathsf{i}\}) = \{\mathsf{g}'(\mathtt{Succ}'m)^\sharp y \to \mathsf{g}'m^\sharp z\}$$

von Dependency-Pairs für den Zyklus $\{\mathsf{a}, \mathsf{c}, \mathsf{e}, \mathsf{f}, \mathsf{i}\}$, wobei die Berechnung von $\mathrm{calls}_{\mathsf{NG}}(\mathsf{g}\ n\ y, \mathsf{a}, \{\mathsf{a}, \mathsf{c}, \mathsf{e}, \mathsf{f}, \mathsf{i}\})$ über die Variablenexpansion von Knoten f hinweg geht und so den Knoten i erreicht. Natürlich ist es in dem Narrowing-Graph in Abbildung 5.5 leicht sichtbar, das $\mathsf{g}\ m$ immer auf `Zero` appliziert wird, aber die Funktion `applyToZero` könnte ja wesentlich komplexer sein, so daß keine einfache Heuristik die Parameter leicht abschätzen könnte, da diese Frage, worauf $\mathsf{g}\ m$ innerhalb einer anderen Funktion angewendet wird, wie bereits erwähnt, unentscheidbar ist.

Zu einem DP-Problem gehört nicht nur eine Menge von Dependency-Pairs, sondern auch eine Menge von Regeln. Bei der Dependency-Pair-Methode aus [AG00] wären das alle Regeln eines Termersetzungssystems. Andererseits wird dort im Abschnitt 3.2 die Menge der Usable rules aus Definition 30 für die Innermost-Auswertungsstrategie vorgestellt. Es hängt also von der Auswertungsstrategie ab, welche Regeln für die jeweiligen Dependency-Pairs abgelesen werden müssen. Diesen Zusammenhang nutzt die Funktion $\mathrm{neededRules}_{\mathsf{NG}} : \mathsf{Q} \times \mathrm{Pot}(\mathsf{Q}) \longrightarrow \mathrm{TES}$, welche die Regeln so aufsammelt, wie sie für die von dem Eingabeknoten aus gebildeten Dependency-Pairs unter Beachtung der Haskellauswertungsstrategie benötigt werden. Sie ist folgendermaßen definiert:

$$\mathrm{neededRules}_{\mathsf{NG}}((\underline{cs}, u), Q) := \bigcup_{(\underline{cs}', u') \in \mathrm{neededNodes}_{\mathsf{NG}}(\mathrm{calls}_{\mathsf{NG}}(u, (\underline{cs}, u), Q))} \mathrm{rules}_{\mathsf{NG}}(u', (\underline{cs}', u'))$$

Die Entscheidung, zu welchem Knoten nun wirklich Regeln gebildet werden, wird dabei durch die Funktion

$$\mathrm{neededNodes}_{\mathsf{NG}} : \mathrm{Pot}(\mathsf{H}_{\mathsf{B}}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q} \times (\mathsf{H}_{\mathsf{B}}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q})) \longrightarrow \mathrm{Pot}(\mathsf{Q})$$

getroffen. Sie sammelt gerade die Knoten auf, welche von den Parameterknoten eines Instanzknotens erreicht werden können. Die anderen Regeln, welche im normalen Dependency-Pair-Ansatz auch noch hinzugenommen werden müßten, sind hier nicht nötig, da der Narrowing-Graph durch seinen Aufbau garantiert, daß diese nicht durch die Haskellauswertungsstrategie aufgerufen werden. Von diesen erreichbaren Knoten werden die ersten Knoten, an denen eine Fallunterscheidung angewendet worden ist, in die Ergebnismenge aufgenommen und von

da ab nur noch die Nachfolger, in die eine Generalisierungskante zeigt. Konkret bewerkstelligt sie das wie folgt:

$$\text{neededNodes}_{\mathsf{NG}}(K) := \Bigg(\bigcup_{(l,q,(r,Q))\in K} Q \;\cup \bigcup_{\substack{(l,\,q,\,(r,\,Q))\in\, K \\ (q,\,x,\,q')\in \text{ subterm} \\ m\in\,\mathbb{N}}} \left(\text{successors}^m_{\mathsf{NG}}(\{q'\}) \cap \text{range}(\text{generalisation})\right) \Bigg) \setminus \text{freeapps}_{\mathsf{NG}}$$

$$\begin{aligned} &\text{successors}_{\mathsf{NG}} : \text{Pot}(\mathsf{Q}) \longrightarrow \text{Pot}(\mathsf{Q}) \\ &\text{successors}_{\mathsf{NG}}(Q) := \{q \in \mathsf{Q} \mid q' \in Q \;\wedge\; ((q',\sigma,q) \in \text{case} \\ &\qquad\qquad \vee\, (q',i,q) \in \text{subterm} \\ &\qquad\qquad \vee\, \text{eval}(q') = q \\ &\qquad\qquad \vee\, \text{generalisation}(q') = q)\ \} \end{aligned}$$

Ein Element $(l,q,(r,Q)) \in K$, wobei K eine Ergebnismenge von $\text{calls}_{\mathsf{NG}}$ ist, enthält die Knotenmenge Q, in der laut den Definitionen von $\text{term}_{\mathsf{NG}}$ und $\text{calls}_{\mathsf{NG}}$ die ersten Knoten ab den Parameternachfolgern von q enthalten sind, auf welchen Fallunterscheidungen angewendet worden sind. Zu allen Parameternachfolgern von q werden (erreichbar durch subterm) die Nachfolger aufgesammelt, in die Generalisierungskanten zeigen.

Aus dem Narrowing-Graph in Abbildung 5.6 wird durch die Funktion dps_{NG} für Knoten a und der maximalen starken Zusammenhangskomponente $\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}$ die Menge $\mathcal{P}_{\mathtt{div}}$

$$\begin{aligned} \mathcal{P}_{\mathtt{div}} :=& \text{dps}_{\mathsf{NG}}(\mathsf{a},\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}) \\ =& \{\mathtt{div}'(\mathtt{Succ}'y)^\sharp(\mathtt{Succ}'m) \to \mathtt{div}'(\mathtt{minus}'y'm)^\sharp(\mathtt{Succ}'m)\} \end{aligned}$$

von Dependency-Pairs abgelesen, und zu $\mathcal{P}_{\mathtt{div}}$ erzeugt $\text{neededRules}_{\mathsf{NG}}$ die folgende Menge von Regeln:

$$\begin{aligned} \mathcal{R}_{\mathtt{div}} :=& \text{neededRules}_{\mathsf{NG}}(\mathsf{a},\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}) \\ =& \{\mathtt{minus}'n'\mathtt{Zero} \;\to\; n, \\ & \;\;\mathtt{minus}'n'(\mathtt{Succ}'y) \;\to\; \mathtt{minus}'(\mathtt{dec}\ n)'y, \\ & \;\;\mathtt{dec}'\mathtt{Zero} \;\to\; \mathtt{Zero}, \\ & \;\;\mathtt{dec}'(\mathtt{Succ}'m) \;\to\; m \qquad\qquad \} \end{aligned}$$

Bei dieser Berechnung sind folgende Zwischenergebnisse berechnet worden:

$$\begin{aligned} \text{neededRules}_{\mathsf{NG}}(\mathsf{a},\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}) &= \bigcup_{(\underline{cs}',\,u')\in\, \text{neededNodes}_{\mathsf{NG}}(\text{calls}_{\mathsf{NG}}(\mathtt{div}\ x\ n,\mathsf{a},\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}))} \text{rules}_{\mathsf{NG}}(u',(\underline{cs}',u')) \\ &= \text{rules}_{\mathsf{NG}}(\mathtt{minus}\ n\ x,\mathsf{o}) \end{aligned}$$

$$
\begin{aligned}
&\text{neededNodes}_{\mathsf{NG}}(\text{calls}_{\mathsf{NG}}(\mathtt{div}\ x\ n, \mathsf{a}, \{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\})) \\
&= \text{neededNodes}_{\mathsf{NG}}(\{(\ldots,\mathsf{i},(\ldots,\emptyset))\}) \\
&= \text{successors}^m_{\mathsf{NG}}(\{\mathsf{i}\}) \cup \text{range}(\text{generalisation}) \\
&= \bigcup_{(\mathsf{i},x,q')\in\text{subterm}} \text{successors}^m_{\mathsf{NG}}(\{q'\}) \cap \{\mathsf{a},\mathsf{o}\} \\
&= (\text{successors}^m_{\mathsf{NG}}(\{\mathsf{j}\}) \cap \{\mathsf{a},\mathsf{o}\}) \cup (\text{successors}^m_{\mathsf{NG}}(\{\mathsf{k}\}) \cap \{\mathsf{a},\mathsf{o}\}) \\
&= \text{successors}^m_{\mathsf{NG}}(\{\mathsf{k}\}) \cap \{\mathsf{a},\mathsf{o}\} \\
&= \{\mathsf{o}\}
\end{aligned}
$$

Da nun erklärt ist, wie die beiden Mengen $\mathcal{P}$ und $\mathcal{R}$ eines DP-Problems zu einer maximalen starken Zusammenhangskomponente direkt aus einem Narrowing-Graph abgelesen werden können, kann auch die Funktion

$$\text{dpProblem}_{\mathsf{NG}} :: \text{Pot}(\mathsf{Q}) -> \text{TES} \times \text{TES}$$

wie folgt definiert werden:

$$
\begin{aligned}
\text{dpProblem}_{\mathsf{NG}}(Z) :=& (\bigcup_{q\in Z'} \text{dps}_{\mathsf{NG}}(q), \bigcup_{q\in Z'} \text{neededRules}_{\mathsf{NG}}(q, Z)) \\
&\text{mit } Z' := Z \cap \text{range}(\text{generalisation})
\end{aligned}
$$

Die Funktion $\text{dpProblem}_{\mathsf{NG}}$ liest, unter der Verwendung der Funktionen dps_{NG} und $\text{neededRules}_{\mathsf{NG}}$, für die maximale starke Zusammenhangskomponente Z das zugehörige DP-Problem ab. Davor jedoch sortiert sie Knoten der Zusammenhangskomponente aus, in welche keine Generalisierungskanten zeigen, denn für diese sind keine Dependency-Pairs und Regeln nötig.

Zum Beipspiel ergibt sich für die maximale starke Zusammenhangskomponente $\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}$ des Narrowing-Graphen in Abbildung 5.6 das folgende DP-Problem:

$$\text{dpProblem}_{\mathsf{NG}}(\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{g},\mathsf{i}\}) := (\mathcal{P}_{\mathtt{div}}, \mathcal{R}_{\mathtt{div}})$$

während wir für die andere maximale starke Zusammenhangskomponente $\{\mathsf{o},\mathsf{q},\mathsf{s}\}$ dieses DP-Problem erhalten:

$$
\begin{aligned}
\text{dpProblem}_{\mathsf{NG}}(\{\mathsf{o},\mathsf{q},\mathsf{s}\}) = (\{&\mathtt{minus'}n^\sharp(\mathtt{Succ'}y) \rightarrow \mathtt{minus'}(\mathtt{dec}\ n)^\sharp y\}, \\
&\{\mathtt{dec'Zero} \rightarrow \mathtt{Zero}, \\
&\ \mathtt{dec'}(\mathtt{Succ'}m) \rightarrow m \qquad\qquad \})
\end{aligned}
$$

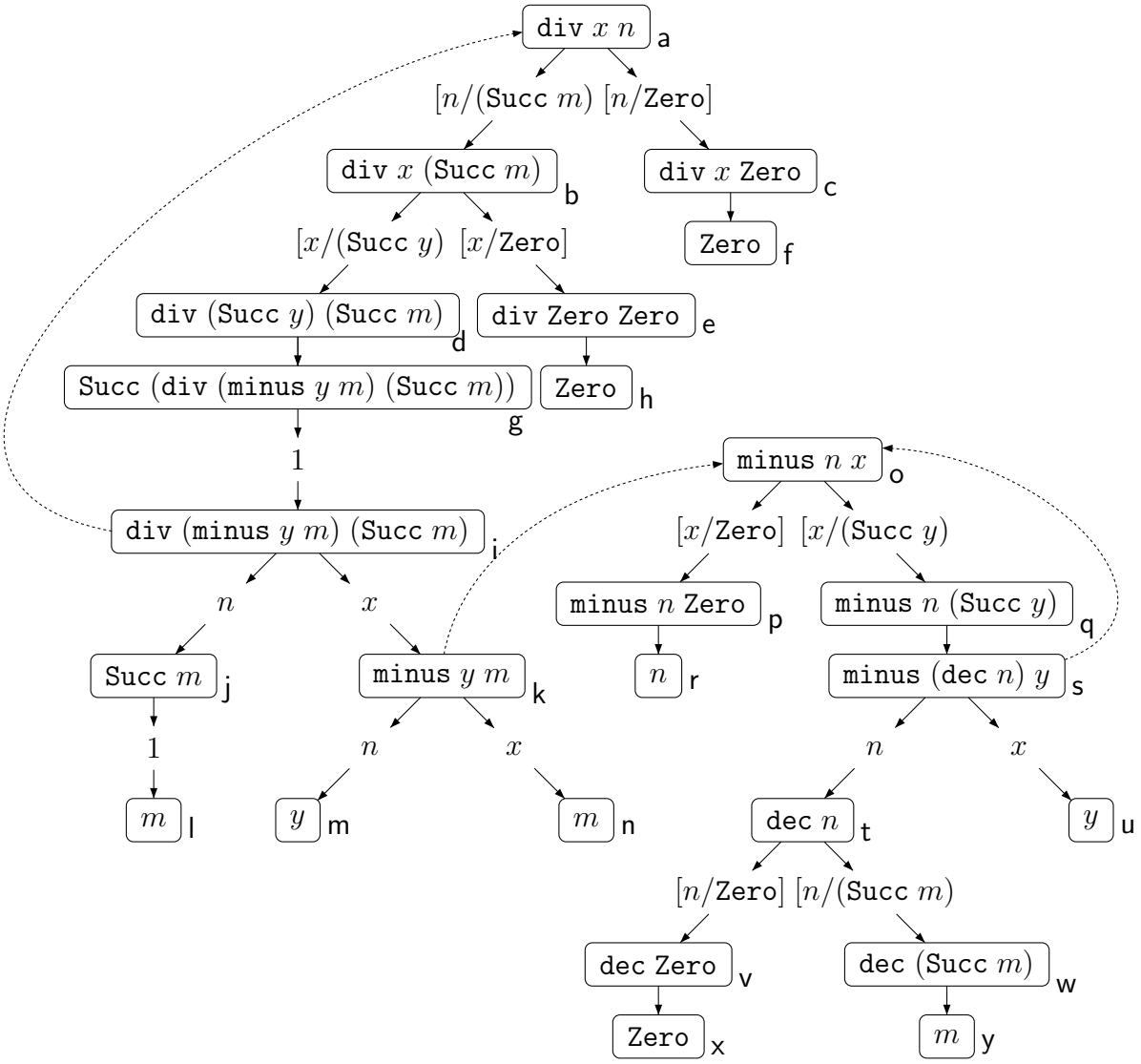

Abbildung 5.6: Ein zyklischer Narrowing-Graph zu Beispiel 5.2 für Startterm div x n

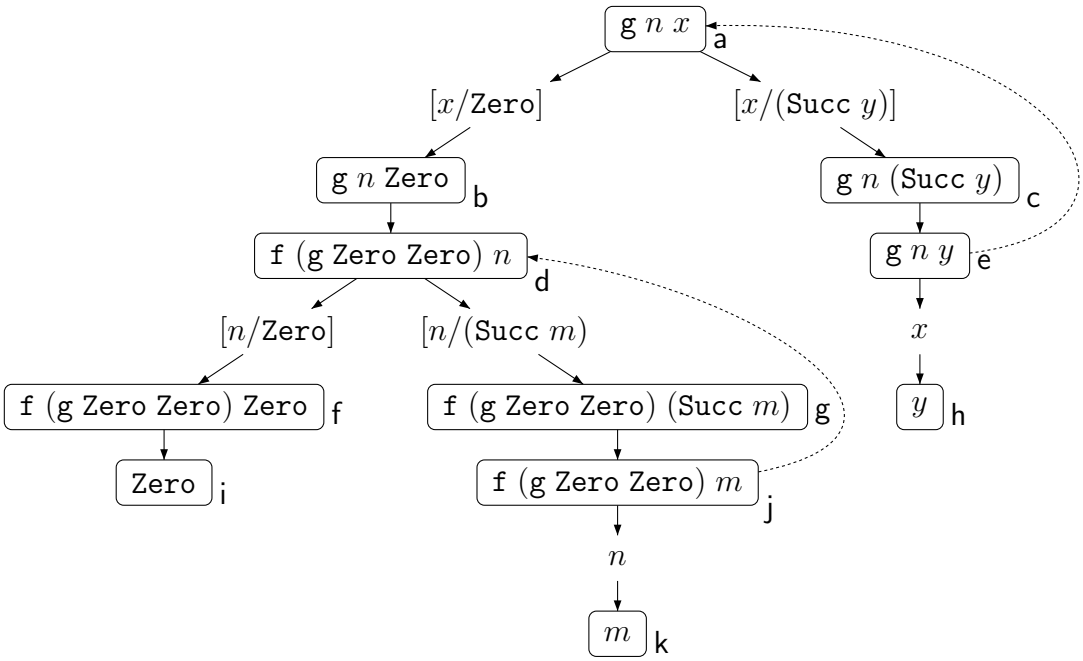

Abbildung 5.7: Ein zyklischer Narrowing-Graph zu Beispiel 5.3 für Startterm $\mathsf{g}\ n\ x$

Betrachten wir einmal das folgende Haskellprogramm:

Beispiel 5.3

$$\begin{array}{lllll} \mathsf{g} & n & \mathsf{Zero} & = & \mathsf{f}\ (\mathsf{g}\ \mathsf{Zero}\ \mathsf{Zero})\ n \\ \mathsf{g} & n & (\mathsf{Succ}\ y) & = & \mathsf{g}\ n\ y \end{array}$$

$$\begin{array}{lllll} \mathsf{f} & w & \mathsf{Zero} & = & \mathsf{Zero} \\ \mathsf{f} & w & (\mathsf{Succ}\ m) & = & \mathsf{f}\ w\ m \end{array}$$

Zu diesem ergibt sich zum Startterm $\mathsf{g}\ n\ x$ der Narrowing-Graph in Abbildung 5.7. In diesem Narrowing-Graph gibt es die beiden maximalen starken Zusammenhangskomponenten $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$ und $\{\mathsf{d}, \mathsf{g}, \mathsf{j}\}$, zu welchen die beiden DP-Probleme A und B abgelesen werden können:

$$\begin{aligned} A &:= \text{dpProblem}_{\mathsf{NG}}(\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) \\ &= (\{\mathsf{g}'n^\sharp(\mathsf{Succ}\ y) \to \mathsf{g}'n^\sharp y\}, \emptyset) \end{aligned}$$

$$\begin{aligned} B &:= \text{dpProblem}_{\mathsf{NG}}(\{\mathsf{d}, \mathsf{g}, \mathsf{j}\}) \\ &= (\{\mathsf{f}'(\mathsf{g}'\mathsf{Zero}'\mathsf{Zero})^\sharp(\mathsf{Succ}'m) \to \mathsf{f}'(\mathsf{g}'\mathsf{Zero}'\mathsf{Zero})^\sharp m\}, \emptyset) \end{aligned}$$

Die Endlichkeit der DP-Probleme A und B kann mit AProVE nachgewiesen werden. Das folgende Termersetzungssystem $\mathcal{R}_{\mathsf{a,d}}$ wurde durch direktes Ablesen

der Regeln ab den beiden Knoten a und d, in die jeweils Generalisierungskanten zeigen, erzeugt:

$$\begin{aligned}\mathcal{R}_{\mathtt{a,d}} &= \text{rules}_{\mathsf{NG}}(\mathtt{g}\ n\ x, \mathtt{a}) \cup \text{rules}_{\mathsf{NG}}(\mathtt{f}\ (\mathtt{g}\ \mathtt{Zero}\ \mathtt{Zero})\ n, \mathtt{d}) \\ &= \{\ \begin{array}{lcll} \mathtt{g}'n'(\mathtt{Succ}\ y) & \to & \mathtt{g}'n'y, & \\ \mathtt{g}'n'\mathtt{Zero} & \to & \mathtt{f}'(\mathtt{g}'\mathtt{Zero}'\mathtt{Zero})'n, & \\ \mathtt{f}'(\mathtt{g}'\mathtt{Zero}'\mathtt{Zero})'(\mathtt{Succ}'m) & \to & \mathtt{f}'(\mathtt{g}'\mathtt{Zero}'\mathtt{Zero})'m, & \\ \mathtt{f}'(\mathtt{g}'\mathtt{Zero}'\mathtt{Zero})'\mathtt{Zero} & \to & \mathtt{Zero} & \} \end{array}\end{aligned}$$

Es ist zu erkennen, daß die zweite Regel nicht terminiert und AProVE weist sogar das Termersetzungssystemen $\mathcal{R}_{\mathtt{a,d}}$ als nicht-terminierend aus. Im Gegensatz zu dem Termersetzungssystem $\mathcal{R}_{\mathtt{a,d}}$ ist in die Bildung der DP-Probleme A und B die Information eingeflossen, daß die Funktion f ihr erstes Argument gar nicht braucht. Die Information geht beim direkten Regelablesen aus dem Narrowing-Graph wieder verloren, so daß in dem Termersetzungssystem, in dem jede Reduktionsstrategie erlaubt ist, nicht mehr angenommen werden kann, daß der Teilterm g′Zero′Zero von der rechten Seite der zweiten Regel keine Auswertung anstößt. Der Narrowing-Graph ist zwar keine perfekte Repräsentation des Auswertungsverhaltens eines Terms, aber bei seinem Aufbau wird innerhalb der Auswertungs-Transformation exakt die Auswertungsstrategie von Haskell eingehalten. Es ist also immer ratsam, sogar im first-order Fall nicht den Umweg über ein Termersetzungssystem zu nehmen und daraus DP-Probleme zu erzeugen, sondern direkt DP-Probleme aus dem Narrowing-Graph abzulesen.

