4.4.2 Lazy Narrowing-Strategien

Strikte Strategien haben die generelle Eigenschaft, dass alle Teilausdrücke ausgewertet werden. Wie wir wissen, hat dies den potenziellen Nachteil, dass strikte Strategien

- bei nichtterminierenden Funktionen unvollständig
- im Allgemeinen nicht optimal

sind.

Dagegen wertet eine lazy Strategie Teilterme nur dann aus, falls es notwendig ist, sodass man damit ein potenziell besseres Verhalten erzielen kann. Was bedeutet allerdings „falls es notwendig ist“?
Eine präzise formale Definition ist schon bei Reduktionsstrategien nicht einfach und bei Narrowing-Strategien, wo wir noch die Belegung freier Variablen berücksichtigen müssen, noch schwieriger zu definieren. Aus diesem Grund versuchen wir als ersten Ansatz eine einfache Definition, indem wir „lazy“ durch „outermost“ ersetzen.

Definition 4.7 *Ein Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ heißt **outermost**, falls für alle Narrowing-Schritte $t \leadsto_{p',\sigma'} t''$ gilt: $p' \not< p$ (p' nicht über p). In analoger Weise können wir auch **leftmost outermost** definieren als: $p' \not< p$ und p' nicht links von p.*

Leider zeigt sich, dass Outermost Narrowing im Allgemeinen unvollständig ist (auch wenn es nur eine outermost Position gibt, was im Gegensatz zur Reduktion steht!).

Beispiel: Betrachten wir das folgende Termersetzungssystem:

$$\begin{array}{rcl} f(0,0) & \to & 0 \\ f(s(x),0) & \to & s(0) \\ f(x,s(y)) & \to & s(0) \end{array}$$

Dieses Termersetzungssystem ist konfluent, terminierend und total definiert. Betrachten wir nun die Gleichung:

$$f(f(i,j),k) \doteq 0$$

Mittels Innermost Narrowing kann die folgende Lösung berechnet werden:

$$\begin{array}{lll} f(f(i,j),k) \doteq 0 & \leadsto_{1\cdot 1,\{i\mapsto 0, j\mapsto 0\}} & f(0,k) \doteq 0 \\ & \leadsto_{1,\{k\mapsto 0\}} & 0 \doteq 0 \end{array}$$

Die hierbei berechnete Lösung ist: $\{i \mapsto 0, j \mapsto 0, k \mapsto 0\}$

Mittels Outermost Narrowing wird allerdings *keine* Lösung berechnet, da hier nur ein Outermost Narrowing-Schritt möglich ist:

$$f(f(i,j),k) \doteq 0 \quad \leadsto_{1,\{k\mapsto s(y)\}} \quad s(0) \doteq 0$$

Um die Vollständigkeit von Outermost Narrowing sicherzustellen sind weitere Restriktionen notwendig [Echahed 88, Padawitz 87]. Informell ist **Outermost Narrowing vollständig** (im Sinn von Satz 4.1 von Hullot), falls das Termersetzungssystem konfluent und

terminierend ist und jeder Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ (bzgl. dieser Strategie) **uniform** ist. Letzteres bedeutet, dass Narrowing-Positionen invariant unter Normalforminstanzierung sind, d.h. $\forall\varphi$ mit $\varphi(x)$ ist in Normalform $\forall x \in \mathcal{D}om(\varphi)$ existiert ein Narrowing-Schritt $\varphi(t) \leadsto_{p,\sigma'} t''$.

Beispiel: Outermost Narrowing ist nicht uniform, da z.B.

$$f(f(i,j),k) \doteq 0 \leadsto_{1\cdot 1,\{k \mapsto s(y)\}} 0 \doteq 0$$

ein Narrowing-Schritt ist, aber ein Narrowing-Schritt für

$$f(f(i,j),0) \doteq 0 \leadsto_{1\cdot 1,\ldots}$$

an der gleichen Position nicht möglich ist.

Dieses theoretische Vollständigkeitskriterium hat einige Nachteile:

- Uniformität ist schwer zu prüfen.
- Uniformität ist eine harte Einschränkung (dies ist eine stärkere Einschränkung als total definiert!).
- Die immer noch geforderte Terminierungseigenschaft verhindert bestimmte funktionale Programmiertechniken.