Neben diesen Narrowing-Graphen und ihren dazugehörigen DP-Problemen gibt es aber auch Narrowing-Graphen, in denen ein subtiles Problem zutage tritt, welches ebenfalls behandelt werden muß. Diese Behandlung wird mit der Menge $\text{freeapps}_{\mathsf{NG}}$ gesteuert. Betrachten wir hierzu den Narrowing-Graph in Abbildung 5.8 zum folgenden Haskellprogramm[2]:

Beispiel 5.4

$$\begin{array}{llllll} \mathtt{appN}\ \mathtt{Zero} & x & e & = & e & \\ \mathtt{appN}\ (\mathtt{Succ}\ m) & x & e & = & x\ (\mathtt{appN}\ m\ x\ e) & \\ \\ \mathtt{h}\ n\ e\ \mathtt{Nil} & & & = & e & \\ \mathtt{h}\ n\ e\ (\mathtt{Cons}\ x\ xs) & & & = & \mathtt{h}\ n\ (\mathtt{appN}\ n\ x\ e)\ xs & \end{array}$$

[2]Die Funktion appN berechnet die n-fache Applikation von x auf e. Die Funktion h berechnet $\mathtt{h}\ n\ e\ [f_1, \ldots, f_m] = f_m^n(\ldots f_2^n(f_1^n(e))\ldots)$.

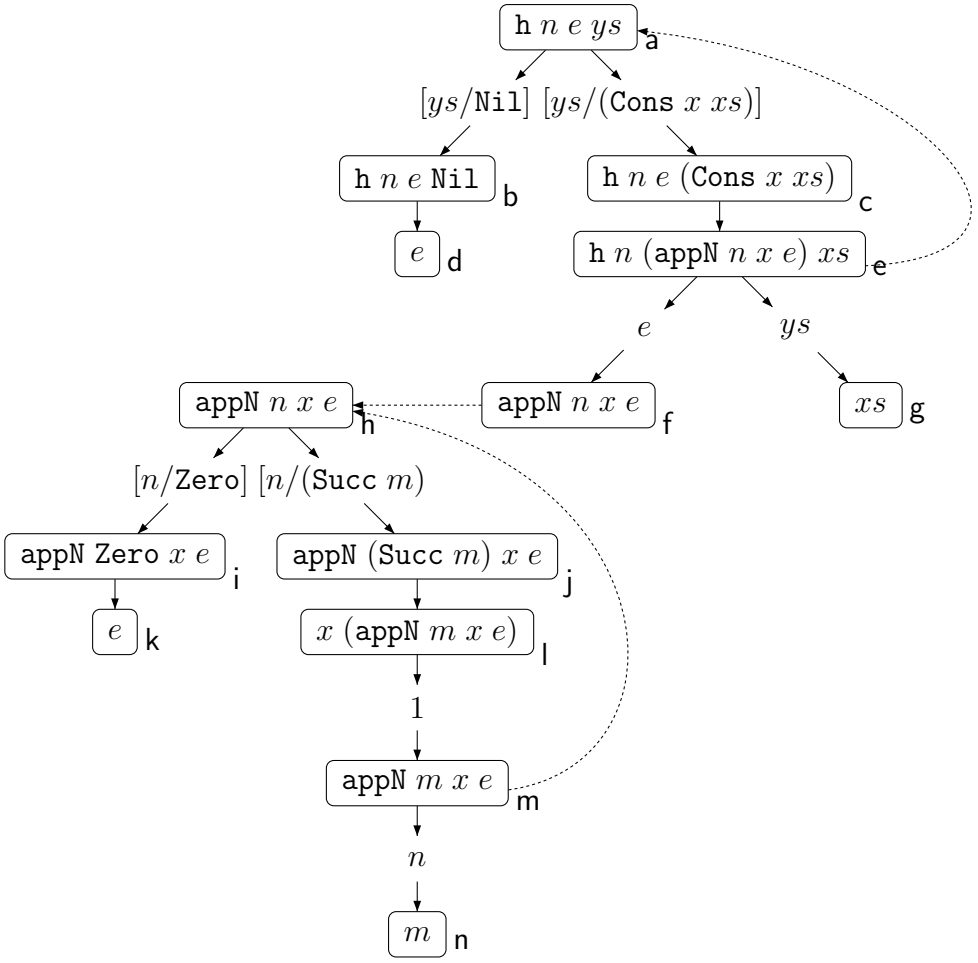

Abbildung 5.8: Ein zyklischer Narrowing-Graph zu Beispiel 5.4 für Startterm $\mathtt{h}\ e\ n\ ys$

Zu der maximalen starken Zusammenhangskomponente $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$ des Narrowing-Graphen in Abbildung 5.8 ergibt sich das folgende DP-Problem:

$$\text{dpProblem}_{\mathsf{NG}}(\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) = (\mathcal{P}_h, \mathcal{R}_h)$$

wobei

$$\mathcal{P}_h := \{\mathtt{h}'n'e^\sharp(\mathtt{Cons}\ x\ xs) \to \mathtt{h}'n'(\mathtt{appN}\ n\ x\ e)^\sharp xs\}$$

und

$$\begin{aligned}\mathcal{R}_\mathtt{h} := \text{rules}_{\mathsf{NG}}(\mathtt{appN}\ n\ x\ e, \mathsf{h}) = \{&\mathtt{appN}'\mathtt{Zero}'x'e &&\to\ e,\\ &\mathtt{appN}'(\mathtt{Succ}\ m)'x'e &&\to\ x'(\mathtt{appN}'m'x'e)\}\end{aligned}$$

Die grundlegende Idee für die Regeln eines DP-Problems ist, daß es alle Regeln beinhaltet, die von den rechten Seiten der Dependency-Pairs her erreicht werden

können. Für $\mathcal{R}_h$ trifft das aber nicht zu, denn die rechte Seite des Dependency-Pairs $\mathtt{h}'n'(\mathtt{appN}\ n\ x\ e)^\sharp xs$ kann bei geeigneter Instantiierung von x, indirekt über die zweite Regel der Funktion `appN`, jede mehrstellige Funktion aufrufen, die typkorrekt in dem Haskellprogramm 5.4 wären. Bei genauerer Betrachtung der rechten Seite des Dependency-Pairs fällt auf, daß der Term $\mathtt{appN}\ n\ x\ e$ zu jedem Ergebnis ausgewertet werden kann, je nachdem, welche Funktion für x eingesetzt wird. Da es unmöglich ist, nur mit Informationen aus dem Narrowing-Graph für alle Instanzen eines solchen Terms die Normalformen abzuschätzen, ersetzen wir diesen Term einfach durch eine frische Variable und haben damit alle Ergebnisse abgedeckt.

Diese Idee auf die Regeln auszudehnen, führt leider nicht zu guten Ergebnissen in der Terminierungsanalyse mit Hilfe der Dependency-Pair-Methode. Würden wir die zweite Regel aus $\mathcal{R}$ durch

$$\mathtt{appN}'(\mathtt{Succ}\ m)'x'e \to y$$

ersetzen, muß die Dependency-Pair-Methode scheitern, da die Regeln eines DP-Problems zu dessen Endlichkeitsnachweis durch eine fundierte Ordnung mindestens als gleich interpretiert werden müssen. Die Variable y kann aber mit jedem Term instantiiert sein, so daß eine Anordnung in jedem Fall scheitern muß. Während in der Menge der Dependency-Pairs durch Argumentfilterungen, die häufig im Rahmen der Dependency-Pair-Methode angewendet werden, eine solche frische Variable möglicherweise weggefiltert werden kann, so daß noch Aussicht auf einen erfolgreichen Terminierungsbeweis besteht.

Um in den Dependency-Pairs solche Funktionsaufrufe, die freie Applikationen der oben beschrieben Art enthalten, durch frische Variablen zu ersetzen, müssen diese Funktionen erst einmal in den Narrowing-Graphen endeckt werden. Eine solche freie Applikation hat die Form $(\underline{cs}, v\ u_1\ \ldots\ u_n)$, wobei $v \in \mathsf{V_L}$ und $n > 0$ sein muß. Ausgehend von Knoten, deren Terme freie Applikationen sind, sammeln wir rückwärts über Auswertungskanten, Fallunterscheidungskanten, Parameteraufteilungskanten und Generalisierungskanten alle Knoten auf und erhalten so die Menge, welche wie folgt definiert ist:

$$\mathrm{freeapps}_{\mathsf{NG}} := \bigcup_{(\underline{cs},v\ u_1\ \ldots\ u_n)\in\mathsf{Q},v\in\mathsf{V_L},n>0,m\in\mathbb{N}} \mathrm{predecessor}^m_{\mathsf{NG}}(\{(\underline{cs}, v\ u_1\ \ldots\ u_n)\})$$

$$\begin{aligned}
&\mathrm{predecessor}_{\mathsf{NG}} : \mathrm{Pot}(\mathsf{Q}) \longrightarrow \mathrm{Pot}(\mathsf{Q})\\
&\mathrm{predecessor}_{\mathsf{NG}}(Q) := \{q' \in \mathsf{Q} \mid q \in Q\ \wedge\ ((q', \sigma, q) \in \mathrm{case}\\
&\qquad\qquad\qquad\qquad \vee\ (q', i, q) \in \mathrm{subterm}\\
&\qquad\qquad\qquad\qquad \vee\ \mathrm{eval}(q') = q\\
&\qquad\qquad\qquad\qquad \vee\ \mathrm{generalisation}(q') = q)\ \}
\end{aligned}$$

Die Wirkung der Menge $\text{freeapps}_{\mathsf{NG}}$ innerhalb der Funktion $\text{calls}_{\mathsf{NG}}$ ist, daß Terme, die zu einer freien Applikation ausgewertet werden können, direkt durch frische Variablen ersetzt werden. Denn die Filtermenge Q, welche eine Teilmenge von $\text{freeapps}_{\mathsf{NG}}$ ist, wird der Funktion $\text{term}_{\mathsf{NG}}$ mit übergeben. Die Funktion $\text{term}_{\mathsf{NG}}$ liefert dann für einen Knoten in $\text{freeapps}_{\mathsf{NG}}$ eine frische Variable zurück, außer, an diesem Knoten wurde eine Variablenexpansion, eine Auswertung oder eine Parameteraufteilung für einen Konstruktor angewendet. Es werden also die oben vorgestellte Art von Dependency-Pairs gebildet. Außerdem werden keine Regeln mehr zu Knoten, die in der Menge $\text{freeapps}_{\mathsf{NG}}$ enthalten sind, gebildet.

Das DP-Problem zu dem Narrowing-Graph in Abbildung 5.8 und dessen maximaler starker Zusammenhangskomponente $\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}$ verändert sich deswegen zu:

$$\text{dpProblem}_{\mathsf{NG}}(\{\mathsf{a}, \mathsf{c}, \mathsf{e}\}) = (\mathcal{P}_h, \mathcal{R}_h)$$

wobei

$$\mathcal{P}_h := \{\mathtt{h}'n'e^\sharp(\mathtt{Cons}\ x\ xs) \to \mathtt{h}'n'w^\sharp xs\}$$

und

$$\mathcal{R}_\mathtt{h} := \emptyset$$

Die frische Variable w ersetzt nun wie gewünscht den Term $\mathtt{appN}'n'x'e$. Die Regelmenge $\mathcal{R}_\mathtt{h}$ ist leer, weil die Knoten a, c, e, f, h, j und l in der Menge $\text{freeapps}_{\mathsf{NG}}$ enthalten sind und so die Regelbildung blockieren.

Ab jetzt können für jeden Narrowing-Graph und dessen maximalen starken Zusammenhangskomponenten die zugehörigen DP-Probleme mit Hilfe der Funktion $\text{dpProblem}_{\mathsf{NG}}$ abgelesen werden.

Die DP-Probleme, welche wie oben beschrieben aus einem Narrowing-Graph abgelesen werden, können nun von bekannten Terminierungsanalyse-Tools (also auch von AProVE [GTSKF04]) untersucht werden. Bei nachgewiesener Endlichkeit all dieser DP-Probleme kann gefolgert werden, daß der Startterm, zu dem der Narrowing-Graph erstellt wurde, NF-terminiert.

5.3 Korrektheit der Terminierungsanalyse

In diesem Abschnitt wird die Korrektheit der Terminierungsanalyse gezeigt. Es wird nachgewiesen, daß ein Starterm für ein Haskellprogramm NF-terminiert, wenn alle aus dem zugehörigen geschlossenen Narrowing-Graph abgelesenen DP-Probleme endlich sind.

Bevor der eigentliche Beweis geführt wird, werden einige noch nötige Begriffe eingeführt und einige Hilfslemmata nachgewiesen.

Dabei gehen wir insgesamt wie folgt vor:

- Als erstes wird die $\overline{\mathsf{NF}}$-Nachfolgerfunktion erklärt, von dieser wird anschließend gezeigt, daß sie die Nicht-NF-Terminierung eines Knotens auf die Nicht-NF-Terminierung eines Nachfolgerknotens überträgt. Mit Hilfe der $\overline{\mathsf{NF}}$-Nachfolgerfunktion können dann die $\overline{\mathsf{NF}}$-Ketten definiert werden. Eine $\overline{\mathsf{NF}}$-Kette ist eine Kette von nicht-NF-terminierenden Knoten innerhalb eines Narrowing-Graphen.

- Daraufhin wird die Auswertungsgleichheit für Haskell-Basisterme erklärt, wobei zwei Haskell-Basisterme auswertungsgleich sind, wenn keine Haskell-Funktion diese unterscheiden kann. Anschließend wird die Auswertungsgleichheit für die abgelesenen Regeln zu einem DP-Problem nachgewiesen. Damit wird bestätigt, daß eine abgelese Regel auf einen Term nur semantikerhaltend angewendet werden kann.

- Es wird die Menge der Regelstartknoten RQ_Z eingeführt, welche alle Knoten enthält, zu denen Regeln für ein DP-Problem der maximalen starken Zusammenhangskomponente Z abgelesen wurden. Mit den Regelstartknoten wird die Menge der Regelkopfterme RH_Z erklärt. Von den Regeln wird gezeigt, daß deren rechten Seiten in der Menge der Regelkopfterme bleiben, so daß später im Lemma 5.11 gezeigt werden kann, daß ein Regelkopfterm mit Hilfe der abgelesenen Regeln jedes Pattern erreichen kann, welches dieser auch mit der normalen Haskellauswertung erreichen kann.

- Die Auswertungsreihenfolge in bezug zu einem Pattern wird erklärt. Es wird nachgewiesen, daß eine Auswertung an einer beliebigen Stelle eines Terms diesen in der Auswertungsreihenfolge nicht zurückwirft. Mit Hilfe dieser Ordnung wird die Induktion im Lemma 5.11 ermöglicht.

- Dann wird die Hauptaussage im Satz 5.1 gezeigt, daß nämlich zu jeder $\overline{\mathsf{NF}}$-Kette eine unendliche Kette des zugehörigen DP-Problems existiert, wobei das Lemma 5.11 benutzt wird, um zu zeigen, daß die rechte Seite eines Dependency-Pairs immer die linke Seite des nachfolgenden Dependency-Pairs innerhalb der unendlichen Kette eines DP-Problems für eine $\overline{\mathsf{NF}}$-Kette erreicht, weil die Teilterme der rechten Seite eines Dependency-Pairs Regelkopfterme sind und so die Regeln des DP-Problems ausreichen, die Patterns zu erreichen, welche die linke Seite des nachfolgenden Dependency-Pairs einfordert.

- Abschließend wird die Korrektheit der Terminierungsanalyse im Satz 5.2 nachgewiesen. Es wird gezeigt, daß aus der Endlichkeit der abgelesenen DP-Probleme die NF-Terminierung des Startterms vom Narrowing-Graph folgt.

Mit Hilfe der $\overline{\mathsf{NF}}$-Nachfolgerfunktion können die unendlichen Zurückführungspfade aufgebaut werden, welche zu Kapitelbeginn intuitiv motiviert wurden.

Definition 5.9 ($\overline{\mathsf{NF}}$-Nachfolgerfunktion) *Die Funktion*

$$\mathrm{succ}_{\mathsf{NG}}^{\neg\mathsf{NF}} :: \mathsf{Q} \times \mathsf{SUB}(\mathsf{D},\mathsf{V}) \longrightarrow \mathsf{Q} \times \mathsf{SUB}(\mathsf{D},\mathsf{V})$$

liefert den normalform-instantiierten nicht NF-terminierenden Nachfolger zu einem normalform-instantiierten nicht NF-terminierenden Knoten aus dem geschlossenen Narrowing-Graph NG *zurück. Sie ist folgendermaßen definiert:*

$$\mathrm{succ}_{\mathsf{NG}}^{\neg\mathsf{NF}}((\underline{cs},u),\sigma) :=$$
$$\begin{cases}
((\underline{cs}',u'),\sigma), & \textit{wenn } \mathsf{M}((\underline{cs},u)) = \underline{\mathrm{eval}},\\
& \quad \mathrm{eval}((\underline{cs},u)) = (\underline{cs}',u'),\\
& \quad u\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \textit{ (Fall 1)}\\
((\underline{cs},u\ x),\sigma'), & \textit{wenn } \mathsf{M}((\underline{cs},u)) = \underline{\mathrm{varexp}},\\
& \quad \mathrm{varexp}((\underline{cs},u)) = (\underline{cs},u\ x),\\
& \quad \exists t \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}}\colon u\ t \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \ \wedge\ x\sigma' = t{\downarrow_{\mathsf{HP}}}\\
& \quad \forall y \in \mathsf{V}_{\mathsf{free}}(u)\colon y\sigma' = y\sigma,\\
& \quad u\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \textit{ (Fall 2)}\\
((\underline{cs}_i,u_i),\sigma), & \textit{wenn } \mathsf{M}((\underline{cs},u)) = \underline{\mathrm{parsplit}},\\
& \quad ((\underline{cs},u),i,(\underline{cs}_i,u_i)) \in \mathrm{subterm},\\
& \quad u_i\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}},\\
& \quad \forall j \in \{1,\ldots,i-1\}\colon u_j\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}},\\
& \quad u\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \textit{ (Fall 3)}\\
((\underline{cs},u\delta_i),\sigma'), & \textit{wenn } \mathsf{M}((\underline{cs},u)) = \underline{\mathrm{case}}\\
& \quad ((\underline{cs},u),\delta_i,(\underline{cs},u\delta_i)) \in \mathrm{case},\\
& \quad u\delta_i\sigma' = u\sigma,\\
& \quad u\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \textit{ (Fall 4)}\\
((\underline{cs}_i,u_i),\sigma), & \textit{wenn } \mathsf{M}((\underline{cs},u)) = \underline{\mathrm{instance}},\\
& \quad ((\underline{cs},u),x_i,(\underline{cs}_i,u_i)) \in \mathrm{subterm},\\
& \quad u_i\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}},\\
& \quad \forall j \in \{1,\ldots,i-1\}\colon u_j\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}},\\
& \quad u\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \textit{ (Fall 5)}\\
((\underline{cs}',u'),\sigma'), & \textit{wenn } \mathsf{M}((\underline{cs},u)) = \underline{\mathrm{instance}},\\
& \quad \mathrm{generalisation}((\underline{cs},u)) = (\underline{cs}',u'),\\
& \quad \forall((\underline{cs},u),x_i,(\underline{cs}_i,u_i)) \in \mathrm{subterm}\colon u_i\sigma \in \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}},\\
& \quad \forall x_i \in \mathsf{V}_{\mathsf{free}}(u)\colon x_i\sigma' = u_i\sigma{\downarrow_{\mathsf{HP}}},\\
& \quad u\sigma \notin \mathsf{NF}_{\mathsf{HP}}^{\mathsf{G}} \textit{ (Fall 6)}
\end{cases}$$

Das Nachfolgerlemma bestätigt uns, daß die aufgebauten unendlichen Zurückführungspfade unserer Idee entsprechen, daß sich aus der Nicht-NF-Terminierung eines Knotens, die Nicht-NF-Terminierung eines Nachfolgers ergibt.

Lemma 5.1 (Nachfolgerlemma) *Sei*

$$\mathsf{NG} = \langle \mathsf{Q}, \mathsf{M}, \text{case}, \text{subterm}, \text{eval}, \text{varexp}, \text{generalisation} \rangle$$

ein geschlossener Narrowing-Graph, so gibt es für jeden Knoten $(\underline{cs}, u) \in \mathsf{Q}$ *und jede Normalform-Substitution* σ*, für die* $u\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt und* $u\sigma$ *typkorrekt ist, einen Nachfolger* $(\underline{cs}', u')$ *von Knoten* $(\underline{cs}, u)$ *und eine Normalform-Substitution* σ'*, für die ebenfalls* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt und* $u\sigma'$ *typkorrekt ist.*

Beweis:
Dieser Beweis wird mit Hilfe der Funktion $\text{succ}^{\neg\mathsf{NF}}_{\mathsf{NG}}$ *konstruktiv geführt.*
Zu zeigen ist nun, wenn $u\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *und* $\text{succ}^{\neg\mathsf{NF}}_{\mathsf{NG}}((\underline{cs}, u), \sigma) = ((\underline{cs}', u'), \sigma')$ *gelten,* σ *eine Normalform-Substitution ist und* $u\sigma$ *typkorrekt ist, daß* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt und* σ' *eine Normalform-Substitution, so daß* $u'\sigma'$ *typkorrekt ist.*

Sei nun σ *eine Normalform-Substitution so daß* $u\sigma$ *typkorrekt ist,*

$$\text{succ}^{\neg\mathsf{NF}}_{\mathsf{NG}}((\underline{cs}, u), \sigma) = ((\underline{cs}', u'), \sigma')$$

und $u\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$.

Fall 1 *Aus der Definition der Auswertung folgt* $u \rightarrow^{\mathsf{EXT}}_{\mathsf{HP}} u'$ *und damit folgt aus dem Korrektheitslemma 4.1, daß* $u'\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt und* $u'\sigma$ *typkorrekt ist. Wegen* $\sigma' = \sigma$ *gilt so auch* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$

Fall 2 *Aus der Definition 4.5 und* $u\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *folgt, daß ein Term* $t \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *existiert mit* $u\sigma\ t \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*, so daß* $u\sigma\ t$ *typkorrekt ist. Wegen der Eindeutigkeit der Haskellauswertung, gilt so auch* $u\sigma\ (t{\downarrow_{\mathsf{HP}}}) \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*. Aus* $x\sigma' = t{\downarrow_{\mathsf{HP}}}$ *und* $\forall y \in \mathsf{V}_{\mathsf{free}}(u)\colon y\sigma' = y\sigma$ *folgt, daß* $u\ x\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt und* σ' *eine Normalform-Substitution ist.*

Fall 3 *Durch Korrektheitslemma 4.2 folgt, daß ein*

$$((\underline{cs}, u), i, (\underline{cs}_i, u_i)) \in \text{subterm}$$

existiert, für das $u_i\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt. Wegen* $\sigma' = \sigma$ *und* $u' = u_i$ *gilt so auch* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$.

Fall 4 *Die Fallunterscheidung gibt es für Variablen und Typvariablen, daher sind hier zwei Fälle zu betrachten.*

Fall 4.1 *Sei auf* $(\underline{cs}, u)$ *eine Fallunterscheidung für eine Variable angewendet worden. Aus der Definition der Fallunterscheidung für Variablen folgt* $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u) = \underline{\text{error}}\ x_{|d\ \tau_1\ \ldots\ \tau_m|}$ *und für jeden Konstruktor* c_i *aus* $\{c_1, \ldots, c_n\} = \text{constr}_{\mathsf{D}}(d)$*, daß* $((\underline{cs}, u), \delta_i, (\underline{cs}, u\delta_i)) \in \text{case}$ *mit* $\delta_i = [x_{|d\ \tau_1\ \ldots\ \tau_m|}/(c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|}]$ *gilt.*

Wegen $u\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *und weil* σ *eine Normalform-Substitution ist, existieren für* $x_{|d\ \tau_1\ \ldots\ \tau_m|}$ *die Terme* $t_1, \ldots, t_{n_i} \in \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$*, so daß*

$$x_{|d\ \tau_1\ \ldots\ \tau_m|}\sigma = (c_i\ x_{i,1}\ \ldots\ x_{i,n_i})_{|d\ \tau_1\ \ldots\ \tau_m|}[x_{i,1}/t_1, \ldots, x_{i,n_i}/t_{n_i}]$$

für ein $i \in \{1, \ldots, n\}$ *gilt.*

$x_{|d\ \tau_1\ \ldots\ \tau_m|}\sigma = \underline{\text{error}}$ *kann nicht gelten, da sonst* $u\sigma \rightarrow_{\mathsf{HP}} \underline{\text{error}}$ *und* $u\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gelten würden und so der Annahme* $u\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *widersprechen würde.*

Insgesamt folgt, daß

$$\sigma' = [x_{i,1}/t_1, \ldots, x_{i,n_i}/t_{n_i}]\sigma$$

eine Normalform-Substitution ist und $u\delta_i\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *gilt. Mit* $u' = u\delta_i$ *folgt so auch* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*, und daß* $u'\sigma'$ *typkorrekt ist.*

Fall 4.2 *Sei nun auf* $(\underline{cs}, u)$ *eine Fallunterscheidung für eine Typvariable angewendet worden. Da* σ *eine Normalform-Substitution und* $u\sigma$ *typkorrekt ist, ist jeder Typ ausreichend instantiiert (der Typchecker würde andernfalls den Term zurückweisen). Aus der Definition der Fallunterscheidung für Typvariablen folgt, daß es ein*

$$((\underline{cs}, u), i, (\underline{cs_i}, u_i)) \in \text{subterm}$$

und eine Normalform-Substitution σ' *geben muß, für die* $u_i\sigma' = u\sigma$ *gelten muß, weil in der Fallunterscheidung alle gültigen Typinstanzen abgedeckt sind.*

Fall 5 *Aus der Bedingung* $u_i\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *für diesen Fall und wegen* $u' = u_i$ *und* $\sigma' = \sigma$ *folgt direkt* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*.*

Fall 6 *Aus der Definition der Instantiierung folgt* $\mu = [x_1/u_1, \ldots, x_n/u_n]$ *und* $u = u'\mu$*. Zusammen mit* $u\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *folgt deswegen* $u'\mu\sigma \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*. Für alle* $i \in \{1, \ldots, n\}$ *gilt durch die Bedingungen für diesen Fall* $u_i\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *und so auch* $x\mu\sigma \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *für alle* $x \in \mathsf{V}_{\mathsf{free}}(u')$*. Wegen der Eindeutigkeit der Haskellauswertung folgt so auch* $u'\sigma' \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$*, und daß* $u'\sigma'$ *typkorrekt ist, weil* $x\sigma' = x\mu\sigma{\downarrow_{\mathsf{HP}}}$ *gilt.*

□

Die $\overline{\mathsf{NF}}$-Kette entspricht einem unendlichen Zurückführungspfad ab einem Knoten, in den eine Generalisierungskante zeigt, denn durch das Zykellemma 4.8 auf Seite 82 wissen wir bereits, daß Zyklen auf denen unendliche Zurückführungspfade nur liegen können, eine Generalisierungskante enthalten.

Definition 5.10 ($\overline{\mathsf{NF}}$-Kette) *Eine unendliche Folge* $(q_1, \sigma_1), (q_2, \sigma_2), (q_3, \sigma_3), \ldots$ *aus Paaren von Knoten und Substitutionen ist eine* $\overline{\mathsf{NF}}$*-Kette ab Knoten* q_1*, wenn* $q_1 = (\underline{cs}_1, u_1)$*,* $q_1 \in \text{range(generalisation)}$*,* $u_1 \notin \mathsf{NF}_{\mathsf{HP}}$ *und* σ_1 *eine Normalform-Substitution mit* $u_1\sigma_1 \notin \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$ *ist, so daß*

$$\text{succ}^{\neg\mathsf{NF}}_{\mathsf{NG}}(q_i, \sigma_i) = (q_{i+1}, \sigma_{i+1})$$

für alle $i > 0$ *gilt.*

Die booleschen Funktionen ermöglichen eine einfache Charakterisierung der Auswertungsgleichheit von zwei Haskell-Basistermen

Definition 5.11 (Boolesche Funktionen zu einem Typ) *Die Menge* $\mathsf{B}_{|\tau|} \subseteq \mathsf{V}_{\mathsf{F}}$ *ist die Menge aller Funktionsvariablen, die jeweils an eine Funktion mit dem Typschema* $\emptyset \Rightarrow \tau \to \texttt{Bool}$ *gebunden sind. Sie ist definiert wie folgt:*

$$\mathsf{B}_{|\tau|} := \{v_{|\tau\to\texttt{Bool}|} \mid (v_{|\tau\to\texttt{Bool}|}, \emptyset \Rightarrow \tau \to \texttt{Bool}, \underline{fs}) \in \mathsf{F}(\mathsf{D}, \mathsf{V})\}$$

Die Auswertungsgleichheit sagt aus, daß zwei Terme innerhalb eines Haskellprogamms nicht durch Patternmatching unterschieden werden können. Zwei auswertungsgleiche Terme unterscheiden sich also höchstens in der Anzahl der Auswertungsschritte, die sie benötigen, um ein bestimmtes Pattern zu erreichen, da die Schrittanzahl mit keiner Funktion in Haskell gezählt werden kann.

Definition 5.12 (Auswertungsgleichheit) *Zwei Terme* $t_{|\tau|} \in \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ *und* $u_{|\tau|} \in \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ *sind genau dann auswertungsgleich(kurz* $t_{|\tau|} \equiv_{\mathsf{HP}} u_{|\tau|}$*), wenn*

$$(v_{|\tau\to\texttt{Bool}|}u_{|\tau|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}} \wedge v_{|\tau\to\texttt{Bool}|}t_{|\tau|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}) \Rightarrow (v_{|\tau\to\texttt{Bool}|}u_{|\tau|}{\downarrow_{\mathsf{HP}}} = v_{|\tau\to\texttt{Bool}|}t_{|\tau|}{\downarrow_{\mathsf{HP}}})$$

und

$$v_{|\tau\to\texttt{Bool}|}u_{|\tau|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}} \Leftrightarrow v_{|\tau\to\texttt{Bool}|}t_{|\tau|} \in \mathsf{NF}^{\mathsf{G}}_{\mathsf{HP}}$$

für alle $v_{|\tau\to\texttt{Bool}|} \in \mathsf{B}_{|\tau|}$ *gilt. Zusätzlich sind zwei Terme* $t, u \in \mathsf{H}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ *genau dann auswertungsgleich, wenn für alle Grundsubstitutionen* σ

$$t\sigma \equiv_{\mathsf{HP}} u\sigma$$

gilt.

Mit der Patternerreichbarkeit wird ausgesagt, daß ein Term zu einem Term ausgewertet werden kann, der ein gegebenes Pattern matcht.

Definition 5.13 (Patternerreichbarkeit) *Sei* $t, u \in \mathsf{H_B(D,V)}$ *und* p *ein lineares Pattern aus* $\mathsf{P_B(D,V)}$ *mit* $t \to^*_{\mathsf{HP}} u$. *So gilt*

$$t \to^{>p}_{\mathsf{HP}} u$$

genau dann, wenn p *den Term* u *matcht und dieser gleichzeitig der erste Term in der Reduktion* $t \to^*_{\mathsf{HP}} u$ *ist, der von* p *gematcht wird.*

Außerdem ist $\mathrm{len}(t \to^{>p}_{\mathsf{HP}})$ *die Anzahl der benötigten Auswertungsschritte, um vom Term* t *aus das Pattern* p *zu erreichen. Wenn* t *das Pattern* p *nicht erreicht, gilt* $\mathrm{len}(t \to^{>p}_{\mathsf{HP}}) = \infty$.

Wenn ein Pattern durch einen Term erreichbar ist, so kann auch jeder zu diesem auswertungsgleichen Term dieses Pattern erreichen.

Lemma 5.2 (Patternlemma)
Wenn $t \equiv_{\mathsf{HP}} t'$, $t \to^{>p}_{\mathsf{HP}} u$ *und* p *eine lineares Pattern aus* $\mathsf{P_B(D,V)}$ *ist, dann folgt, daß ein* u' *existiert mit* $t' \to^{>p}_{\mathsf{HP}} u'$ *und* $u \equiv_{\mathsf{HP}} u'$.

Beweis:
Aus $t \equiv_{\mathsf{HP}} t'$ *und* $t \to^{>p}_{\mathsf{HP}} u$ *folgt* $t' \equiv_{\mathsf{HP}} t \equiv_{\mathsf{HP}} u$. *Sei* τ *der Typ von* t. *Außerdem gibt es ein boolesche Funktion, welche an* $v \in \mathsf{B}_{|\tau|}$ *gebunden ist, mit den Regeln*

$$v\ p = \texttt{True}$$
$$v\ \_ = \texttt{False}$$

so daß $v\ p\sigma{\downarrow_{\mathsf{HP}}} = \texttt{True}$ *für jede Substitution* σ *gilt. Also gilt* $v\ u{\downarrow_{\mathsf{HP}}} = \texttt{True}$ *und so auch* $v\ t'{\downarrow_{\mathsf{HP}}} = \texttt{True}$, *wegen* $t' \equiv_{\mathsf{HP}} u$. *So muß* t' *das Pattern* p *erreichen, also gibt es* u'. □

Das folgende Lemma wird benötigt, um später zeigen zu können, daß auch Regeln wie sie zu DP-Problemen aus Narrowing-Graphen abgelesen werden, die Auswertungsgleichheit erhalten.

Lemma 5.3 ($\mathrm{term}_{\mathsf{NG}}$ erhält Auswertungsgleichheit)
Wenn $(t, R) = \mathrm{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset)$ *gilt, folgt* $t \equiv_{\mathsf{HP}} u$

Beweis:
Diese Aussage wird per Induktion über die Aufrufe von $\mathrm{term}_{\mathsf{NG}}$ *gezeigt, da* $\mathrm{term}_{\mathsf{NG}}$ *terminiert.*

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\mathrm{eval}}$. *So gilt* $\mathrm{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = \mathrm{term}_{\mathsf{NG}}((\underline{cs}', u'), \emptyset)$, $(\underline{cs}', u') = \mathrm{eval}(\underline{cs}, u)$ *und* $u \to^{\mathsf{EXT}}_{\mathsf{HP}} u'$. *Wegen* $(t, R) = \mathrm{term}_{\mathsf{NG}}((\underline{cs}', u'), \emptyset)$ *folgt per Induktionshypothese* $t \equiv_{\mathsf{HP}} u'$ *und so auch* $t \equiv_{\mathsf{HP}} u$, *weil* $\to^{\mathsf{EXT}}_{\mathsf{HP}}$ *die Auswertungsgleichheit*

per Definition erhält.

Für $\mathsf{M}((\underline{cs}, u)) \in \{\underline{\text{varexp}}, \underline{\text{case}}\}$ *gilt die Aussage direkt, weil dabei* $u = t$ *gilt.*

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\text{parsplit}}$. *So gilt* $\text{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = (h\ t_1\ \ldots\ t_n, \bigcup_{i=1}^{n} R_i)$ *für* $u = h\ u_1\ \ldots\ u_n$ *und* $(t_i, R_i) = \text{term}_{\mathsf{NG}}((\underline{cs}_i, u_i), \emptyset)$. *Wegen* $t_i \equiv_{\mathsf{HP}} u_i$ *gilt sofort* $h\ t_1\ \ldots\ t_n \equiv_{\mathsf{HP}} h\ u_1\ \ldots\ u_n$.

Fall $\mathsf{M}((\underline{cs}, u)) = \underline{\text{instance}}$ *ist analog zu Fall* $\mathsf{M}((\underline{cs}, u)) = \underline{\text{parsplit}}$, *da Haskellprogramme Konstruktorsysteme sind.*

□

Die Auswertungsgleichheit der abgelesenen Regeln garantiert uns, daß eine abgelesene Regel durch eine Anwendung einen Term in einen anderen überführt, der dieselbe Semantik in Haskell hat.

Lemma 5.4 (Regeln aus $\text{rules}_{\mathsf{NG}}$ **erhalten Auswertungsgleichheit)**
Wenn $t \equiv_{\mathsf{HP}} u$ *gilt, folgt* $l \equiv_{\mathsf{HP}} r$ *für alle* $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u))$

Beweis:
Diese Aussage wird per Induktion über die Aufrufe von $\text{rules}_{\mathsf{NG}}$ *gezeigt, da* $\text{rules}_{\mathsf{NG}}$ *terminiert.*

Sei $t \equiv_{\mathsf{HP}} u$.

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\text{eval}}$.
Es gilt $\text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u)) = \text{rules}_{\mathsf{NG}}(t, (\underline{cs}', u'))$, $(\underline{cs}', u') = \text{eval}(\underline{cs}, u)$ *und* $u \to_{\mathsf{HP}}^{\mathsf{EXT}} u'$. *So folgt* $u \equiv_{\mathsf{HP}} u'$ *und damit auch* $t \equiv_{\mathsf{HP}} u'$, *wegen* $t \equiv_{\mathsf{HP}} u$. *Per Induktionshypothese gilt* $l \equiv_{\mathsf{HP}} r$ *für alle* $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t, (\underline{cs}', u'))$ *und so auch* $l \equiv_{\mathsf{HP}} r$ *für alle* $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u))$.

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\text{case}}$. *So gilt*

$$\text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u)) = \bigcup_{((\underline{cs}, u), \delta, (\underline{cs}', u\delta)) \in \text{case}} \text{rules}_{\mathsf{NG}}(t\delta, (\underline{cs}', u\delta))$$

Aus $t \equiv_{\mathsf{HP}} u$ *folgt* $t\delta \equiv_{\mathsf{HP}} u\delta$ *per Definition von* $\equiv_{\mathsf{HP}}$ *und so ist die Induktionshypothese wieder anwendbar und das Geforderte folgt.*

Für $\mathsf{M}((\underline{cs}, u)) = \underline{\text{varexp}}$ *folgt* $\text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u)) = \{[\![t]\!]_{\mathsf{APP}} \to [\![u]\!]_{\mathsf{APP}}\}$ *und so wegen* $t \equiv_{\mathsf{HP}} u$ *direkt auch das Gewünschte.*

Für $\mathsf{M}((\underline{cs}, u)) \in \{\underline{\text{instance}}, \underline{\text{parsplit}}\}$ *folgt*

$$\text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u)) = \{[\![t]\!]_{\mathsf{APP}} \to [\![w]\!]_{\mathsf{APP}}\} \cup \bigcup_{(\underline{cs}', u') \in R} \text{rules}_{\mathsf{NG}}(u', (\underline{cs}', u'))$$

für $(w, R) = \text{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset)$*. Per Induktionshypothese folgt für jede Regel aus* $\text{rules}_{\mathsf{NG}}(u', (\underline{cs}', u'))$ *bereits das Gewünschte. Wegen Lemma 5.3 folgt* $w \equiv_{\mathsf{HP}} u$ *und mit* $t \equiv_{\mathsf{HP}} u$ *folgt so* $t \equiv_{\mathsf{HP}} w$*.*

□

Diese Definition wird benötigt, um die Beweise auf Regeln einfacher gestalten zu können.

Definition 5.14 (*R* ist $\mathcal{R}$ auf Haskelltermen) *Für jedes* $\mathcal{R}$ *aus einem DP-Problem* $(\mathcal{P}, \mathcal{R})$*, welches mit Hilfe der Funktion* $\text{dpProblem}_{\mathsf{NG}}$ *aus einem geschlossen Narrowing-Graph abgelesen worden ist, existiert das Termersetzungssystem* R *auf Haskelltermen, so daß* $l \to r \in R$ *für jedes* $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \mathcal{R}$ *gilt (i.e. in der Funktion* $\text{rules}_{\mathsf{NG}}$ *können die* APP*-Aufrufe einfach weggelassen werden und man erhält* R*).*

Definition 5.15 (Regelstartknoten RQ_Z) *Die Menge* $\mathsf{RQ}_Z \subseteq \mathsf{Q}$ *ist die Menge aller Knoten, zu denen* $\text{rules}_{\mathsf{NG}}$ *Regeln für eine maximale starke Zusammenhangskomponente* Z *abgelesen hat.*

$$\mathsf{RQ}_Z := \bigcup_{q \in Z} \text{rnodes}^{\mathsf{DP}}_{\mathsf{NG}}(q, Z)$$

wobei:

$$\text{rnodes}^{\mathsf{DP}}_{\mathsf{NG}}((\underline{cs}, u), Q) := \bigcup_{q \in \text{neededNodes}_{\mathsf{NG}}(\text{calls}_{\mathsf{NG}}(u, (\underline{cs}, u), Q))} \text{rnodes}_{\mathsf{NG}}(q)$$

$\text{rnodes}_{\mathsf{NG}} : \mathsf{H}_{\mathsf{B}}(\mathsf{D}, \mathsf{V}) \times \mathsf{Q} \longrightarrow \text{Pot}(\mathsf{Q})$,

$$\text{rnodes}_{\mathsf{NG}}(q) := \begin{cases} \text{rnodes}_{\mathsf{NG}}(\text{eval}(q)), & \text{für } \mathsf{M}(q) = \underline{\text{eval}} \\ \emptyset, & \text{für } \mathsf{M}(q) = \underline{\text{varexp}} \\ \bigcup\limits_{(q, \delta_i, q_i) \in \text{case}} \text{rnodes}_{\mathsf{NG}}(q_i), & \text{für } \mathsf{M}(q) = \underline{\text{case}} \\ W \cup \bigcup\limits_{(q \in W)} \text{rnodes}_{\mathsf{NG}}(q), & \\ & \text{für } \mathsf{M}(q) \in \{\underline{\text{instance}}, \underline{\text{parsplit}}\}, \\ & (t, W) = \text{term}_{\mathsf{NG}}(q, \emptyset) \end{cases}$$

Es ist zu beachten, daß die Funktionen $\text{rnodes}_{\mathsf{NG}}$ *und* $\text{rnodes}^{\mathsf{DP}}_{\mathsf{NG}}$ *dieselbe Rekursionsstruktur aufweisen wie die Funktionen* $\text{rules}_{\mathsf{NG}}$ *und* $\text{neededRules}_{\mathsf{NG}}$*.*

Lemma 5.5 (Marken von RQ_Z)
Wenn $q \in \mathsf{RQ}_Z$ *gilt* $\mathsf{M}(q) \in \{\underline{\text{eval}}, \underline{\text{case}}, \underline{\text{varexp}}\}$

Beweis:
Per Definition von $\text{rnodes}_{\mathsf{NG}}$ *und* $\text{rnodes}^{\mathsf{DP}}_{\mathsf{NG}}$*, sind alle Knoten aus* RQ_Z *durch* $\text{term}_{\mathsf{NG}}$ *aufgenommen worden oder direkt durch* $\text{neededNodes}_{\mathsf{NG}}$ *in* RQ_Z *aufgenommen worden. Durch* $\text{term}_{\mathsf{NG}}$ *können nur Knoten mit der Marke* $\underline{\text{case}}$ *aufgenommen werden.*

Ansonsten sind es bei $\text{neededNodes}_{\mathsf{NG}}$ *nur Knoten, in die Generalisierungskanten zeigen, diese können nur die Marken* $\underline{\text{eval}}$, $\underline{\text{case}}$ *haben, außer es ist der Knoten* $(\emptyset, x\ y)$*. Der Knoten* $(\emptyset, x\ y)$ *ist aber eine freie Applikation und wird deswegen direkt durch* $\text{freeapps}_{\mathsf{NG}}$ *wieder entfernt.* □

Definition 5.16 (Baum-Nachfolger) *Der Knoten q ist ein Baum-Nachfolger von einem Knoten* q' *in einem Narrowing-Graph, wenn auf dem Pfad von* q' *nach q keine Generalisierungskanten oder Variablenexpansionskanten überschritten werden.*

Mit diesem Lemma wird gezeigt, daß zu dem Generalisierungskanten-Nachfolger eines Knotens, an dem eine Regel endet, ebenfalls Regeln abgelesen wurden. Diese Aussage brauchen wir, um später Lemma 5.8 nachweisen zu können.