Aus diesem Grund betrachten wir nun Strategien, bei denen auf die Terminierungsforderung verzichtet wird. Stattdessen fordern wir, dass das Termersetzungssystem konstruktorbasiert und schwach orthogonal ist, was wir auch mit **KB-SO** abkürzen.

Zu beachten ist, dass bei nichtterminierenden Termersetzungssystemen die reflexive Gleichheit $\doteq$ nicht sinnvoll ist (vgl. Kapitel 4.2), sodass wir nachfolgend immer die strikte Gleichheit `=:=` betrachten (vgl. Definition 4.4).

Definition 4.8 (Lazy Narrowing (informell)) *Eine Narrowing-Position ist **lazy**, falls dies die Wurzel ist oder der Wert an dieser Stelle notwendig ist, um eine Regel an einer darüberliegenden lazy Position anzuwenden (die entsprechende formale Definition kann man in [Moreno-Navarro/Rodríguez-Artalejo 92] finden).*

Beispiel: Wir betrachten noch einmal das obige Termersetzungssystem

$$\begin{array}{rcl} f(0,0) & \to & 0 \\ f(s(x),0) & \to & s(0) \\ f(x,s(y)) & \to & s(0) \end{array}$$

und den Term

$$f(f(i,j),k)$$

1. Das äußere Vorkommen von f ist eine lazy Narrowing-Position, weil dies die Wurzelposition des Terms ist.

2. Das innere Vorkommen von f ist auch eine lazy Narrowing-Position, weil diese notwendig ist, um die erste oder zweite Regel an der Wurzelposition anzuwenden.

Definition 4.9 *Ein Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ heißt **lazy**, falls p eine lazy Position in t ist.*

Satz 4.4 ([Moreno-Navarro/Rodríguez-Artalejo 92]) *Lazy Narrowing ist vollständig für KB-SO Termersetzungssysteme und strikte Gleichungen.*

Somit haben wir damit eine Narrowing-Strategie, die das Lösen von Gleichungen mit unendlichen Strukturen ermöglicht. Betrachten wir hierzu als Beispiel das folgende Termersetzungssystem:

$$\begin{array}{rcl} from(n) & \to & n : from(s(n)) \\ take(0, xs) & \to & [] \\ take(s(n), x : xs) & \to & x : take(n, xs) \end{array}$$

Dann können wir die folgende Gleichung über potenziell unendlichen Strukturen lösen:

$$\begin{array}{ll} take(x,\ \underline{from(y)}) \texttt{ =:= } [0] & \\ \leadsto_{\{\}} & \underline{take}(x,\ y : from(s(y))) \texttt{ =:= } [0] \\ \leadsto_{\{x \mapsto s(x_1)\}} & y : take(x_1,\ from(s(y))) \texttt{ }\underline{\texttt{=:=}}\texttt{ } 0 : [] \\ \leadsto_{\{\}} & y \underline{\texttt{=:=}} 0 \texttt{ \&\& } take(x_1,\ from(s(y))) \texttt{ =:= } [] \\ \leadsto_{\{y \mapsto 0\}} & True \texttt{ }\underline{\texttt{\&\&}}\texttt{ } take(x_1,\ from(s(y))) \texttt{ =:= } [] \\ \leadsto_{\{\}} & \underline{take}(x_1,\ from(s(y))) \texttt{ =:= } [] \\ \leadsto_{\{x_1 \mapsto 0\}} & [] \texttt{ =:= } [] \\ \leadsto_{\{\}} & True \end{array}$$

Somit ist die berechnete Lösung: $\{x \mapsto s(0), y \mapsto 0\}$

Dagegen laufen alle strikten Strategien hier in eine Endlosschleife!

Im Gegensatz zur Reduktion in funktionalen Sprachen gibt es für einen Term t nicht nur eine lazy Position, sondern eventuell mehrere lazy Narrowing-Positionen. Dies kann tatsächlich zu dem Problem führen, dass sich manche lazy Narrowing-Schritte als überflüssig bzgl. bestimmter Substitutionen herausstellen, was auf Grund der Interaktion von Variablenbindung und Positionsauswahl passieren kann.