Lemma 5.6 (Instantiierungen und RQ_Z)
Wenn $n \in \mathsf{RQ}_Z$ *und sei q ein Baum-Nachfolger von n mit* $\mathsf{M}(q) = \underline{\text{instance}}$*, so gilt* $\text{generalisation}(q) \in \mathsf{RQ}_Z$

Beweis:
Die Menge RQ_Z *ist abhängig von* $\text{neededNodes}_{\mathsf{NG}}$*, welche die Nachfolger anhand der Ergebnisse von* $\text{calls}_{\mathsf{NG}}$ *zusammensetzt. Für ein Ergebnis*

$$(t, q', (t, W)) \in \text{calls}_{\mathsf{NG}}(\dots)$$

gilt $(t, W) = terms(q', X)$ *für eine Menge X. Die Funktion* $\text{neededNodes}_{\mathsf{NG}}$ *nimmt W direkt in die Ergebnismenge auf, während zu q die Nachfolger gebildet werden. Die Nachfolger werden über alle Kanten bis auf Variablenexpansionskanten abgelesen, während* $\text{term}_{\mathsf{NG}}$ *und* $\text{rules}_{\mathsf{NG}}$ *jeweils nie über Generalisierungskanten und nie über Variablenexpansionskanten hinweg gehen. Und so muß die Menge der Nachfolger von* q' *immer größer sein, als die der von* $\text{term}_{\mathsf{NG}}$ *oder* $\text{rules}_{\mathsf{NG}}$ *durchlaufenen Knoten. Und zu allen Nachfolgern von* q'*, in die eine Generalisierungskante zeigt, werden Regeln gebildet und so folgt* $\text{generalisation}(q') \in \mathsf{RQ}_Z$*, außer* $\text{freeapps}_{\mathsf{NG}}$ *entfernt* $\text{generalisation}(q')$ *wieder. Das kann* $\text{freeapps}_{\mathsf{NG}}$ *nicht,*

weil folgender Sachverhalt für $\text{freeapps}_{\mathsf{NG}}$ *gilt:*

Wenn $q' \in \text{freeapps}_{\mathsf{NG}}$ *und* $q' \in \text{range(generalisation)}$ *gilt,*

$$generalisation^{-1}(q') \in \text{freeapps}_{\mathsf{NG}}$$

Deswegen ist nach dem Herausnehmen von $\text{freeapps}_{\mathsf{NG}}$ *aus den bis jetzt für* q' *aufgesammelten Knoten immer noch* $\text{generalisation}(q)$ *enthalten oder* q *wurde mit entfernt.* □

Die Menge RH_Z ist die Menge der Haskellterme $\mathsf{H_B(D,V)}$, die mit Hilfe von Regeln aus R zu jedem Pattern ausgewertet werden, welches von diesen Termen genauso mit der erweiterten Auswertungsfunktion $\text{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}$ erreicht werden kann.

Definition 5.17 (Regelkopfterme RH_Z zu RQ_Z) *Die Menge der Regelkopfterme* $\mathsf{RH}_Z \subseteq \mathsf{H_B(D,V)}$ *ist die kleinste Menge, für die*

- $\mathsf{V_L} \subseteq \mathsf{RH}_Z$,
- $\{t_{|\tau\to\tau'|} | t_{|\tau\to\tau'|} \in \mathsf{H^G_B(D,V)}\} \subseteq \mathsf{RH}_Z$,
- $u[x_1/t_1 \ \ldots \ x_n/t_n] \in \mathsf{RH}_Z$, *für* $(\underline{cs}, u) \in \mathsf{RQ}_Z$, $t_i \in \mathsf{RH}_Z$ *und* $x_i \in \mathsf{V_L}(u)$,
- $c\ t_1 \ \ldots \ t_n \in \mathsf{RH}_Z$, *für* $c \in \underline{\mathsf{Cons_D}}$, $n = arity(c)$ *und* $t_i \in \mathsf{RH}_Z$ *und*
- $\underline{\text{error}} \in \mathsf{RH}_Z$

gilt. Eine Substitution σ *ist eine* $\mathsf{RH}_Z \cap \mathsf{H^G_B(D,V)}$*-Substitution, wenn für jedes* $x \in Dom(\sigma)$ *der Term* $x\sigma$ *aus* $\mathsf{RH}_Z \cap \mathsf{H^G_B(D,V)}$ *ist.*

Sei beispielsweise $\mathsf{RQ}_Z = \{(\emptyset, \mathtt{f}\ x\ (\mathtt{g}\ x), (\emptyset, \mathtt{h}\ y)\}$ mit $\mathtt{f}, \mathtt{g} \in \mathsf{V_F}$ und $x, y \in \mathsf{V_L}$. Es sind also nur Regeln zu den Knoten $(\emptyset, \mathtt{f}\ x\ (\mathtt{g}\ x)$ und $(\emptyset, \mathtt{h}\ x)$ vorhanden, so daß die Menge RH_Z neben anderen die Terme

- $\mathtt{h}\ x$
- $\mathtt{f}\ x\ (\mathtt{g}\ x)$
- $\mathtt{f}\ (\mathtt{h}\ y)\ (\mathtt{g}\ (\mathtt{h}\ y))$
- $\mathtt{h}\ (\mathtt{f}\ x\ (\mathtt{g}\ x))$
- $\mathtt{f}\ (\mathtt{f}\ y\ (\mathtt{g}\ y))\ (\mathtt{g}\ (\mathtt{f}\ x\ (\mathtt{g}\ x)))$
- ...

enthält, aber beispielsweise diese Terme *nicht* enthält:

- $\mathtt{f}\ x\ y$
- $\mathtt{f}\ (\mathtt{h}\ y)\ (\mathtt{g}\ x)$
- $\mathtt{g}\ x$
- ...

Es gilt $\mathtt{f}\ x\ y \notin \mathsf{RH}_Z$, weil $\mathtt{f} \in \mathsf{V_F}$ gilt und es keinen Knoten der Form $(\underline{cs'}, \mathtt{f}\ x\ y)$ in RQ_Z gibt. Genauso gibt es keine passenden Knoten für die beiden letzten Terme.

Im folgenden Lemma wird gezeigt, daß die Funktion $\mathrm{term}_{\mathsf{NG}}$ bei richtiger Verwendung wieder Terme aus RH_Z zurückliefert. Damit können wir im anschließenden Lemma zeigen, daß Regeln aus R die Menge der RH_Z nicht verlassen.

Lemma 5.7 ($\mathrm{term}_{\mathsf{NG}}$ liefert Terme aus RH_Z) *Sei $(t, W) = \mathrm{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset)$ und $(\underline{cs}, u)$ ein Baum-Nachfolger eines Knotens aus RQ_Z. Wenn $W \subseteq \mathsf{RQ}_Z$ gilt, ist $t \in \mathsf{RH}_Z$.*

Beweis:
Diese Aussage wird per Induktion über die Aufrufe von $\mathrm{term}_{\mathsf{NG}}$ gezeigt, da $\mathrm{term}_{\mathsf{NG}}$ terminiert.

Sei $W \subseteq \mathsf{RQ}_Z$.

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\mathrm{eval}}$. So gilt $\mathrm{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = \mathrm{term}_{\mathsf{NG}}((\underline{cs'}, u'), \emptyset)$, und per Induktionshypothese folgt das Gewünschte.

Für $\mathsf{M}((\underline{cs}, u)) = \underline{\mathrm{varexp}}$ ist $t = u$ und u ist vom Funktionstyp und es gilt $u \in \mathsf{RH}_Z$.

Für $\mathsf{M}((\underline{cs}, u)) = \underline{\mathrm{case}}$ gilt $(\underline{cs}, u) \in W$. Damit gilt $(\underline{cs}, u) \in \mathsf{RQ}_Z$ und so folgt $u \in \mathsf{RH}_Z$.

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\mathrm{parsplit}}$, so gilt

$$\mathrm{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = (h\ t_1\ \ldots\ t_n, \bigcup_{i=1}^{n} R_i)$$

für $u = h\ u_1\ \ldots\ u_n$ und $(t_i, R_i) = \mathrm{term}_{\mathsf{NG}}((\underline{cs_i}, u_i), \emptyset)$. So gilt $R_i \subseteq W \subseteq \mathsf{RQ}_Z$ und es folgt $t_i \in \mathsf{RH}_Z$ per Induktionshypothese. Wegen des Aufbaus von RH_Z gilt $h\ t_1\ \ldots\ t_n \in \mathsf{RH}_Z$.

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\text{instance}}$, *so gilt*

$$\text{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = (u'[x_1/t_1, \ldots, x_n/t_n], \bigcup_{i=1}^{n} R_i)$$

für $(\underline{cs}', u') = \text{generalisation}((\underline{cs}, u))$ *und* $(t_i, R_i) = \text{term}_{\mathsf{NG}}((\underline{cs}_i, u_i), \emptyset)$. *So gilt* $R_i \subseteq W \subseteq \mathsf{RQ}_Z$ *und es folgt* $t_i \in \mathsf{RH}_Z$ *per Induktionshypothese. Wegen Lemma 5.6 gilt* $(\underline{cs}', u') \in \mathsf{RQ}_Z$ *und zusammen ergibt sich* $u'[x_1/t_1, \ldots, x_n/t_n] \in \mathsf{RH}_Z$ *aus dem Aufbau von* RH_Z. □

Das folgende Lemma weist nach, daß eine Regel, welche auf einen Term aus RH_Z angewendet wird, wieder einen Term liefert, der in RH_Z liegt.

Lemma 5.8 (Regeln bleiben in RH_Z)
Wenn $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u))$ *und* $(\underline{cs}, u) \in \mathsf{RQ}_Z$, *dann gilt* $r \in \mathsf{RH}_Z$.

Beweis:
Wegen $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t, (\underline{cs}, u))$ *und der Definition von* $\text{rules}_{\mathsf{NG}}$, *muß es einen Knoten* $(\underline{cs}', u')$ *mit* $\mathsf{M}((\underline{cs}', u')) \in \{\underline{\text{varexp}}, \underline{\text{instance}}, \underline{\text{parsplit}}\}$ *und einen Term* t' *geben, so daß* $[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t', (\underline{cs}', u'))$ *gilt.*

Für den Fall, daß

$$[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t', (\underline{cs}', u'))$$

mit $\mathsf{M}((\underline{cs}', u')) = \underline{\text{varexp}}$ *gilt, hat* r *auf jeden Fall den Funktionstyp und ist so in* RH_Z *enthalten.*

Für den Fall, daß

$$[\![l]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}} \in \text{rules}_{\mathsf{NG}}(t', (\underline{cs}', u'))$$

mit $\mathsf{M}((\underline{cs}', u')) \in \{\underline{\text{instance}}, \underline{\text{parsplit}}\}$ *gilt, ist* $(r, W) = \text{term}_{\mathsf{NG}}((\underline{cs}', u'), \emptyset)$. *Weil die Rekursionsstrukturen von den Funktionen* $\text{rules}_{\mathsf{NG}}$ *und* $\text{neededRules}_{\mathsf{NG}}$ *und den Funktionen* $\text{rnodes}_{\mathsf{NG}}$ *und* $\text{rnodes}^{\mathsf{DP}}_{\mathsf{NG}}$ *gleich sind und* $\text{rnodes}^{\mathsf{DP}}_{\mathsf{NG}}$ *und* $\text{neededRules}_{\mathsf{NG}}$ *für dieselben Knoten aufgerufen werden, gilt*

$$\mathsf{RQ}_Z \supseteq \text{rnodes}_{\mathsf{NG}}((\underline{cs}, u)) \supseteq \text{rnodes}_{\mathsf{NG}}((\underline{cs}', u')) \supseteq W$$

für $(r, W) = \text{term}_{\mathsf{NG}}((\underline{cs}', u'), \emptyset)$. *Weil* $(\underline{cs}', u')$ *Baum-Nachfolger von* $(\underline{cs}, u) \in \mathsf{RQ}_Z$ *und* $\mathsf{RQ}_Z \supseteq W$ *gilt, folgt* $r \in \mathsf{RH}_Z$ *durch das Lemma 5.7.* □

Die Auswertungsreihenfolge sagt aus, daß ein Term mehr Auswertungsschritte braucht als ein anderer Term um dasselbe Pattern zu erreichen. Diese Ordnung ist fundiert für Terme, die dasselbe Pattern in endlicher Anzahl von Haskell-Auswertungsschritten erreichen können. Anhand dieser Ordnung wird per Induktion in Lemma 5.11 nachgewiesen, daß ein Grundterm aus RH_Z mit Hilfe von Regeln aus R dieselben Patterns erreichen kann, wie mit Hilfe der Auswertungsfunktionen $\text{evaluate}_{\mathsf{HP}}$.

Definition 5.18 (Auswertungsreihenfolge) *Die Auswertungsreihenfolge*

$$\succ_P \subseteq \mathsf{H_B}(\mathsf{D},\mathsf{V}) \times \mathsf{H_B}(\mathsf{D},\mathsf{V})$$

zu einem linearen Pattern p ist definiert wie folgt:

$t \succ_p u$ gdw. $t \equiv_{\mathsf{HP}} u$ und für jede Grundtermsubstitution σ zu der $t\sigma$ typkorrekt ist

$$(\text{len}(t\sigma \to_{\mathsf{HP}}^{>p}) \neq \infty) \Rightarrow (\text{len}(t\sigma \to_{\mathsf{HP}}^{>p}) > \text{len}(u\sigma \to_{\mathsf{HP}}^{>p}))$$

gilt. Die Relation $\succeq_p$ ist genauso definiert, nur daß $>$ durch $\geq$ ersetzt wird. Außerdem gilt $t \succeq_{\mathsf{P_B}(\mathsf{D},\mathsf{V})} u$, wenn $t \succeq_p u$ für jedes lineare Pattern $p \in \mathsf{P_B}(\mathsf{D},\mathsf{V})$ gilt.

Das folgende Lemma zeigt, daß nach einer Auswertung an einem Teilterm, der *nicht* unbedingt der Redex des Terms ist, die Auswertung zu einem Pattern nicht mehr Schritte benötigt als vorher.

Lemma 5.9 (Schachtelung der Auswertungsreihenfolge)
Aus $t|_\pi \succeq_{\mathsf{P_B}(\mathsf{D},\mathsf{V})} t'$ folgt $t \succeq_{\mathsf{P_B}(\mathsf{D},\mathsf{V})} t[t']_\pi$.

Beweis:
Sei σ eine Grundtermsubstitution, wobei $t\sigma$ typkorrekt ist und sei p ein lineares Pattern.

Für $\text{len}(t\sigma \to_{\mathsf{HP}}^{>p}) = \infty$ gilt die Aussage bereits. Sei also $\text{len}(t\sigma \to_{\mathsf{HP}}^{>p}) \neq \infty$.

Weil $\text{len}(t\sigma \to_{\mathsf{HP}}^{>p}) = n$ gilt, gibt es eine Auswertung $t\sigma \to_{\mathsf{HP}}^{*} u$ für einen Term u mit $t\sigma \to_{\mathsf{HP}}^{>p} u$ der Länge n.

Innerhalb von $t\sigma \to_{\mathsf{HP}}^{*} u$ muß $t|_\pi\sigma$ ein Pattern p' erreichen. Da $t'\sigma$ dieses Pattern p' aber wegen $t|_\pi \succeq_{\mathsf{P_B}(\mathsf{D},\mathsf{V})} t'$ mindestens gleich schnell erreicht, gilt

$$\text{len}(t[t']_\pi\sigma \to_{\mathsf{HP}}^{>p}) \leq n$$

und $t \succeq_p t[t']_\pi$. □

Nun können wir mit dem Lemma 5.9 zeigen, daß $\text{term}_{\mathsf{NG}}$ ebenfalls einen Term in der Auswertungsreihenfolge hin zu einem Pattern nicht zurückwirft.

Lemma 5.10 ($\text{term}_{\mathsf{NG}}$ erhält Auswertungsreihenfolge)
Wenn $(t, W) = \text{term}_{\mathsf{NG}}((cs, u), \emptyset)$ gilt, dann gilt $u \succeq_{\mathsf{P_B(D,V)}} t$.

Beweis:
Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\text{eval}}$. So gilt $\text{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = \text{term}_{\mathsf{NG}}((\underline{cs'}, u'), \emptyset)$, $(\underline{cs'}, u') = \text{eval}(\underline{cs}, u)$ und $u \to^{\mathsf{EXT}}_{\mathsf{HP}} u'$. So folgt aus der Definition von $\to^{\mathsf{EXT}}_{\mathsf{HP}}$, daß $u \succ_{\mathsf{P(D,V)}} u'$ gilt. Wegen

$$(t, R) = \text{term}_{\mathsf{NG}}((\underline{cs'}, u'), \emptyset)$$

folgt per Induktionshypothese, daß $u \succ_p u' \succeq_p t$ gilt.

Für $\mathsf{M}((\underline{cs}, u)) \in \{\underline{\text{varexp}}, \underline{\text{case}}\}$ gilt die Aussage direkt, weil dabei $u = t$ gilt.

Sei $\mathsf{M}((\underline{cs}, u)) = \underline{\text{parsplit}}$. So gilt $\text{term}_{\mathsf{NG}}((\underline{cs}, u), \emptyset) = (h\ t_1\ \ldots\ t_n, \bigcup_{i=1}^{n} R_i)$ für $u = h\ u_1\ \ldots\ u_n$ und $(t_i, R_i) = \text{term}_{\mathsf{NG}}((\underline{cs}_i, u_i), \emptyset)$. Per Induktionshypothese gilt $u_i \succeq_{\mathsf{P_B(D,V)}} t_i$. Mit Lemma 5.9 folgt so $u = h\ u_1\ \ldots\ u_n \succeq_p h\ t_1\ \ldots\ t_n$.

Fall $\mathsf{M}((\underline{cs}, u)) = \underline{\text{instance}}$ ist analog zu Fall $\mathsf{M}((\underline{cs}, u)) = \underline{\text{parsplit}}$. □

Durch das folgende Lemma wird nachgewiesen, daß ein Grundterm aus RH_Z mit den Regeln aus $\to_R$ die gleichen Patterns erreichen kann wie mit der Auswertungsfunktion $\text{evaluate}_{\mathsf{HP}}$. Mit Hilfe diese können wir in Satz 5.1 annehmen, das innerhalb einer Kette eines DP-Problems die rechte Seite eines Dependency-Pairs immer zu der linken Seite des nachfolgenden Dependency-Pairs, ausgewertet werden kann, wie es die zugehörige $\overline{\mathsf{NF}}$-Kette im Narrowing-Graphen vorgibt.

Lemma 5.11 ($\to_R$ erreicht gleiche Patterns wie $\to_{\mathsf{HP}}$)
Wenn $t \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$, $t \to^{>p}_{\mathsf{HP}} m$ und p ein lineares Pattern ist, dann gibt es $m' \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ mit $t \to^*_R m'$ und p matcht m' und $t \equiv_{\mathsf{HP}} m'$ gilt.

Beweis:
Wenn $p \in \mathsf{V_L}$ folgt bereits das Gewünschte, also sei $p \notin \mathsf{V_L}$.

Der Beweis wird per Doppelinduktion über die Anzahl der Auswertungsschritte und über die Teiltermrelation geführt.

Sei $t \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ und $t \to^{>p}_{\mathsf{HP}} m$.

Wegen $t \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ gibt es ein $(\underline{cs}, u) \in \mathsf{RQ}_Z$ mit $u[x_1/t_1, \ldots, x_n/t_n] = t$.

Wegen $t \to_{\mathsf{HP}}^{>p} m$ *gibt es die minimalen linearen Patterns* p_i *mit* $t_i \to_{\mathsf{HP}}^{>p_i} m_i$*, so daß der Term* w *und die Substitutionen* σ *und* σ' *existieren für die*

- $u[x_1/p_1, \ldots, x_n/p_n]\sigma' \to_{\mathsf{HP}}^{>p} w$,
- $\forall x_{|\tau|} \in \mathrm{Dom}(\sigma')$: $(\tau = \tau \to \tau') \vee (\tau \in \mathsf{V_T}) \vee (x_{|\tau|}\sigma' = \underline{\mathrm{error}})$ *und*
- $u[x_1/p_1, \ldots, x_n/p_n]\sigma \equiv_{\mathsf{HP}} t$

gilt.

Weil $t_i \lhd t$ *gilt, folgt durch die Induktionshypothese, daß es zu* $t_i \to_{\mathsf{HP}}^{>p_i} m_i$ *den Term* $m_i' \in \mathsf{RH}_Z \cap \mathsf{H}_\mathsf{B}^\mathsf{G}(\mathsf{D}, \mathsf{V})$ *gibt, so daß* p_i *den Term* m_i' *matcht und* $t_i \to_R^* m_i'$ *und* $t_i \equiv_{\mathsf{HP}} m_i'$ *gilt.*

Wegen $t_i \equiv_{\mathsf{HP}} m_i'$ *folgt*

$$u[x_1/t_1, \ldots, x_n/t_n] \equiv_{\mathsf{HP}} u[x_1/m_i', \ldots, x_n/m_i']$$

und $u[x_1/m_i', \ldots, x_n/m_i']$ *erreicht auch das Pattern* p*.*

Wir können nun dem Pfad im Narrowing-Graph ab Knoten $(\underline{cs}, u)$ *entlang den Fallunterscheidungen folgen, die auf die Substitution* $[x_1/p_1, \ldots, x_n/p_n]\sigma$ *passen. Wir folgen dem Pfad so lange, bis wir den ersten Knoten* $(\underline{cs}', u')$ *mit*

$$\mathsf{M}((\underline{cs}', u')) \in \{\underline{\mathrm{instance}}, \underline{\mathrm{parsplit}}\}$$

erreichen. Dieser Pfad existiert, weil die Fallunterscheidungen alle Konstruktorfälle vollständig abdecken und weil alle Terme der Knoten auf dem Pfad vor Knoten $(\underline{cs}', u')$ *nicht vom Pattern* p *gematcht werden, da deren Terme jeweils an oberster Stelle eine Funktionsvariable enthalten, wie es für Knoten mit den Marken* $\underline{\mathrm{case}}$ *und* $\underline{\mathrm{eval}}$ *vorausgesetzt wird. Die Marke* $\underline{\mathrm{varexp}}$ *kommt nicht vor, da* $u[x_1/p_1, \ldots, x_n/p_n]\sigma$ *auf jeden Fall nicht vom Funktionstyp ist, weil das Pattern* $p \notin \mathsf{V_L}$ *erreicht wird. Es wurde keine Fallunterscheidung mit einem noch nicht ausgewerteten Term der Form* $v\ w_1\ \ldots\ w_n$ *mit* $v \in \mathsf{V_F}$ *bedient, weil sonst per Definition von* $\mathrm{evaluate}_{\mathsf{HP}}$ *der Term* $u[x_1/p_1, \ldots, x_n/p_n]\sigma'$ *direkt zu* $\underline{\mathrm{error}}$ *ausgewertet werden würde und so nicht das Pattern* p *erreichen könnte, weil dieses kein Variablenpattern ist und so* $\underline{\mathrm{error}}$ *nicht matcht.*

Zu diesem Pfad wurden Regeln ab Knoten $(\underline{cs}, u)$ *abgelesen und per Definition von* $\mathrm{rules}_{\mathsf{NG}}$ *gibt es eine Regel bis zum Knoten* $(\underline{cs}', u')$*. Also gilt*

$$\mathrm{rules}_{\mathsf{NG}}(u\delta, (\underline{cs}', u')) = \{[\![u\delta]\!]_{\mathsf{APP}} \to [\![r]\!]_{\mathsf{APP}}\} \cup \bigcup_{(\underline{cs}'', u'') \in R} \mathrm{rules}_{\mathsf{NG}}(u'', (\underline{cs}'', u''))$$

mit $(r, W) = \mathrm{term}_{\mathsf{NG}}((\underline{cs}', u'), \emptyset)$ *für einen Term* r*, eine Menge* W *von Knoten und eine Substitution* δ*.*

Aus den Definitionen der Fallunterscheidung und $\mathrm{rules}_{\mathsf{NG}}$ *folgt, daß* $u\delta$ *den Term* $u[x_1/p_1, \ldots, x_n/p_n]\sigma$ *matcht. Dadurch matcht* $u\delta$ *auch* $u[x_1/m'_i, \ldots, x_n/m'_i]$*, da jedes* p'_i *den Term* m'_i *matcht. Also gibt es die Substitution* μ *mit*

$$u\delta\mu = u[x_1/m'_i, \ldots, x_n/m'_i]$$

und μ *ist eine* $\mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{C}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$*-Substitution, weil* $m'_1, \ldots, m'_n \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{C}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ *gilt und* δ *nur Konstruktoren einbringen kann, so daß von den einzelnen* m'_i *nur die Konstruktor-Kontexte durch* δ *weggenommen werden können und in* μ *nur die Rümpfe übrig bleiben. Wegen des Aufbaus von* RH_Z *müssen diese Rümpfe wieder in* $\mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{C}}_{\mathsf{B}}(\mathsf{D}, \mathsf{V})$ *liegen.*

Auf dem Pfad ab $(\underline{cs}, u)$ *für die Substitution* $[x_1/p_1, \ldots, x_n/p_n]\sigma$ *gibt es mindestens eine Auswertung.* $\mathsf{M}((\underline{cs}, u)) \in \{\underline{\text{case}}, \underline{\text{eval}}\}$ *folgt aus* $(\underline{cs}, u) \in \mathsf{RQ}_Z$ *mit Lemma 5.5.*

- *Für* $\mathsf{M}((\underline{cs}, u)) = \underline{\text{eval}}$ *ist die Auswertung schon im ersten Schritt vorhanden.*

- *Für* $\mathsf{M}((\underline{cs}, u)) = \underline{\text{case}}$ *muß eine Auswertung an einem späteren Knoten folgen, da wegen des Aufbaus der Narrowing-Graphen und da* u *nicht den Funktionstyp hat, weil* $u[x_1/p_1, \ldots, x_n/p_n]\sigma$ *das Pattern* $p \notin \mathsf{V}_{\mathsf{L}}$ *erreicht, auf einen Knoten mit Marke* $\underline{\text{case}}$ *wieder nur Knoten mit den Marken* $\underline{\text{case}}$ *oder* $\underline{\text{eval}}$ *folgen können.*

Da auf dem Pfad mindestens eine Auswertung liegt, muß

$$u[x_1/m'_i, \ldots, x_n/m'_i] \succ_p u'\mu$$

gelten, da Fallunterscheidungen keine echten Modifikationen vornehmen, sondern nur Fälle zuordnen. Wegen $(r, W) = \mathrm{term}_{\mathsf{NG}}((\underline{cs}', u'), \emptyset)$ *folgt per Lemma 5.10, daß* $u' \succeq_p r$ *gilt und so auch* $u'\mu \succeq_p r\mu$*. Zusammen ergibt sich*

$$u[x_1/m'_i, \ldots, x_n/m'_i] \succ_p u'\mu \succeq_p r\mu$$

Durch das Lemma 5.4 und $u[x_1/t_1, \ldots, x_n/t_n] \equiv_{\mathsf{HP}} u[x_1/m'_i, \ldots, x_n/m'_i]$ *folgt* $u[x_1/m'_i, \ldots, x_n/m'_i] \equiv_{\mathsf{HP}} r\mu$ *und so muß* $r\mu$*, wegen des Patternlemmas 5.2 das Pattern* p *ebenfalls erreichen.*

Wegen des Lemmas 5.8 gilt $r \in \mathsf{RH}_Z$*, so daß, weil* μ *eine* $\mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D},\mathsf{V})$*-Substitution ist, auch* $r\mu \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D},\mathsf{V})$ *folgt.*

Wegen

$$t = u[x_1/t_1, \ldots, x_n/t_n] \succeq_p u[x_1/m'_i, \ldots, x_n/m'_i] \succ_p r\mu$$

ist die Auswertung $r\mu \to^{>p}_{\mathsf{HP}} m''$ *für ein* m'' *mindestens einen Schritt kürzer als die von* $t \to^{>p}_{\mathsf{HP}} m$*. Mit* $r\mu \in \mathsf{RH}_Z \cap \mathsf{H}^{\mathsf{G}}_{\mathsf{B}}(\mathsf{D},\mathsf{V})$ *folgt per Induktionshypothese aus* $r\mu \to^{>p}_{\mathsf{HP}} m$*, daß es einen Term* m' *gibt mit* $r\mu \to^*_R m'$*,* $m' \equiv_{\mathsf{HP}} r\mu$ *und* p *matcht* m'*.*

Insgesamt folgt also, p *matcht* m'*,*

$$t = u[x_1/t_1, \ldots, x_n/t_n] \to^*_R u[x_1/m'_i, \ldots, x_n/m'_i] \to_R r\mu \to^*_R m'$$

und

$$m' \equiv_{\mathsf{HP}} r\mu \equiv_{\mathsf{HP}} u[x_1/m'_i, \ldots, x_n/m'_i] \equiv_{\mathsf{HP}} u[x_1/t_1, \ldots, x_n/t_n] = t$$

□

Mit dem folgenden zentralen Satz kann aus der Endlichkeit der DP-Probleme zu einem Narrowing-Graph die NF-Terminierung des Startterms gefolgert werden. In diesem Satz wird gezeigt, daß zu jeder $\overline{\mathsf{NF}}$-Kette eine unendliche Kette in einem der zugehörigen DP-Probleme existiert.

Satz 5.1 (Existenz einer unendlichen $(\mathcal{P},\mathcal{R})$**-Kette zu einer** $\overline{\mathsf{NF}}$**-Kette)**
Sei q *ein Knoten aus einer maximalen starken Zusammenhangskomponente* Z *eines geschlossenen Narrowing-Graphen. So gibt es zu jeder* $\overline{\mathsf{NF}}$*-Kette ab Knoten* $q \in Z$ *eine unendliche* $(\mathcal{P},\mathcal{R})$*-Kette zu dem abgelesenen DP-Problem* $(\mathcal{P},\mathcal{R})$ *der maximalen starken Zusammenhangskomponente* Z*.*

Beweis:
Die $\overline{\mathsf{NF}}$*-Kette ab Knoten* q *läßt sich unterteilen in unendlich viele Segmente zwischen Generalisierungskanten. Denn jede unendliche Folge von Nachfolgerknoten in einem geschlossenen Narrowing-Graph läuft wegen seiner Endlichkeit in einen Zykel. Wegen des Zykellemmas 4.8 gibt es so mindestens eine Generalisierungskante, die unendlich oft von einer unendlichen Folge von Nachfolgerknoten überschritten wird.*

Jedes Segment beginnt an einem Knoten, in den eine Generalisierungskante führt, und endet an einem, aus dem eine Generalisierungskante entspringt.

Die $\overline{\mathsf{NF}}$*-Kette läßt sich als Folge der Segmente*

$$\begin{aligned}((\underline{cs}_1, u_1), \sigma_1) &- ((\underline{cs}'_1, u'_1), \sigma'_1)\\ ((\underline{cs}_2, u_2), \sigma_2) &- ((\underline{cs}'_2, u'_2), \sigma'_2)\\ ((\underline{cs}_3, u_3), \sigma_3) &- ((\underline{cs}'_3, u'_3), \sigma'_3)\\ &\vdots\end{aligned}$$

darstellen. Für jedes Segment wird jetzt gezeigt, daß es einem Dependency-Pair aus $\mathcal{P}$ *des DP-Problems* $(\mathcal{P}, \mathcal{R})$ *zugeordnet werden kann. Diese lassen sich dann durch eine geeigente Substitution* σ *zu einer unendlichen* $(\mathcal{P}, \mathcal{R})$*-Kette verknüpfen.*

Ab jetzt sei R *das Termersetzungssystem auf Haskelltermen zu* $\mathcal{R}$*.*

Im folgenden wird pro Segment die Substitution σ *immer weiter festgelegt und das nächste Dependency-Pair für die* $(\mathcal{P}, \mathcal{R})$*-Kette ausgewählt.*

Die erste Festlegung für σ *ist, daß sie auf den Variablen* $\mathsf{V_L}(u_1)$ *genauo arbeitet wie die Substitution* σ_1*.*

Nun folgt die Argumentation für jedes einzelne Segment.

Weil σ_i *eine Normalform-Substitution ist, existiert per Definition der Ablesefunktion* $\mathrm{dpProblem}_{\mathsf{NG}}$ *ein Dependency-Pair* $[\![u_i\delta_i]\!]_{\mathsf{APP}} \to [\![t_i]\!]_{\mathsf{APP}} \in \mathcal{P}$*, für das der Term* $u_i\delta_i$ *den Term* $u_i\sigma_i$ *matcht. Dies kann wie folgt gezeigt werden:*

Per Definition von $\mathrm{dpProblem}_{\mathsf{NG}}$ *folgt, daß* δ_i *in ihrem Bildbereich nur aus Konstruktoren aufgebaute Terme enthält, und daß nur für Variablen* $x_{|d\ \tau_1\ \ldots\ tau_m|} \in \mathrm{Dom}(\delta)$*, für die* $d \in \underline{\mathsf{TyCons}}_{\mathsf{D}} \setminus \{\to\}$ *gilt. Da die Fallunterscheidungen alle Konstruktoren abdecken, gibt es keine Konstruktorkombinationen, die nicht durch einen Pfad ab Knoten* $(\underline{cs}_i, u_i)$ *abgedeckt werden, außer die Substitution setzt den Term* $\underline{\mathrm{error}}$ *ein. In diesem Fall aber ist per Definition der Auswertungsfunktion* $\mathrm{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}$*, die die Fallunterscheidung bestimmt, sichergestellt, daß der Term* $\underline{\mathrm{error}}$ *zur Auswertung kommt und die Haskellauswertungsstrategie die gesamte Auswertung mit* $\underline{\mathrm{error}}$ *abbricht. Das heißt, der Term* $u_i\sigma_i$ *terminiert mit der Normalform* $\underline{\mathrm{error}}$*, das ist aber laut Voraussetzung gerade nicht der Fall.*

*Für Fallunterscheidungen auf Typvariablen funktioniert eine analoge Argumentation, nur daß hier kein „*error*“-Typ zur Verfügung steht und deswegen die Substitution* σ_i *nur eine korrekte Instanz für eine Member-Variable wählen kann, welche in den abgedeckten Bereich dieser Fallunterscheidung fällt, da sonst unweigerlich der Typchecker den Term* $u_i\sigma_i$ *als ungültig zurückweisen würde.*

Nicht jeder Pfad ab Knoten $(\underline{cs}_i, u_i)$ *ist in einem Dependency-Pair aus* $\mathcal{P}$ *repräsentiert, sondern nur die, die auf Generalisierungskanten zeigen. Auch diese Bedingung muß* σ_i *erfüllen, da sonst der Zyklus, auf dem sich die* $\overline{\mathsf{NF}}$*-Kette befindet, verlassen wird, weil* $\mathrm{succ}_{\mathsf{NG}}^{\neg\mathsf{NF}}$ *bei Fallunterscheidungen immer den Pfad wählt, der auf die begleitende Substitution paßt.*

Weil vom Knoten $(\underline{cs}'_i, u'_i)$ *eine Generalisierungskante zum Knoten* $(\underline{cs}_{i+1}, u_{i+1})$ *führt, gibt es die Substitution* $\mu = [x_1/w_1, \ldots, x_n/w_n]$ *mit* $u_{i+1}\mu = u'_i$ *und* $x_j \in \mathsf{V}_\mathsf{L}(u_{i+1})$ *wegen* $((\underline{cs}'_i, u'_i), x_j, (\underline{ds}_j, w_j)) \in \mathrm{subterm}$*.*

Durch die Bedingungen vom Fall 6 der Funktion $\mathrm{succ}_{\mathsf{NG}}^{\neg\mathsf{NF}}$ *folgt, daß* $w_j\sigma_i{\downarrow_\mathsf{HP}}$ *existiert. So gibt es zu* $w_j\sigma_i{\downarrow_\mathsf{HP}}$ *ein maximales lineares Pattern* p_j*, welches* $w_j\sigma_i{\downarrow_\mathsf{HP}}$ *matcht.*

Durch die Funktion $\mathrm{term}_\mathsf{NG}$ *wurde der Term* t_i *für das aktuelle Dependency-Pair zusammengesetzt. Es gilt also* $(t_i, W) = terms((\underline{cs}'_i, u'_i), X)$ *für eine Filtermenge* X*. Dabei wurde der Term* u_{i+1} *mit einer Substitution*

$$\mu' = [x_1/w'_1, \ldots, x_n/w'_n]$$

verfeinert, wobei $(w'_j, S_j) = \mathrm{term}_\mathsf{NG}((\underline{ds}, w_j), X)$*, so daß* $t_i = u_{i+1}\mu'$ *gilt.*

Sei $(w''_j, S'_j) = \mathrm{term}_\mathsf{NG}((\underline{ds}, w_j), \emptyset)$*. Aus der Definition von* $\mathrm{term}_\mathsf{NG}$ *folgt, daß* w'_j *den Term* w''_j *matcht mit Hilfe der Substitution* α'_j*, weil die Filtermenge* X *die Funktion* $\mathrm{term}_\mathsf{NG}$ *dazu veranlaßt, einige Teilterme von* w''_j *durch frische Variablen zu ersetzen.*

Sei $x \in \mathrm{Dom}(\alpha'_j)$ *und* $w'_j|_\pi = x$*. Per Definition von* $\mathrm{term}_\mathsf{NG}$ *und* $\mathrm{freeapps}_\mathsf{NG}$ *liegt die Stelle* π *so hoch, daß diese schon im Pattern* p_j *liegt. Nun kann die Substitution* α *so gewählt werden, daß* $x\alpha = w_j\sigma_i{\downarrow_\mathsf{HP}}|_\pi$ *gilt, weil* p_j *die Normalform* $w_j\sigma_i{\downarrow_\mathsf{HP}}$ *matcht und so ist* α *auch eine* $\mathsf{RH}_Z \cap \mathsf{H}_\mathsf{B}^\mathsf{C}(\mathsf{D}, \mathsf{V})$*-Substitution.*

Nun kann σ *so gewählt werden, daß sie auf den Domains von allen* α_j *wie alle* α_j *wirkt, weil* α_j *und* α_k *disjunkte Domains für* $j \neq k$ *haben.*

Wegen Lemma 5.7 ist $w'_j \in \mathsf{RH}_Z$*, da es*

$$(u_i\delta_i, (\underline{cs}'_i, u'_i), (t_i, W)) \in \mathrm{calls}_\mathsf{NG}(u_i, (\underline{cs}_i, u_i))$$

gibt mit $S_j \subseteq W \subseteq \mathsf{RQ}_Z$*.*

Es existiert ein m_j, so daß $w'_j\sigma \rightarrow_{\mathsf{HP}}^{>p_j} m_j$ gilt und da $w'_j \in \mathsf{RH}_Z$ und σ eine $\mathsf{RH}_Z \cap \mathsf{H}_\mathsf{B}^\mathsf{G}(\mathsf{D},\mathsf{V})$-Substitution ist, gilt wegen Lemma 5.11, daß es ein m'_j gibt mit

$$w'_j\sigma \rightarrow_R^* m'_j$$

und p_j matcht m'_j mit $m'_j \equiv_{\mathsf{HP}} m_j$.

So gilt

$$t_i\sigma = u_{i+1}\mu'\sigma = u_{i+1}[x_1/w'_1,\ldots,x_n/w'_n]\sigma \rightarrow_R^* u_{i+1}[x_1/m'_1,\ldots,x_n/m'_n] = t'_i$$

Die linke Seite des nächsten Dependency-Pairs für das nächste Segment ist damit durch Regeln aus R aus der rechten Seite des jetzigen erreichbar.

O.B.d.A können wir annehmen, daß alle freien Variablen, die in dem nächsten gewählten Dependency-Pair $[\![u_{i+1}\delta_{i+1}]\!]_{\mathsf{APP}} \rightarrow [\![t_{i+1}]\!]_{\mathsf{APP}} \in \mathcal{P}$ auftreten, nicht in einem der bis jetzt gewählten Dependency-Pairs vorkommen.

Nun können wir σ für die Variablen $\mathsf{V}_\mathsf{L}(u_{i+1}\delta_{i+1})$ aus der linken Seite des nächsten Dependency-Pairs so festlegen, daß $t'_i = u_{i+1}\delta_{i+1}\sigma$ gilt.

Die Folge

$$\begin{aligned}
[\![u_1\delta_1]\!]_{\mathsf{APP}} &\rightarrow [\![t_1]\!]_{\mathsf{APP}}\\
[\![u_2\delta_2]\!]_{\mathsf{APP}} &\rightarrow [\![t_2]\!]_{\mathsf{APP}}\\
[\![u_3\delta_3]\!]_{\mathsf{APP}} &\rightarrow [\![t_3]\!]_{\mathsf{APP}}\\
&\vdots
\end{aligned}$$

ist mit der Substitution $[\![\sigma]\!]_{\mathsf{APP}}$ zusammen eine unendliche $(\mathcal{P},\mathcal{R})$-Kette. □

Mit Hilfe des Folgenlemmas wird nachgewiesen, daß von einem nicht NF-terminierenden Startknoten ein der Ausgangsknoten einer $\overline{\mathsf{NF}}$-Kette erreicht werden kann.

Lemma 5.12 (Folgenlemma) *Wenn der Startterm vom Startknoten $(\underline{cs}, u)$ nicht NF-terminiert, gibt es in dem geschlossenen Narrowing-Graph eine $\overline{\mathsf{NF}}$-Kette ab einem Knoten q.*

Beweis:
Wegen $u \notin \mathsf{NF}_{\mathsf{HP}}$ gibt es eine $\mathsf{NF}_{\mathsf{HP}}^\mathsf{G}$-Substitution σ' für u, so daß $u\sigma'$ typkorrekt ist und $u\sigma' \notin \mathsf{NF}_{\mathsf{HP}}$ gilt, also gibt es auch eine typkorrekte Normalform-Substitution σ für u, so daß $u\sigma \notin \mathsf{NF}_{\mathsf{HP}}$ gilt. Wegen des Nachfolgerlemmas 5.1 läßt sich mit Hilfe der Nachfolgerfunktion $\mathrm{succ}_{\mathsf{NG}}^{\neg\mathsf{NF}}$ die unendliche Folge $(q_1,\sigma_1),(q_2,\sigma_2),(q_3,\sigma_3),\ldots$

aus Paaren von Knoten und Substitutionen zusammensetzen, so daß $\sigma = \sigma_1$, $q_1 = (\underline{cs}, u)$ *und*

$$\mathrm{succ}_{\mathsf{NG}}^{\neg\mathsf{NF}}(q_i, \sigma_i) = (q_{i+1}, \sigma_{i+1})$$

für alle $i > 0$ *gilt. Da der Narrowing-Graph endlich ist, läuft die unendliche Folge in einen Zyklus. Durch das Zykellemma* 4.8 *wissen wir, daß es mindestens eine Generalisierungskante in einem Zyklus geben muß. Sei* q_j *der erste Knoten von der unendlichen Folge, welcher in einem Zyklus liegt und aus dem eine Generalisierungskante entspringt. Zusätzlich folgt aus dem Nachfolgerlemma, daß die Substitution* σ_j *eine Normalform-Substitution sein muß, weil* σ_1 *eine Normalform-Substitution ist, so daß die unendliche Folge ab Knoten* q_j *eine* $\overline{\mathsf{NF}}$*-Kette ist.* □

Die Korrektheit der Terminierungsanalyse kann nun durch einfaches Zusammenbauen der vorher bewiesenen Sätze und Lemmata geführt werden.

Satz 5.2 (Korrektheit der Terminierungsanalyse)
Wenn alle zu einem abgeschlossenen Narrowing-Graph abgelesenen DP-Probleme endlich sind, NF-terminiert der Startterm.

Beweis:
Seien alle DP-Probleme endlich. Annahme, der Startterm NF-terminiert nicht. Wegen des Folgenlemmas 5.12 gibt es so eine $\overline{\mathsf{NF}}$*-Kette ab Knoten* q*. Da* q *in einem Zyklus liegt, gibt es zu diesem die maximale starke Zusammenhangskomponente* Z *mit* $q \in Z$*. Sei* $(\mathcal{P}, \mathcal{R})$ *das zu* Z *durch die Funktion* $\mathrm{dpProblem}_{\mathsf{NG}}$ *abgelesene DP-Problem. Aus dem Satz 5.1 folgt so, daß es eine unendliche* $(\mathcal{P}, \mathcal{R})$*-Kette geben muß. Das widerspricht der Voraussetzung, daß alle DP-Probleme endlich sind und so folgt, daß der Startterm NF-terminiert.* □

Kapitel 6

Narrowing-Strategie

Im Kapitel 5 wurde gezeigt, wie aus einem Narrowing-Graph DP-Probleme abgelesen werden können. Der Aufbau dieser DP-Probleme wird dabei ausschließlich von dem Narrowing-Graph bestimmt. Um möglichst einfache DP-Probleme für einen Endlichkeitsnachweis mit AProVE [GTSKF04] zu erzeugen, muß der Aufbau des Narrowing-Graphen entsprechend gesteuert werden. Die Transformationensfolge aus dem Abschlußlemma 4.10 zum Schließen eines Narrowing-Graphen ist, wie wir noch sehen werden, nicht optimal. Deswegen werden in diesem Kapitel bessere Vorgehensweisen für den Aufbau eines Narrowing-Graphen motiviert, welche am Ende in dem Algorithmus BuildGraph zusammengefaßt werden. Dieser erzeugt Narrowing-Graphen, aus denen für den Endlichkeitsnachweis einfachere DP-Probleme abgelesen werden können.