Betrachten wir hierzu das folgende Termersetzungssystem:

$$\begin{array}{rcl} 0 \leq n & \to & True \\ s(m) \leq 0 & \to & False \\ s(m) \leq s(n) & \to & m \leq n \end{array} \qquad \begin{array}{rcl} 0 + n & \to & n \\ s(m) + n & \to & s(m + n) \end{array}$$

Der auszuwertende Ausdruck sei

$$x \leq y + z$$

Dieser Ausdruck hat zwei lazy Narrowing-Positionen: der gesamte Ausdruck bzw. das zweite Argument.

Eine Lazy Narrowing-Ableitung ist daher z.B.

$$x \leq y + z \quad \leadsto_{\{x \mapsto 0\}} \quad True$$

Eine weitere Lazy Narrowing-Ableitung ist z.B.

$$x \leq y + z \quad \leadsto_{\{y \mapsto 0\}} \quad x \leq z \quad \leadsto_{\{x \mapsto s(x_1), z \mapsto 0\}} \quad False$$

Jedoch ist auch dies eine Lazy Narrowing-Ableitung:

$$x \leq y + z \quad \leadsto_{\{y \mapsto 0\}} \quad x \leq z \quad \leadsto_{\{x \mapsto 0\}} \quad True$$

Vergleichen wir nun die erste und die dritte Ableitung:

1. Ableitung: berechnete Lösung: $\{x \mapsto 0\}$, Schritte: 1

3. Ableitung: berechnete Lösung: $\{x \mapsto 0, y \mapsto 0\}$, Schritte: 2

Die 3. Lazy Narrowing-Ableitung enthält somit einen überflüssigen Schritt und berechnet eine zu spezielle Lösung. Somit können wir festhalten:

Lazy Narrowing ist nicht wirklich lazy!

Den „Fehler“, der in der 3. Ableitung gemacht wurde, können wir wie folgt erklären:

> Das Argument $y + z$ wurde ausgewertet, weil die 2. oder 3. Regel für $\leq$ später angewendet werden sollte, d.h. x sollte später eigentlich an $s(x_1)$ (und nicht an 0) gebunden werden. Trotzdem haben wir nachher (weil dies ja ein zulässiger Lazy Narrowing-Schritt ist) x an 0 gebunden und die 1. Regel angewendet.

Wir können daher dieses Problem nur vermeiden, indem wir x frühzeitig an den beabsichtigten Wert binden. Somit wäre eine mögliche Strategie:

- Binde x an 0 oder $s(x_1)$.
- Wende, je nach Bindung, entweder die 1. $\leq$-Regel an (falls x an 0 gebunden wurde) oder werte das Argument $y + z$ aus (falls x an $s(x_1)$ gebunden wurde).

Diese verbesserte Strategie wird auch **Needed Narrowing** genannt [Antoy/Echahed/Hanus 00]. Die Grundidee bei der Auswertung des Aufrufs $f(t_1, \ldots, t_n)$ ist hierbei:

1. Bestimme ein Argument t_i, dessen Wert von allen Regeln für f verlangt wird (bei der Funktion $\leq$ ist dies das 1. Argument, nicht jedoch das 2. Argument).
2. Falls t_i
   - ein Konstruktor $c(\ldots)$ ist: wähle die passende(n) Regel(n) mit diesem Konstruktor und mache damit weiter
   - ein Funktionsaufruf $g(\ldots)$ ist: werte diesen aus
   - eine Variable ist: binde diese (nichtdeterministisch) an die verschiedenen Konstruktoren und mache weiter