An dem Haskellprogramm aus Beispiel 6.1 soll die Wirkung der Transformationsfolge aus dem Beweis des Abschlußlemmas 4.10 verdeutlicht werden. Bei Anwendung dieser Transformationsfolge für den Startterm `take` (`from` i) n entsteht der Narrowing-Graph aus Abbildung 6.1, bei dem aber Knoten weggelassen wurden, die keine Verbindung zum Startknoten besitzen,

Beispiel 6.1

```
data  Nat      =  Succ Nat | Zero
data  List  a  =  Cons a (List a) | Nil

take  _          Zero      =  Nil
take  Nil        _         =  Nil
take  (Cons y ys)  (Succ n)  =  Cons y (take ys n)

from x  =  Cons x (from (Succ x))
```

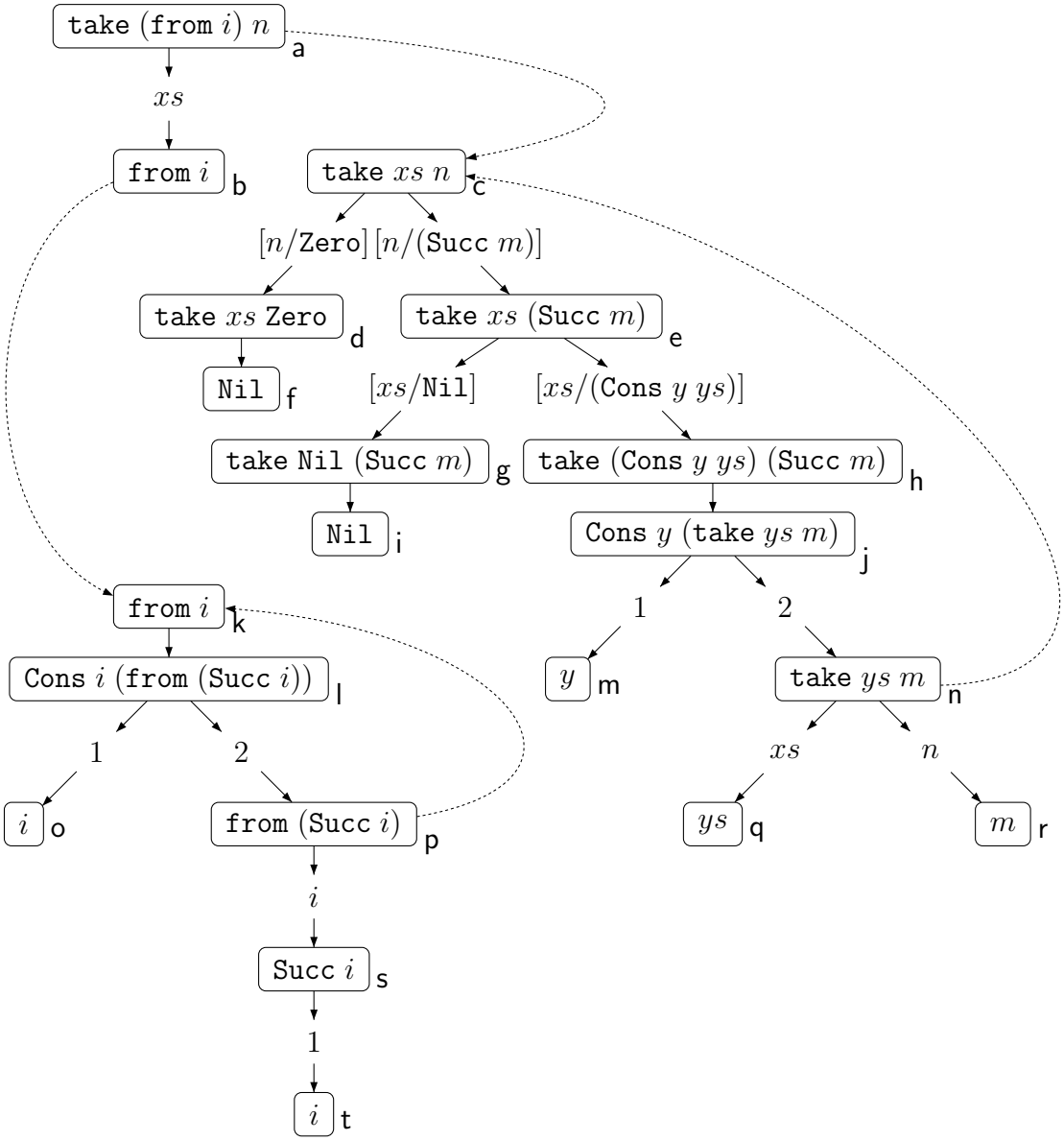

Abbildung 6.1: Narrowing-Graph zu Beispiel 6.1 und Startterm take (from i) n.

Zu diesem Narrowing-Graph ergeben sich zu den maximalen starken Zusammenhangskomponenten $\{\mathsf{c},\mathsf{e},\mathsf{h},\mathsf{j},\mathsf{n}\}$ und $\{\mathsf{k},\mathsf{l},\mathsf{p}\}$ die DP-Probleme

$$\mathrm{dpProblem}_{\mathsf{NG}}(\{\mathsf{c},\mathsf{e},\mathsf{h},\mathsf{j},\mathsf{n}\}) = (\{\mathtt{take}'(\mathtt{Cons}'y'ys)^\sharp(\mathtt{Succ}'m) \ \rightarrow\ \mathtt{take}'ys'm\}, \emptyset)$$

und

$$\mathrm{dpProblem}_{\mathsf{NG}}(\{\mathsf{k},\mathsf{l},\mathsf{p}\}) = (\{\mathtt{from}^\sharp i \ \rightarrow\ \mathtt{from}^\sharp(\mathtt{Succ}'i)\}, \emptyset),$$

in denen jeweils die Regelmengen leer sind. Eine Analyse dieser DP-Probleme zeigt, daß das erste endlich ist, während das zweite nicht endlich ist, obwohl der

Startterm take (from i) n NF-terminiert. Der Grund dafür ist, daß from i als einzelner Term nicht NF-terminiert. Aber kombiniert mit dem Term take xs n ergibt sich der Term take (from i) n, welcher doch NF-terminiert. Für einen NF-terminierenden Term $t \in \mathsf{NF_{HP}}$ mit $t|_\pi \in \mathsf{V_L}$ kann $t[w]_\pi \in \mathsf{NF_{HP}}$ gelten, auch wenn w nicht NF-terminiert. Die Forderung, die durch eine Erweiterung um den Knoten k innerhalb der Transformationsfolge entstanden ist, daß der Term from i NF-terminieren muß, um nachzuweisen, daß der Startterm NF-terminiert, ist also zu stark.

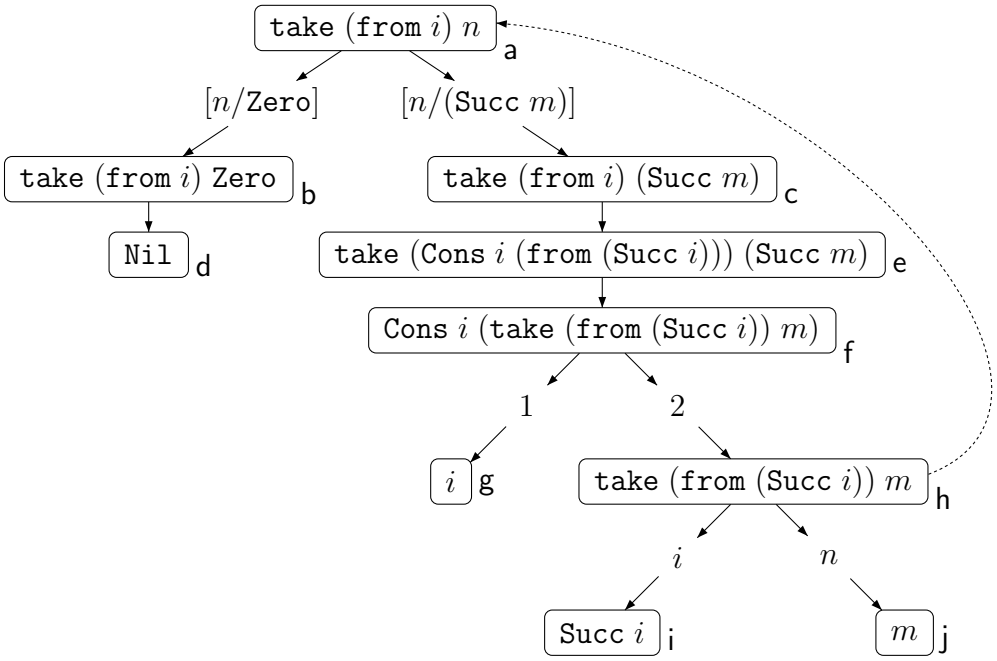

Abbildung 6.2: Verbesserter Narrowing-Graph zu Beispiel 6.1 und Startterm take (from i) n

Der NF-Terminierungsnachweis für den Startterm take (from i) n gelingt, wenn die Erweiterung um den Knoten $(\emptyset, \mathtt{from}\ i)$ ausgelassen wird, wie es bei der Erstellung des Narrowing-Graphen aus Abbildung 6.2 geschehen ist. Zu dessen maximaler starker Zusammenhangskomponente $\{\mathsf{a}, \mathsf{c}, \mathsf{e}, \mathsf{f}, \mathsf{h}\}$ ergibt sich das DP-Problem

$$\begin{aligned}&\mathrm{dpProblem}_{\mathsf{NG}}(\{\mathsf{a}, \mathsf{c}, \mathsf{e}, \mathsf{f}, \mathsf{h}\}) := \\ &\quad (\{\mathtt{take}'(\mathtt{from}'i)^\sharp(\mathtt{Succ}'y) \ \rightarrow\ \mathtt{take}'(\mathtt{from}'(\mathtt{Succ}'i))^\sharp y\}, \emptyset)\end{aligned}$$

dessen Endlichkeitsnachweis gelingt.

Die Transformationsfolge aus dem Abschlußlemma 4.10 ist also offensichtlich nicht optimal. Ein kritischer Punkt beim Erstellen der Narrowing-Graphen ist die Erweiterung um einen Knoten. Aber einfach die Erweiterung wegzulassen, führt dazu, daß sehr viele Narrowing-Graphen nicht mehr geschlossen werden können, wie in Abbildung 6.3 zu sehen ist.

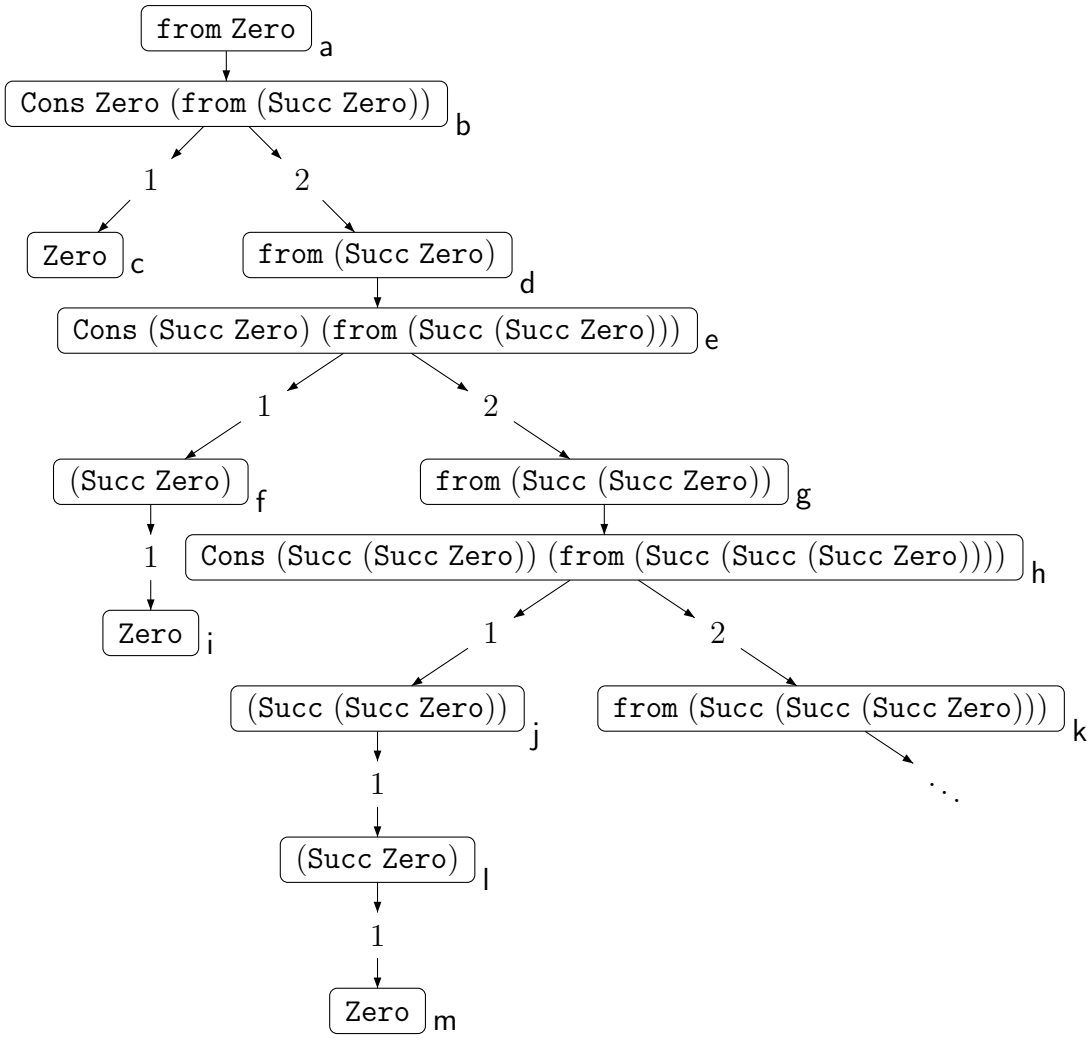

Abbildung 6.3: Offener Narrowing-Graph zu Beispiel 6.1 und Startterm `from Zero`.

Jeder Versuch, mit Hilfe der Instantiierung einen Abschluß des Narrowing-Graphen aus Abbildung 6.3 zu erreichen, schlägt fehl, weil es jeweils zu den Knoten d, g und k keinen allgemeineren gibt, zu dem eine Generalisierungskante gezogen werden könnte. Erst eine Erweiterung um den Knoten $(\emptyset, \texttt{from}\ i)$ würde den Abschluß des Graphen ermöglichen. Denn nun können die Generalisierungskanten von d, g und k zu Knoten $(\emptyset, \texttt{from}\ i)$ gezogen werden, während der Knoten $(\emptyset, \texttt{from}\ i)$ wie Knoten k in Abbildung 6.1 behandelt werden kann.

Es wurden nun zwei Beispiele vorgestellt, in denen der Knoten $(\emptyset, \texttt{from}\ i)$ durch Anwesenheit die Terminierungsanalyse unmöglich macht und ein anderes Mal durch seine Abwesenheit den Abschluß der Narrowing-Graphen verhindert und so wiederum die Terminierungsanalyse blockiert. Nun könnte man argumentieren, daß seine Abwesenheit die bessere Wahl sei, weil der Abschluß eines Narrowing-Graphen mit einem nicht NF-terminierenden Startterm nicht von Relevanz ist, da dessen Terminierungsbeweis ohnehin scheitern muß.

Dieser Ansatz greift aber zu kurz, wie der Narrowing-Graph in Abbildung 6.4 für den Startterm $\texttt{plus Zero}\ n$ und dem Haskellprogramm aus Beispiel 6.2 zeigt.

Beispiel 6.2

$$\begin{array}{lllll} \texttt{data} & \texttt{Nat} & = & \texttt{Succ Nat} \mid \texttt{Zero} \\ \\ \texttt{plus} & x & (\texttt{Succ}\ y) & = & \texttt{plus}\ (\texttt{Succ}\ x)\ y \\ \texttt{plus} & x & \texttt{Zero} & = & x \end{array}$$

Dieser kann ohne eine Erweiterung nicht geschlossen werden, weil Knoten i sonst fehlen würde, und da der Knoten d keine Instanz eines anderen Knotens wäre, könnten so an diesem nur andere Transformationen angewendet werden. Eine Auswertung, Variablenexpansion oder eine Parameteraufteilung wären in dem Fall nicht möglich, denn deren Voraussetzungen werden nicht erfüllt. Es bleibt nur eine Fallunterscheidung ähnlich der von Knoten a. Darauf folgend könnten an den so neu enstanden Knoten, die jeweils wieder den Knoten b und c gleichen, Auswertungen vorgenommen werden. Bei dem zweiten würden wir eine zu dem Term von Knoten e ähnliche Normalform erhalten. Für den anderen neuen Knoten der Form $(\emptyset, \texttt{plus}\ (\texttt{Succ}\ (\texttt{Succ Zero}))\ l)$ bleibt wieder nur eine Fallunterscheidung, da für diesen wieder keine Instantiierung verwendet werden kann. Erst die Erweiterung um den Knoten $(\emptyset, \texttt{plus}\ (\texttt{Succ}\ i)\ n)$ ermöglicht einen Abschluß, so daß die NF-Terminierung des Terms $\texttt{plus Zero}\ n$ anhand des DP-Problems

$$\text{dpProblem}_{\mathsf{NG}}(\{\mathsf{i}, \mathsf{k}, \mathsf{m}\}) := (\{\texttt{plus}'(\texttt{Succ}'i)^\sharp(\texttt{Succ}'y) \to \texttt{plus}'(\texttt{Succ}'(\texttt{Succ}'i))^\sharp\}, \emptyset)$$

der maximalen starken Zusammenhangskomponente $\{\mathsf{i}, \mathsf{k}, \mathsf{m}\}$ durch AProVE nachgewiesen werden kann.

Es zeigt sich, daß die Erweiterung nicht weggelassen werden kann, da sonst eigentlich erfolgreiche Terminierungsnachweise wegen unschließbarer Narrowing-Graphen scheitern müßten und so die Mächtigkeit dieser Methode stark eingeschränkt wäre. Andererseits sollte sie mit Bedacht verwendet werden, da eine Erweiterung mit ungeeigneten Knoten sich ebenfalls negativ auswirken kann. Es stellt sich die Frage, wie ein guter Kandidat für den Term in einem Erweiterungsknoten gefunden werden kann. Zu erkennen ist, daß der Kandidat eine

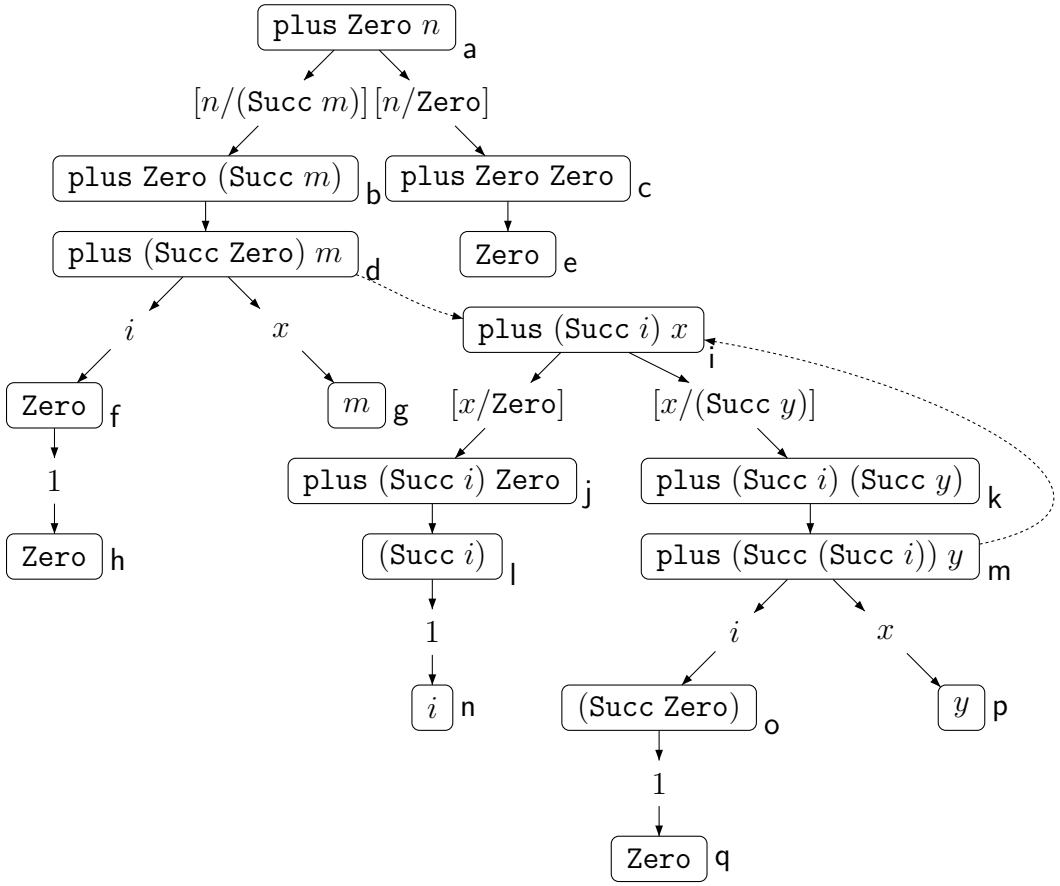

Abbildung 6.4: Narrowing-Graph zu Beispiel 6.2 und Startterm `plus Zero` n.

Generalisierung eines Terms von einem anderen Knotens sein sollte. In unserem Beispiel aus Abbildung 6.4 ist der Erweiterungsterm `plus` (`Succ` i) n allgemeiner als jeder der Terme

`plus` (`Succ` (`Zero`)) m

`plus` (`Succ` (`Succ Zero`)) l

`plus` (`Succ` (`Succ` (`Succ Zero`))) k

...

die in dem Narrowing-Graph entstehen würden, wenn keine Erweiterung zur Verfügung stünde. Offensichtlich wächst der zweite Parameter von `plus` beständig, während im Erweiterungsterm an dieser Stelle eine frische Variable steht. Die Idee ist also, frische Variablen an Positionen zu setzen, an denen Terme während der Erstellung des Narrowing-Graphen wachsen. Dieses Wachstum wird durch die n-fache Schachtelung so charakterisiert, daß eine automatische Erstellung der Graphen ermöglicht wird.

Definition 6.1 (n-fache Schachtelung) *Eine Stelle $\pi_1 \in Occ(u)$ eines Terms u ist n-fach geschachtelt, wenn die Stellen $\pi_2, \ldots, \pi_n \in Occ(u)$ und der Kopf $h \in \mathsf{V_F} \cup \underline{\mathsf{Cons_D}}$ mit $\epsilon \leq_{\text{pre}} \pi_1 \leq_{\text{pre}} \ldots \leq_{\text{pre}} \pi_n$, $\pi_i \neq \pi_j$ für alle $i \neq j$ und $u|_{\pi_i} = h\ t_{i,1}\ \ldots\ t_{i,m_i}$ für alle i existieren.*
Zusätzlich wird noch die Funktion nestedPos $:: \mathsf{H_B(D,V)} \times \mathbb{N} \longrightarrow \mathbb{N}^*$ *benötigt, welche zu jedem Term die am weitesten links außen liegende, n-fach geschachtelte Stelle liefert. Sie ist folgendermaßen definiert:*

$$\text{nestedPos}(u,n) := \begin{cases} \pi, \textit{ falls } \pi \textit{ } n\textit{-fach geschachtelt ist} \\ \quad \textit{und für alle } \pi' \textit{ die } n\textit{-fach geschachtelt} \\ \quad \textit{sind, } \pi <_{\text{lex}} \pi' \textit{ gilt} \\ \epsilon, \textit{ sonst} \end{cases}$$

Ein Term ist n-fach geschachtelt, sobald sich in diesem ein Applikationskopf h (eine Funktionsvariable oder ein Konstruktor) aufstapelt, das heißt, ein Parameter von h enthält wieder einen Term der Form $h\ t_1\ \ldots\ t_n$ und ein t_i enthält wieder solch einen Term usw. bis zur Verschachtelungstiefe n. Zum Beispiel ergibt sich für die Stelle 1 des Terms `plus` (`Succ` (`Succ` (`Succ` `Zero`))) k, daß diese 3-fach geschachtelt ist, so daß sich

$$\text{nestedPos}(\texttt{plus}\ (\texttt{Succ}\ (\texttt{Succ}\ (\texttt{Succ}\ \texttt{Zero})))\ k, 3) = 1$$

ergibt, weil für die drei Stellen 1, 11 und 111 die Bedingungen

- $1 \leq_{\text{pre}} 11 \leq_{\text{pre}} 111$,
- `plus` (`Succ` (`Succ` (`Succ` `Zero`))) $k|_1 =$ (`Succ` (`Succ` (`Succ` `Zero`))),
- `plus` (`Succ` (`Succ` (`Succ` `Zero`))) $k|_{11} =$ (`Succ` (`Succ` `Zero`)) und
- `plus` (`Succ` (`Succ` (`Succ` `Zero`))) $k|_{111} =$ (`Succ` `Zero`)

erfüllt sind und keine anderen Stellen 3-fach geschachtelt sind.

Nachdem eine Stelle π im Term t als n-fach geschachtelt identifiziert wurde, kann diese durch eine frische Variable ersetzt werden. Dabei ist zu beachten, daß dieser neue Term $t[y]_\pi$ mit der frischen Variablen y noch einmal den Typchecker durchlaufen muß, damit der Typ der Variablen y in diesem Term neu bestimmt wird. Das ist nötig, um Situationen wie im Narrowing-Graph aus Abbildung 6.5 für Startterm `wrap Zero` und dem Haskellprogramm aus Beispiel 6.3 zu vermeiden.

Beispiel 6.3

```
data Maybe a = Just a | Nothing

wrap :: a → Nat
wrap x = wrap (Just x)
```

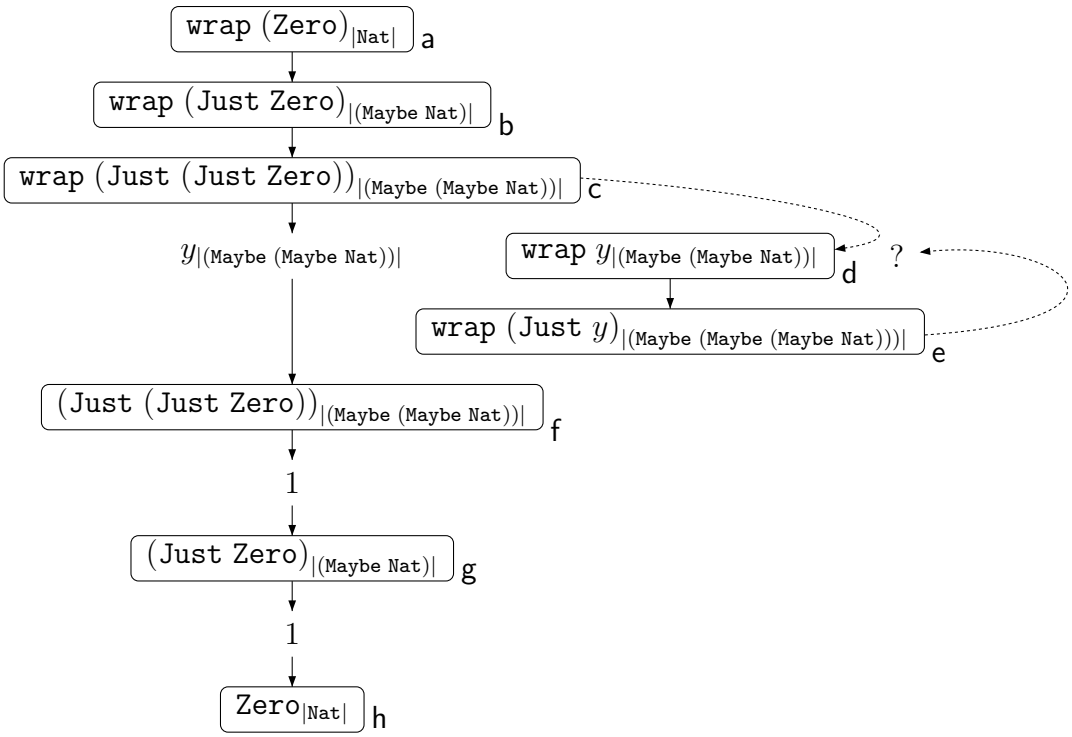

Abbildung 6.5: Narrowing-Graph[1] zu Beispiel 6.3 und Startterm `wrap Zero`.

Die Kante von Knoten `e` nach `d` kann nicht gezogen werden, weil der Typ des Parameters $(\texttt{Just}\ y)_{|\texttt{Maybe (Maybe (Maybe Nat))}|}$ von `wrap` nicht zu dem Typ der Variablen $y_{|\texttt{(Maybe (Maybe Nat))}|}$ paßt. Ein erneutes Typchecken vom Term des Knotens `d` brächte hier die Lösung, da er dann diese Form hätte:

$$(\emptyset, (\texttt{wrap}_{|a \to \texttt{Nat}|}\ y_{|a|})_{|\texttt{Nat}|}) =$$
$$\text{typeChecker}_{\mathsf{HP}}((\texttt{wrap}_{|\texttt{Maybe (Maybe Nat)} \to \texttt{Nat}|}\ y_{|\texttt{(Maybe (Maybe Nat))}|})_{|\texttt{Nat}|})$$

Da Knoten `e` direkt aus Knoten `d` durch eine Auswertung entstanden wäre, hätte Knoten `e` die Form:

$$(\emptyset, (\texttt{wrap}_{|\texttt{Maybe}\ a \to \texttt{Nat}|}\ (\texttt{Just}\ y)_{|\texttt{Maybe}\ a|})_{|\texttt{Nat}|})$$

Anschließend könnte durch die Instantiierung eine Generalisierungskante zwischen diesen Knoten gezogen werden.

[1] Die Terme sind nicht vollständig mit Typannotation versehen, die hier irrelevanten wurden weggelassen um die wesentlichen Typen besser hervorzuheben.

Ein weiterer Grund, eine Generalisierung für einen Term durchzuführen, ist gegeben, wenn der Term eines Knotens eine Instanz eines Terms eines anderen Knotens echt enthält. In Abbildung 6.6 ist der Narrowing-Graph, in dem solch eine Situation auftritt, für den Startterm $\mathtt{g}\ x$ und das Haskellprogramm aus Beispiel 6.4 zu sehen.

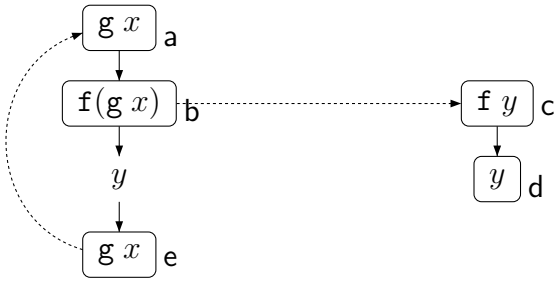

Abbildung 6.6: Narrowing-Graph zu Beispiel 6.4 und Startterm $\mathtt{g}\ x$.

Beispiel 6.4

$$\begin{aligned} \mathtt{f}\ \ y &= y \\ \mathtt{g}\ \ x &= \mathtt{f}\ (\mathtt{g}\ x) \end{aligned}$$

Für den Knoten b kommt normalerweise nur eine weitere Auswertung in Betracht, da er nicht n-fach geschachtelt ist. Nach einigen Auswertungen würde der Term $\mathtt{f}(\mathtt{g}\ \ldots(\mathtt{g}\ x)\ldots)$ in einem Knoten stehen und so würde eine n-fache Schachtelung erkannt. Da aber in Knoten a die Behandlung des Subterms $\mathtt{g}\ x$ von Knoten b begonnen hat, ist es möglich, diese schon auszunutzen, indem dieser Subterm durch die frische Variable y ersetzt und eine Erweiterung um den Knoten c vorgenommen wird. Durch eine Instantiierung von Knoten b mit Knoten c entsteht Knoten e, der jetzt wiederum eine Instantiierung mit Knoten a eingeht und so dessen Behandlung ausnutzt.

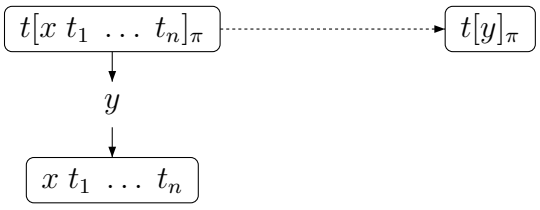

Abbildung 6.7: Behandlung einer freien Applikation

Eine ähnliche Generalisierung wird nötig, wenn der Redex eines Knotens $(\underline{cs}, t)$ eine freie Applikation ist. Das heißt, für den Redex an Stelle π des Terms t gilt

$t|_\pi = x\ t_1\ \ldots\ t_n$, wobei $x \in \mathsf{V_L}$ und $n > 0$ gilt. Auf den Knoten $(\underline{cs}, t)$ kann so keine Transformation angewendet werden. Eine Auswertung von $x\ t_1\ \ldots\ t_n$ ist nicht möglich, da x keine Funktion ist, und eine Fallunterscheidung ist ebenfalls unmöglich, da unklar ist, zu welchem Term der Term $x\ t_1\ \ldots\ t_n$ ausgewertet wird. Es muß aber eine Transformation angewendet werden, damit der Narrowing-Graph geschlossen werden kann. Ein Ausweg ist die Erweiterung um den Knoten $(\underline{cs}, t[y]_\pi)$, wobei y eine frische Variable ist. Nun kann eine Instantiierung auf Knoten $(\underline{cs}, t)$ angewendet werden, wobei eine Generalisierungskante zu dem neuen Knoten $(\underline{cs}, t[y]_\pi)$ entsteht. In Abbildung 6.7 wird diese Situation verdeutlicht.

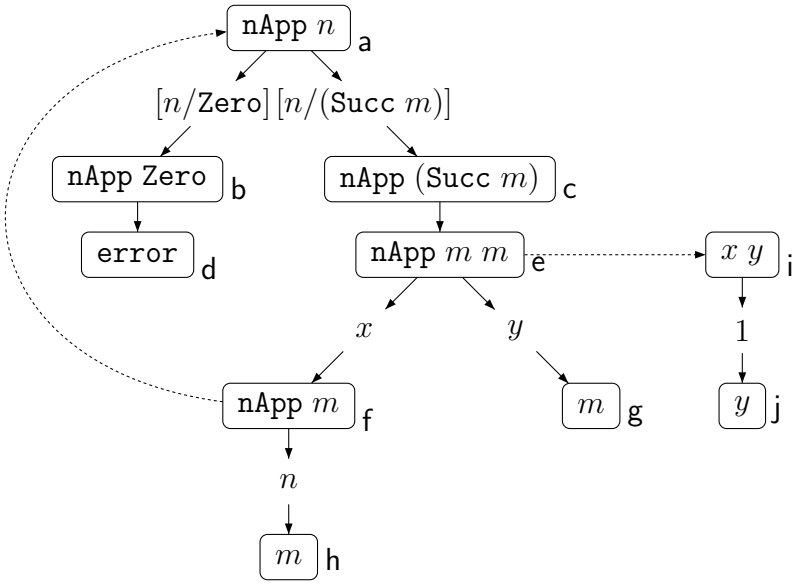

Abbildung 6.8: Narrowing-Graph zu Beispiel 6.5 und Startterm `nApp` n.

Ein anderes subtiles Problem tritt auf, wenn im Laufe einer Auswertung ein Term an immer mehr Terme appliziert wird und dieser dadurch wächst, wie es innerhalb der Funktion `nApp` für den Term `nApp` m aus dem Beispiel 6.5 geschieht.

Beispiel 6.5

```
data Nat    =  Succ Nat | Zero

nApp  :: Nat → a
nApp  (Succ m)  =  nApp m m
```

Beim Erstellen des Narrowing-Graphen aus Abbildung 6.8 für den Startterm `nApp` n fällt auf, daß nach einer Fallunterscheidung und Auswertung der neue Knoten `e` der Form $(\emptyset, \texttt{nApp}\ m\ m)$ entsteht. Alle bis jetzt gebildeten Knoten `a`,

b, c und d bieten keine Möglichkeit einer Instantiierung für den Knoten e. Eine weitere Auswertung, die für Knoten e nur in Frage käme, bringt keine Besserung, es würde später nur ein Knoten der Form $(\emptyset, \texttt{nApp}\ l\ l\ (\texttt{Succ}\ l))$ entstehen, für den wieder keine Knoten zur Instantiierung bereit stehen. Die Lösung ist eine Instantiierung mit Knoten i, obwohl dessen Aufnahme in den Narrowing-Graph völlig unbegründet erscheint, da dieser in keinem Zusammenhang mit dem Startterm steht. Durch eine Instantiierung von e nach i sinkt aber die Anzahl der Parameter und die Instantiierung von Knoten f nach a wird ermöglicht, so daß der Narrowing-Graph geschlossen werden kann. Natürlich kann eine Instantiierung von jedem Knoten, der eine Applikation als Term enthält, zu dem Knoten $(\emptyset, x\ y)$ durchgeführt werden, das aber bringt wiederum wenig, da die so neu entstehenden Knoten nur wieder mit der Variablenexpansion um einen Parameter erweitert werden müßten. Zur Vermeidung dieses Problems müssen Zusatzbedingungen für eine Instantiierung mit Knoten $(\emptyset, x\ y)$ erfüllt sein. Die Überprüfung, ob die Stelligkeit der Funktionsvariablen im Kopf durch die Anzahl der Parameter überschritten wird, erscheint sinnvoll. Es ist zu beachten, daß die Stelligkeit der Funktionsvariablen nicht die Stelligkeit der Typannotation ist, sondern die des im Haskellprogramm festgelegten Typschemas zu dieser Funktionsvariablen. Das heißt, es gilt $\text{arity}(\texttt{nApp}) = 1$, unabhängig von dem Term $(\texttt{nApp}_{|\texttt{Nat}\to(\texttt{Nat}\to a)|}\ m_{|\texttt{Nat}|}\ m_{|\texttt{Nat}|})_{|a|}$, obwohl die Typannotation von nApp dort $\texttt{Nat} \to (\texttt{Nat} \to a)$ ist und so die Stelligkeit 2 hat. Die Stelligkeit der Funktionsvariablen nApp ist nämlich im Haskellprogramm aus dem Beispiel 6.5 durch das Typschema $\emptyset \Rightarrow \texttt{Nat} \to a$ mit der Stelligkeit 1 gegeben. Die Typannotation $\texttt{Nat} \to (\texttt{Nat} \to a)$ ist hier nur eine gültige Instanz des Typs aus dem Typschema $\emptyset \Rightarrow \texttt{Nat} \to a$ der Funktionsvariablen nApp.

Diese oben vorgestellten einzelnen Ideen sind in dem Algorithmus BuildGraph zusammengefaßt, welcher den Algorithmus ExtensionVia einige Male aufruft. Er startet mit der Erstellung der Startkonfiguration (Zeile: 3) des Narrowing-Graphen für einen übergebenen Startterm und dessen Klassenbedingungen. Danach werden in der **while**-Schleife (Zeile: 4) auf die noch unmarkierten Knoten passende Transformationen angewendet (Zeilen: 8, 16, 34, 36 und 38), es sei denn, der Term des aktuellen Knotens ist eine Instanz eines anderen Terms im Narrowing-Graph. In diesem Fall wird dort eine Instantiierung vorgenommen (Zeile: 10). Falls die Funktionsvariable im Kopf des aktuellen Terms mit zu vielen Parametern versehen ist, wird der letzte abgespalten (Zeile: 18). Eine Erweiterung um einen Knoten, dessen Term an einer Position eine frische Variable enthält und sonst dem aktuellen Term gleicht, wird vorgenommen, wenn der aktuelle Term eine Instanz eines anderen Terms echt enthält (Zeile: 26). Es wird ebenfalls generalisiert und instantiiert, wenn das Generalisierungskriterium der n-fachen Schachtelung (Zeile: 19) zutrifft oder freie Applikationen vorhanden sind (Zeile: 11).

Input : Startterm s, Schachtelungstiefe m, Applikationsüberhang k
Output : Narrowing-Graph NG

$\mathsf{NG} := \langle \mathsf{Q}, \mathsf{M}, \emptyset, \emptyset, \bot, \bot, \bot \rangle$, wobei $\mathsf{M} := \bot$ und $\mathsf{Q} := \emptyset$;
Setze NG auf Startkonfiguration für s;
Erweiterung um Knoten $(\emptyset, x\ y)$;
while $\mathrm{Dom}(\mathsf{M}) \neq \mathsf{Q}$ **do**
  $\Delta := \{q \in \mathsf{Q} \mid \mathsf{M}(q) = \underline{\mathrm{case}} \ \vee\ \mathsf{M}(q) = \underline{\mathrm{varexp}} \ \vee\ \mathsf{M}(q) = \underline{\mathrm{eval}}\}$;
  Suche Knoten $(\underline{cs}, u) \in \mathsf{Q}$ mit $(\underline{cs}, u) \notin \mathrm{Dom}(\mathsf{M})$ und Vorgänger q';
  if $u = h\ u_1\ \ldots\ u_n$, $h \in \underline{\mathsf{Cons_D}} \cup \mathsf{V_L}$ *und* $\mathsf{M}(q') \neq \underline{\mathrm{case}}$ **then**
    Parameteraufteilung von $(\underline{cs}, u)$;
  else if u' *matcht* u *für* $(\underline{cs}', u') \in \Delta$ *und* $\mathsf{M}(q') \neq \underline{\mathrm{case}}$ **then**
    Instantiierung von $(\underline{cs}, u)$ nach $(\underline{cs}', u')$;
  if $(x\ t_1\ \ldots\ t_n) \lhd u$ *mit* $x \in \mathsf{V_L}$, $n > 0$ *und* $(\mathsf{M}(q') \neq \underline{\mathrm{case}})$ **then**
    $\pi_1, \ldots, \pi_m :=$ allen Stellen vom Term u mit freien Applikation;
    $y_1, \ldots, y_m :=$ frische Variablen;
    $\mathsf{NG} := \mathrm{ExtensionVia}(\mathsf{NG}, (\underline{cs}, u[y_1]_{\pi_1} \ldots [y_m]_{\pi_m}))$;
  else if $u = (h\ u_1\ \ldots\ u_n)_{|\tau \to \tau'|}$, $h \in \mathsf{V_F}$ *und* $n < (\mathrm{arity}(h) + k)$ **then**
    Variablenexpansion von $(\underline{cs}, u)$;
  else if $u = h\ u_1\ \ldots\ u_n$, $h \in \mathsf{V_F}$ *und* $n > (\mathrm{arity}(h) + k)$ **then**
    Instantiierung von $(\underline{cs}, u)$ nach $(\emptyset, x\ y)$
  else if $\mathrm{nestedPos}(u, m) \neq \epsilon$ **then**
    $\pi := \mathrm{nestedPos}(u, m)$;
    $h\ t_1\ \ldots\ t_n := u|_\pi$;
    $x_1, \ldots, x_n :=$ frische Variablen;
    $\mathsf{NG} := \mathrm{ExtensionVia}(\mathsf{NG}, (\underline{cs}, u[h\ x_1\ \ldots\ x_n]|_\pi))$;
  else
    foreach *Teilterm* t *von* u *an Stelle* $\pi \neq \epsilon$ **do**
      if (u' *matcht* t *für ein* $(\underline{cs}', u') \in \Delta$)
       $\vee$ ($t = y\ t_1\ \ldots\ t_n$ *mit* $y \in \mathsf{V_L}$) **then**
        $y :=$ frische Variable;
        $\mathsf{NG} := \mathrm{ExtensionVia}(\mathsf{NG}, (\underline{cs}, u[y]_\pi))$;
        continue 4;
      endif
    endfch
    else if $\mathrm{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u) = \underline{\mathrm{error}}\ x$ *mit* $x \in \mathsf{V_L}$ **then**
      Fallunterscheidung von $(\underline{cs}, u)$ für x;
    else if $\mathrm{evaluate}^{\mathsf{EXT}}_{\mathsf{HP}}(u) = \underline{\mathrm{error}}\ v$ *mit* $v \in \mathsf{V_F}$ **then**
      Fallunterscheidung von $(\underline{cs}, u)$ für Typ von v;
    else
      Auswertung von $(\underline{cs}, u)$;
    endif
  endif
endw
return NG

Algorithmus 2 : BuildGraph

Input : Narrowing-Graph NG, Knoten $(\underline{cs}, u)$
Output : Narrowing-Graph NG
1 $(\underline{cs}', u') :=$ typeChecker$_{\mathsf{HP}}(u)$;
2 Erweiterung um Knoten $(\underline{cs}', u')$;
3 **foreach** $(\underline{cs}'', u'') \in$ Dom(generalisation) **do**
4 **if** u' *matcht* u'' **then**
5 **foreach** $q \in \mathsf{Q}$ *mit* generalisation$(q) = (\underline{cs}'', u'')$ **do**
6 Hebe Instantiierung von q nach $(\underline{cs}'', u'')$ auf;
7 Instantiierung von q nach $(\underline{cs}', u')$;
8 **endfch**
9 **endif**
10 **endfch**
11 Reduktion des Graphen auf Knoten, welche vom Startterm erreichbar sind $((\underline{cs}', u')$ und $(\emptyset, x\ y)$ werden dabei nicht gelöscht);
12 **return** NG;

Algorithmus 3 : ExtensionVia

Der Algorithmus ExtensionVia ruft den Typchecker für den Term des Eingabeknotens auf und reduziert danach dessen Klassenbedingung. Durch eine Erweiterung wird dann der neue Knoten in den Narrowing-Graph eingebracht (Zeile: 2). Nun wird für jeden Knoten, aus dem eine Generalisierungskante entspringt, geprüft, ob dieser eine Instanz von dem neuen Knoten ist. Trifft dies zu, wird dessen Generalisierungskante gelöscht (Zeile: 6) und eine neue zu dem neuen Knoten erstellt (Zeile: 7). Nach Umleitung von einigen Generalisierungskanten, ist es möglich, daß einige Knoten die Verbindung zum Startknoten verloren haben, welche dann entfernt werden (Zeile: 11), denn diese sind nicht mehr für den Nachweis nötig, daß der Startknoten NF-terminiert. Der neue Knoten wird aber in jedem Fall beibehalten, genauso wie der Knoten $(\emptyset, x\ y)$.