Beispiel: Da bei der Funktion $\leq$ der Wert des ersten Arguments verlangt wird, gibt es die folgenden Needed Narrowing-Ableitungen:

$$x \leq y + z \overset{Binde}{\longmapsto}_{\{x \mapsto 0\}} 0 \leq y + z \to_R True$$

$$\overset{Binde}{\longmapsto}_{\{x \mapsto s(x_1)\}} s(x_1) \leq \underbrace{y + z}_{*} \overset{Binde}{\longmapsto}_{\{y \mapsto 0\}} s(x_1) \leq 0 + z \to_R s(x_1) \leq z$$

$$\overset{Binde}{\longmapsto}_{\{z \mapsto 0\}} s(x_1) \leq 0 \to_R False$$

$$\overset{Binde}{\longmapsto}_{\{z \mapsto s \mapsto z_1)\}} \cdots$$

$$\overset{Binde}{\longmapsto}_{\{y \mapsto s(y_1)\}} \cdots$$

$*$ = auswerten, dabei y verlangt

Man beachte, dass die obige 3. Ableitung hier nicht möglich ist! Allerdings ist der Schritt

$$x \leq y + z \rightsquigarrow_{\{x \mapsto s(x_1), y \mapsto 0\}} s(x_1) \leq z$$

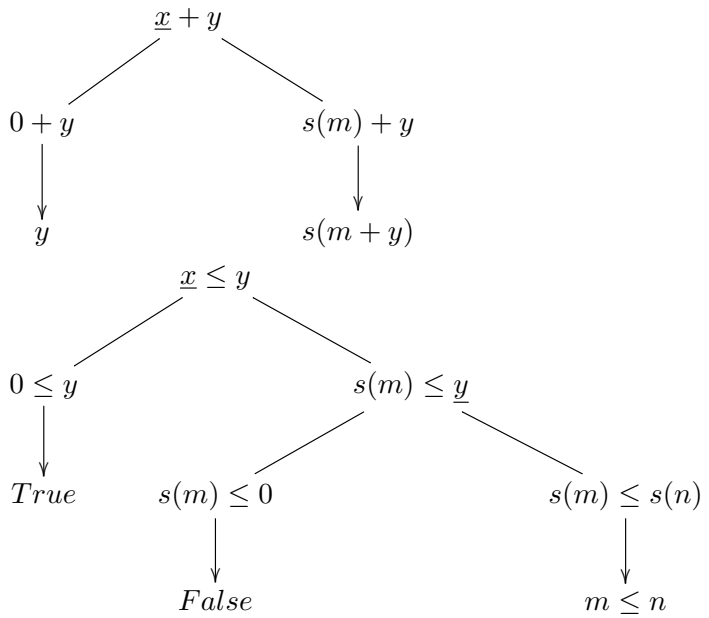
kein Narrowing-Schritt im bisherigen Sinn, da $\{x \mapsto s(x_1), y \mapsto 0\}$ kein **mgu** für den Teilterm $y + z$ und die linke Regelseite ist (sondern dies ist nur ein Unifikator)! Wir sehen also, dass die Berechnung speziellerer Unifikatoren für eine wirkliche „lazy“ Strategie essentiell ist!

Das Problem, was sich stellt, ist:

> Wie findet man die „verlangten“ Argumente?

Generell ist dies ein unentscheidbares Problem für orthogonale Termersetzungssysteme. Allerdings ist dies für induktiv-sequentielle Systeme (vgl. Kapitel 3.2) einfach realisierbar mittels definierender Bäume.

Betrachten wir hierzu die definierenden Bäume für die Funktionen + und $\leq$:



Zum Beispiel legt der definierende Baum für $\leq$ die folgende Strategie zur Auswertung von $t_1 \leq t_2$ fest: Da das 1. Argument t_1 verlangt ist, mache eine Fallunterscheidung über dieses Argument:

1. Falls t_1 ein Funktionsaufruf ist, werte diesen aus und fahre dann mit der gleichen Strategie fort.
2. Falls $t_1 = 0$: wende Regel an
3. Falls $t_1 = s(\ldots)$: Mache eine Fallunterscheidung über das 2. Argument t_2:
   - Falls t_2 ein Funktionsaufruf ist: werte ihn aus
   - Falls $t_2 = 0$ oder $= s(\ldots)$: wende die entsprechende Regel an
   - Falls t_2 eine Variable ist: Binde t_2 an 0 oder $s(x)$ (wobei x eine neue Variable ist) und wende die entsprechende Regel an.
4. Falls t_1 eine Variable ist: Binde t_1 an 0 oder $s(x)$ (wobei x eine neue Variable ist) und mache mit 2. oder 3. weiter.

Formal können wir die Needed Narrowing-Strategie wie folgt definieren. Hierbei ist R ein induktiv-sequentielles Termersetzungssystem.

Definition 4.10 (Needed Narrowing-Schritt) *Sei s ein Term, o eine Position des linkesten äußersten definierten Funktionssysmbols in s (d.h. $s|_o = f(t_1, \ldots, t_n)$) und $\mathcal{T}_f$ ein definierender Baum für f. Die **Needed Narrowing-Strategie** λ berechnet eine Menge von Tripeln der Form $(p, l \to r, \sigma)$, sodass $s \leadsto_{p,l\to r,\sigma} \sigma(s[r]_p)$ ein Narrowing-Schritt ist (beachte allerdings, dass σ nicht unbedingt ein **mgu** ist). λ ist hierbei wie folgt definiert:*

$$\lambda(s) = \{(o \cdot p, l \to r, \sigma) \mid (p, l \to r, \sigma) \in \lambda(s|_o, \mathcal{T}_f)\}$$

wobei $\lambda(t, \mathcal{T})$ die kleinste Tripelmenge mit

$$\lambda(t, \mathcal{T}) \supseteq \begin{cases} \{(\epsilon, l \to r, mgu(t, l))\} & \text{falls } \mathcal{T} = l \to r \\ \lambda(t, \mathcal{T}_i) & \text{falls } \mathcal{T} = branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_n) \\ & \text{und } t \text{ und } pattern(\mathcal{T}_i) \text{ unifizierbar} \\ \{(p \cdot p', l \to r, \sigma \circ \tau)\} & \text{falls } \mathcal{T} = branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_n), \\ & t|_p = g(\ldots) \text{ mit } g \in D,\ \tau = mgu(t, \pi) \\ & \mathcal{T}_g \text{ def. Baum für } g \text{ und } (p', l \to r, \sigma) \in \lambda(\tau(t|_p), \mathcal{T}_g) \end{cases}$$

ist.

Anmerkung: Bei der letzten Alternative ist die Anwendung von τ auf den Teilterm $t|_p$ notwendig wegen eventueller Mehrfachvorkommen von Variablen, wie das folgende Beispiel zeigt:

$$\underline{x} \leq x + x \ \longmapsto_{\{x \mapsto s(x_1)\}}\ s(x_1) \leq s(x_1) + s(x_1) \ \to\ s(x_1) \leq s(x_1 + s(x_1))$$

Hier ist also

$$(2, s(m_1) + n \to s(m_1 + n), \{x \mapsto s(x_1), m_1 \mapsto x_1, n \mapsto s(x_1)\}) \in \lambda(x \leq x + x)$$

Diese formale Definition ist zwar etwas unhandlich, aber tatsächlich ist Needed Narrowing einfach implementierbar analog zum Pattern Matching (denn branch-Knoten entsprechen case-Ausdrücken).

Satz 4.5 ([Antoy/Echahed/Hanus 00]) *Needed Narrowing ist korrekt und vollständig für induktiv-sequentielle Termersetzungssystem bzgl. strikter Gleichungen.*

Needed Narrowing hat darüber hinaus weitere wichtige Eigenschaften:

1. **Optimalität:**
   - Jeder Schritt ist notwendig („needed“), d.h. falls eine Lösung berechnet wird, war kein Schritt in dieser Berechnung überflüssig.
   - Needed Narrowing-Ableitungen haben minimale Länge (dies gilt allerdings nur, falls identische Terme nur einmal berechnet werden, wie dies mittels “sharing” auch bei nicht-strikten funktionalen Sprachen gemacht wird).
   - Minimale Lösungsmenge: Falls σ und σ' Lösungen sind, die durch verschiedene Ableitungen berechnet werden, dann sind σ und σ' unabhängig (d.h. $\sigma(x)$ und $\sigma'(x)$ sind nicht unifizierbar für eine Variable x).
2. **Determinismus:**
   Bei Auswertung variablenfreier Terme ist jeder Schritt deterministisch. In der Konsequenz bedeutet dies, dass funktionale Programme deterministisch ausgewertet werden und Nichtdeterminismus nur bei Vorkommen freier Variablen auftritt.