Satz 6.1 (Algorithmus BuildGraph **schließt Narrowing-Graph)**
Der Algorithmus BuildGraph *schließt einen Narrowing-Graph in Startkonfiguration für einen Startterm*

Beweisidee:
In Zeile 4 von Algorithmus BuildGraph *startet eine* **while***-Schleife, die nur verlassen werden kann, wenn auf jeden Knoten eine Transformation angewendet wurde. Da nur gültige Transformationen angewendet wurden, ist, wenn der Algorithmus* BuildGraph *verlassen wird, ein geschlossener Narrowing-Graph entstanden. Er terminiert für den Fall, daß keine neuen Knoten mehr entstehen und alle Knoten markiert sind. Wenn immer mehr Knoten entstehen und diese nicht durch eine Instantiierung markiert werden können, muß in einem dieser Terme*

ein n-fache Schachtelung oder eine ständige Applikation von Parametern auftreten, weil die Signatur eines Haskellprogramms endlich ist. Beide Fälle werden aber abgefangen, so daß BuildGraph *terminiert.*

In dem AProVE-Projekt existiert ein Prozessor-Interface, wie es von Martin Mertens in seiner Diplomarbeit mit dem Thema „Modularity, Strategies and Proof Management in Automated Program Verification“ beschrieben ist [Mer05]. Zu diesem Interface wurde ein Prozessor erstellt, der einen Narrowing-Graph mit Hilfe der Transformationen, wie sie in Kapitel 4 vorgestellt worden sind, aufbaut. Dabei geht er genauso vor wie Algorithmus BuildGraph und erzeugt so einen geschlossenen Narrowing-Graph. Aus diesem erzeugten geschlossenen Narrowing-Graph ließt er daraufhin DP-Probleme ab, wie es im Kapitel 5 beschrieben ist, und reicht diese als Ergebnis an andere Prozessoren innerhalb des AProVE-Projekts weiter, die deren Endlichkeit prüfen.

Kapitel 7

Zusammenfassung

Das Ziel dieser Arbeit war es, eine Terminierungsanalyse von Haskellprogrammen zu entwickelt und diese als ein Modul innerhalb von AProVE zu implementierten.

Es wurden zunächst mehrere Reduktionen für Haskellprogramme entwickelt und deren Korrektheit und Vollständigkeit nachgewiesen. Diese Reduktionen wurden zu dem Algorithmus Reduce zusammengefaßt, der ein dem Haskell-98-Report ([J+98]) entsprechendes Haskellprogramm in ein vereinfachtes Haskellprogramm übersetzt.

Auf der Basis vereinfachter Haskellprogramme wurden Narrowing-Graphen zu Starttermen eingeführt, welche die Termination Tableaux aus [PSS97] um Typen und Klassenbedingungen erweitern. Außerdem lassen sich in einem Narrowing-Graph beliebige verschränkt-rekursive Zyklen behandeln, welche in den Termination Tableaux entweder aufgelöst werden müssen oder nicht behandelt werden können. Sven Eric Panitz und Manfred Schmidt-Schauß probieren, wie sie in [PSS97] beschreiben, eine fundierte Ordnung für jeden Zyklus in einem Termination Tableau zu finden. Diese Ordnung darf dabei nicht auf überlappende Zyklen angewendet werden. Olivier Fissore, Isabelle Gnaedig und Hélène Kirchner probieren, ähnliches in ihrem Tool Cariboo [FGK02] zu erreichen. Wir hingegen lösen verschränkt rekursive Zyklen nicht auf, sondern lesen DP-Probleme zu den maximalen starken Zusammenhangskomponenten des Narrowing-Graphen ab, anstatt zu probieren, für einzelne Zyklen direkt eine feste Ordnung zu finden, da DP-Probleme auch verschränkt rekursive Zyklen beinhalten dürfen. Ein auf Narrowing-Graphen basierendes Terminierungskriterium (Abwesenheit von $\overline{\mathsf{NF}}$-Ketten) für Startterme wurde vorgestellt. Zusätzlich wurde im Kapitel 5 eine Methode entwickelt, die Narrowing-Graphen in eine Menge von Dependency-Pair-Problemen überführt. Abschließend wurde die Korrektheit dieser Methode in Abschnitt 5.3 nachgewiesen. Zuletzt wurde der Algorithmus BuildGraph entwickelt, welcher einen geschlossenen Narrowing-Graph zu einem Startterm geschickter aufbaut, als der naive Ansatz aus Abschlußlemma 4.10 und so der Terminie-

rungsanalyse entgegenkommt.

Zusätzlich wurden im Rahmen dieser Diplomarbeit die vorgestellten Methoden in einem Haskell-Modul für das AProVE-Projekt [GTSKF04] verwirklicht. Das Haskell-Modul enthält unter anderem:

- einen Haskell-Parser für die dem Haskell-98-Report entsprechenden Haskellprogramme,
- einen Haskell-Typchecker,
- einen um freie Variablen erweitertern Haskell-Typchecker (wird benutzt im Algorithmus BuildGraph),
- einen Haskellauswerter für Basisterme (wird benutzt im Algorithmus BuildGraph),
- die zehn Vereinfachungs-Reduktionen,
- den Algorithmus Reduce als Strategie-Programm,
- den Algorithmus BuildGraph und
- eine Implementation der Ablesestrategie für DP-Probleme aus Narrowing-Graphen.

Da sich das AProVE-Projekt in einer starken Phase des Redesigns befindet, standen einige zentrale Techniken der Terminierungsanalyse für Dependency-Pair-Probleme noch nicht im neuen Design-Konzept zur Verfügung, während das Haskell-Modul bereits das neue Design-Konzept berücksichtigt. Daher konnte noch keine verläßliche Evaluierung zur Mächtigkeit der hier vorgstellten Methoden erstellt werden. Erfreulicherweise kann AProVE jetzt schon die NF-Terminierung der Startterme der im Kapitel 6 vorgestellten Narrowing-Graphen aus den Abbildungen 6.4 und 6.2 auf den Seiten 138 und 135 nachweisen.

Die Dependency-Pair-Methode, wie in [GTSK05] vorgstellt, beinhaltet die Möglichkeit, ein Dependency-Pair-Problem auf Abwesenheit *minimaler* Ketten zu prüfen. Diese Überprüfung ist wesentlich einfacher und würde das gesamte Verfahren stärker machen. Es ist noch offen, ob die Abwesenheit minimaler unendlicher Ketten ausreicht, um die Abwesenheit von $\overline{\mathsf{NF}}$-Ketten eines Narrowing-Graphen nachzuweisen. Ebenso nützlich wäre es, wenn die Abwesenheit von unendlichen *innermost* Ketten genügen würde, um die Abwesenheit von $\overline{\mathsf{NF}}$-Ketten zu garantieren, aber diese Frage ist ebenfalls ungeklärt. Ebenso kann auch in der anderen Richtung untersucht werden, ob die Techniken, die auf die abgelesenen DP-Probleme angewendet werden, nicht so modifiziert werden könnten, daß diese die Haskellauswertungsstrategie ebenfalls berücksichtigen. Dabei müßte im

einzelnen für jede Technik geprüft werden, wie dieser Ansatz umgesetzt werden kann. Genauso ist offen, wie ein Nachweis, ob ein Startterm *nicht* NF-terminiert, mit Hilfe der Narrowing-Graphen möglich ist. Es bleibt also zu klären, ab wann das Terminierungskriterium vollständig ist. Es wäre zum Beispiel vollständig für den Narrowing-Graph des Startterms $(\emptyset, \texttt{from}\ i)$, wie er als Teilgraph unterhalb des Knotens `k` in der Abbildung 6.1 auf Seite 63 zu sehen ist. Wenn ein Zyklus sich über eine Variablenexpansionskante erstreckt, ist durch die grobe Abschätzung der Variablenexpansion nicht mit einer Erhaltung der Vollständigkeit zu rechnen, wenn die Werte, die an die neue Variable gebunden werden können, innerhalb des Terms vom variablenexpandierten Knoten während einer Auswertung eine Verwendung finden. Ähnliches gilt für einen Zykel, der über eine Teiltermkante einer Instantiierung läuft. Denn dabei wurde implizit abgeschätzt, daß die NF-Terminierung dieses Teilterm-Nachfolgers für die des instantiierten Vorgängers relevant ist. Möglicherweise ignoriert der Term des Vorgängers den Teilterm seines Teilterm-Nachfolgers, so daß ein unnötiger Zyklus gebildet worden ist. An dieser Stelle könnte eine Striktheitsanalyse weiterhelfen, welche uns sagt, ob ein Teilterm eines Terms für eine Auswertung relevant ist oder nicht. Wenn Teilterme für eine Auswertung nicht relevant sind, müssen zu diesen keine Teilterm-Nachfolger gebildet werden. Außerdem erhält eine Erweiterung um einen Knoten mit einem generalisierten Term, wie es in dem Narrowing-Graph in Abbildung 6.4 auf Seite 138 geschehen ist, im allgemeinen nicht die Vollständigkeit, da für einen weiteren Term, welcher nicht direkt mit dem Startterm zusammenhängt, die NF-Terminierung nachgewiesen werden muß.

Anhang A

Int-Peano-Implementierung

```
data Int = Zero | Succ Int | Pred Int

primPlusInt  :: Int -> Int -> Int
primPlusInt Zero    Zero      = Zero
primPlusInt Zero    (Succ y)  = Succ y
primPlusInt Zero    (Pred y)  = Pred y
primPlusInt (Succ x) (Zero)   = Succ x
primPlusInt (Succ x) (Succ y) = Succ (Succ (primPlusInt x y))
primPlusInt (Succ x) (Pred y) = primPlusInt x y
primPlusInt (Pred x) (Zero)   = Pred x
primPlusInt (Pred x) (Succ y) = primPlusInt x y
primPlusInt (Pred x) (Pred y) = Pred (Pred (primPlusInt x y))

primMinusInt :: Int -> Int -> Int
primMinusInt Zero    Zero      = Zero
primMinusInt Zero    (Succ y)  = Pred (primNegInt y)
primMinusInt Zero    (Pred y)  = Succ (primNegInt y)
primMinusInt (Succ x) (Zero)   = Succ x
primMinusInt (Succ x) (Succ y) = primMinusInt x y
primMinusInt (Succ x) (Pred y) = Succ (Succ (primMinusInt x y))
primMinusInt (Pred x) (Zero)   = Pred x
primMinusInt (Pred x) (Succ y) = Pred (Pred (primMinusInt x y))
primMinusInt (Pred x) (Pred y) = primMinusInt x y

primMulInt   :: Int -> Int -> Int
primMulInt Zero    Zero      = Zero
primMulInt Zero    (Succ y)  = Zero
primMulInt Zero    (Pred y)  = Zero
primMulInt (Succ x) (Zero)   = Zero
primMulInt (Succ x) (Succ y) = primPlusInt (primMulInt x (Succ y)) (Succ y)
primMulInt (Succ x) (Pred y) = primPlusInt (primMulInt x (Pred y)) (Pred y)
primMulInt (Pred x) (Zero)   = Zero
primMulInt (Pred x) (Succ y) = primMinusInt (primMulInt x (Succ y)) (Succ y)
primMulInt (Pred x) (Pred y) = primMinusInt (primMulInt x (Pred y)) (Pred y)

primNegInt   :: Int -> Int
primNegInt Zero     = Zero
primNegInt (Succ x) = Pred (primNegInt x)
primNegInt (Pred x) = Succ (primNegInt x)

primQuotInt :: Int -> Int -> Int
primQuotInt Zero     Zero     = error ""
primQuotInt Zero     (Succ y) = Zero
primQuotInt Zero     (Pred y) = Zero
primQuotInt (Succ x) Zero     = error ""
```

```
primQuotInt (Succ x) (Succ y) = primDivIntS (Succ x) (Succ y)
primQuotInt (Succ x) (Pred y) = primNegInt (primDivIntS (Succ x) (primNegInt (Pred y)))
primQuotInt (Pred x) Zero     = error ""
primQuotInt (Pred x) (Succ y) = primNegInt (primDivIntS (primNegInt (Pred x)) (Succ y))
primQuotInt (Pred x) (Pred y) = primDivIntS (primNegInt (Pred x)) (primNegInt (Pred y))

primDivInt :: Int -> Int -> Int
primDivInt Zero     Zero     = error ""
primDivInt Zero     (Succ y) = Zero
primDivInt Zero     (Pred y) = Zero
primDivInt (Succ x) Zero     = error ""
primDivInt (Succ x) (Succ y) = primDivIntS (Succ x) (Succ y)
primDivInt (Succ x) (Pred y) = primNegInt (primDivIntP (Succ x) (primNegInt (Pred y)))
primDivInt (Pred x) Zero     = error ""
primDivInt (Pred x) (Succ y) = primNegInt (primDivIntP (primNegInt (Pred x)) (Succ y))
primDivInt (Pred x) (Pred y) = primDivIntS (primNegInt (Pred x)) (primNegInt (Pred y))

primDivIntS :: Int -> Int -> Int
primDivIntS Zero     Zero      = error ""
primDivIntS (Succ x) Zero      = error ""
primDivIntS (Succ x) (Succ y)  = if (primGEqIntS x y) then
                                      Succ (primDivIntS (primMinusIntS x y) (Succ y))
                                      else Zero
primDivIntS Zero     (Succ x)  = Zero

primDivIntP :: Int -> Int -> Int
primDivIntP Zero     Zero      = error ""
primDivIntP (Succ x) Zero      = error ""
primDivIntP (Succ x) (Succ y)  = Succ (primDivIntP (primMinusIntS x y) (Succ y))
primDivIntP Zero     (Succ x)  = Zero

primMinusIntS :: Int -> Int -> Int
primMinusIntS (Succ x) (Succ y) = primMinusIntS x y
primMinusIntS Zero     (Succ y) = Zero
primMinusIntS x        Zero     = x

primGEqIntS :: Int -> Int -> Bool
primGEqIntS (Succ x) Zero = True
primGEqIntS (Succ x) (Succ y) = (primGEqIntS x y)
primGEqIntS Zero (Succ x) = False
primGEqIntS Zero Zero = True

primModIntS :: Int -> Int -> Int
primModIntS Zero     Zero      = error ""
primModIntS (Succ x) Zero      = error ""
primModIntS (Succ x) (Succ y)  = if (primGEqIntS x y) then
                                      (primModIntS (primMinusIntS x y) (Succ y))
                                      else (Succ x)
primModIntS Zero     (Succ x)  = Zero

primModIntP :: Int -> Int -> Int
primModIntP Zero     Zero      = error ""
primModIntP (Succ x) Zero      = error ""
primModIntP (Succ x) (Succ y)  = if (primGEqIntS x y) then
                                      (primModIntP (primMinusIntS x y) (Succ y))
                                      else (primMinusIntS y x)
primModIntP Zero     (Succ x)  = Zero

primRemInt  :: Int -> Int -> Int
primRemInt Zero     Zero     = error ""
primRemInt Zero     (Succ y) = Zero
primRemInt Zero     (Pred y) = Zero
primRemInt (Succ x) Zero     = error ""
```

```
primRemInt (Succ x) (Succ y) = primModIntS (Succ x) (Succ y)
primRemInt (Succ x) (Pred y) = primModIntS (Succ x) (primNegInt (Pred y))
primRemInt (Pred x) Zero     = error ""
primRemInt (Pred x) (Succ y) = primNegInt (primModIntS (primNegInt (Pred x)) (Succ y))
primRemInt (Pred x) (Pred y) = primNegInt (primModIntS (primNegInt (Pred x))
                                       (primNegInt (Pred y)))

primModInt  :: Int -> Int -> Int
primModInt Zero     Zero     = error ""
primModInt Zero     (Succ y) = Zero
primModInt Zero     (Pred y) = Zero
primModInt (Succ x) Zero     = error ""
primModInt (Succ x) (Succ y) = primModIntS (Succ x) (Succ y)
primModInt (Succ x) (Pred y) = primNegInt (primModIntP (Succ x) (primNegInt (Pred y)))
primModInt (Pred x) Zero     = error ""
primModInt (Pred x) (Succ y) = primModIntP (primNegInt (Pred x)) (Succ y)
primModInt (Pred x) (Pred y) = primNegInt (primModIntS (primNegInt (Pred x))
                                       (primNegInt (Pred y)))

primCmpInt     :: Int -> Int -> Ordering
primCmpInt Zero     Zero     = EQ
primCmpInt Zero     (Succ y) = LT
primCmpInt Zero     (Pred y) = GT
primCmpInt (Succ x) (Zero)   = GT
primCmpInt (Succ x) (Succ y) = primCmpInt x y
primCmpInt (Succ x) (Pred y) = GT
primCmpInt (Pred x) (Zero)   = LT
primCmpInt (Pred x) (Succ y) = LT
primCmpInt (Pred x) (Pred y) = primCmpInt x y

primEqInt      :: Int -> Int -> Bool
primEqInt Zero     Zero     = True
primEqInt Zero     (Succ y) = False
primEqInt Zero     (Pred y) = False
primEqInt (Succ x) (Zero)   = False
primEqInt (Succ x) (Succ y) = primEqInt x y
primEqInt (Succ x) (Pred y) = False
primEqInt (Pred x) (Zero)   = False
primEqInt (Pred x) (Succ y) = False
primEqInt (Pred x) (Pred y) = primEqInt x y

primEvenInt :: Int -> Bool
primEvenInt Zero            = True
primEvenInt (Succ Zero)     = False
primEvenInt (Succ (Succ x)) = primEvenInt x
primEvenInt (Succ (Pred x)) = primEvenInt x
primEvenInt (Pred Zero)     = False
primEvenInt (Pred (Succ x)) = primEvenInt x
primEvenInt (Pred (Pred x)) = primEvenInt x
```

Literaturverzeichnis

[AG00] Thomas Arts and Jürgen Giesl. Terminiation of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[DJH02] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification of the Haskell 98 Module System. In *Proceedings of Haskell '02*, pages 17–28, 2002.

[FGK02] Olivier Fissore, Isabelle Gnaedig, and Hélène Kirchner. CARIBOO: An induction based proof tool for termination with strategies. In *Proceedings of PPDP '02*, pages 62–73, 2002.

[GTSK05] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proceedings of LPAR '04*, pages 301–331, 2005.

[GTSKF04] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proceedings of RTA '04*, pages 210–220, 2004.

[Has04] Christian A. Haselbach. Transformation Techniques to Verify Imperative and Functional Programs. Diploma thesis, RWTH Aachen, Germany, 2004.

[J+98] S. Peyton Jones et al. Haskell 98 report. Technical report, `http://www.haskell.org/`, 1998.

[Jon99] Mark P. Jones. Typing Haskell in Haskell*. Technical report, `http://www.cse.ogi.edu/~mpj/pubs/thih.html`, 1999.

[JR98] Mark P. Jones and Alastair Reid. `Hugs` 98. Technical report, `http://www.haskell.org/hugs/`, 1998.

[Käu05] Christian Käunicke. Automatic Termination Analysis of Logic Programs. Diploma thesis, RWTH Aachen, Germany, 2005.

[Mer05] Martin Mertens. Modularity, Strategies and Proof Management in Automated Program Verification. Diploma thesis, RWTH Aachen, Germany, 2005.

[Pan96] S. E. Panitz. Termination Proofs for a Lazy Functional Language by Abstract Reduction. Frank report 06, Fachbereich Informatik, Universität Frankfurt, Germany, June 1996.

[PSS97] Sven Eric Panitz and Manfred Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proceedings of SAS '97*, pages 345–360, 1997.

Index