## 4.4 Narrowing-Strategien

Das Reduktionsprinzip aus funktionalen Sprachen dient dazu, *Werte* zu berechnen. Durch eine Strategie kann man festlegen, an welcher Stelle Reduktionsschritte in einem Term angewendet werden. Mögliche Strategien sind u.a. die strikte und die lazy Auswertung.

Das Narrowing-Verfahren aus logisch-funktionalen Sprachen wird dagegen verwendet, um *Werte und Lösungen* (d.h. Belegungen freier Variablen) zu berechnen. In diesem Kapitel wollen wir erläutern, welche sinnvollen Strategien es hier gibt.

Beim Narrowing-Verfahren sind wir an der Vollständigkeit interessiert, d.h. wir wollen alle Lösungen bzw. Repräsentanten aller Lösungen finden. Nach dem Satz 4.1 von Hullot (Kap. 4.2) wissen wir, dass wir die Vollständigkeit sicherstellen können, in dem wir *alle Regeln* an *allen passenden Stellen* ausprobieren.

Das Ausprobieren aller Regeln ist im Allg. nicht vermeidbar, wie das folgende Beispiel zeigt:

$$\begin{array}{lcl} f(a) & \to & b \\ f(b) & \to & b \end{array}$$

Betrachten wir nun die Gleichung: $f(x) \doteq b$:

- Durch Anwendung der 1. Regel erhalten wir die Lösung: $\sigma = \{x \mapsto a\}$
- Durch Anwendung der 2. Regel erhalten wir die Lösung: $\sigma = \{x \mapsto b\}$

Diese beiden Lösungen sind unabhängig, sodass jede Regel zu einer Lösung beiträgt.

Ist aber das Ausprobieren aller Positionen in einem Narrowing-Schritt vermeidbar?

- Reduktion: ja ($\leadsto$ lazy evaluation)
- Narrowing: hier ist die Antwort nicht so einfach wegen der Belegung freier Variablen

In diesem Kapitel wollen wir daher Narrowing-Strategien kennenlernen, die das einfache Ausprobieren an allen Positionen verbessern.

### 4.4.1 Strikte Narrowing-Strategien

Wir betrachten zunächst strikte Narrowing-Strategien. Das Problem hierbei ist allerdings, das die „innerste“ bzw. „äußerste“ Position eventuell nicht eindeutig ist.

Beispiel: Betrachten wir die Regel

$$rev(rev(l)) \to l$$

und den Term

$$rev(rev(xs))$$

Welche Position ist die „innerste“? Dies könnte einerseits die Wurzelposition sein, weil wir dort die Regel anwenden können:

$$rev(rev(xs)) \leadsto_{1,\ldots,\{\}} xs$$

Auf der anderen Seite können wir die Regel auch auf das innere $rev$ anwenden, wenn wir die Variable $xs$ passend belegen:

$$rev(rev(xs)) \leadsto_{1,\ldots,\{xs \mapsto rev(xs_1)\}} rev(xs_1)$$

Dieses Problem können wir vermeiden, wenn die Termersetzungssysteme *konstrukturbasiert* sind (vgl. Kap.3.2), d.h. wenn bei allen Regeln der Form

$$f(t_1, \ldots, t_n) \to r$$

gilt, dass jedes $t_i$ nur Konstruktoren und Variablen enthält. Die linken Seiten dieser Form werden auch **Muster** genannt (vgl. Definition 3.12).

Nicht zulässig wären dann also die Regeln:

$$\begin{aligned} rev(rev(l)) &\to l \\ (x + y) + z &\to x + (y + z) \end{aligned}$$

Für praktische Programme ist dies keine echte Einschränkung, da funktionale Programme immer konstruktorbasiert sind. Obwohl mit Narrowing auch nicht-konstruktorbasierte Termersetzungssysteme bearbeitet werden, gehören solche Systeme eher in den Bereich der Spezifikation und sind keine effektiv ausführbaren Programme.

Im Folgenden nehmen wir daher an:

**Das Termersetzungssystem ist konstruktorbasiert.**

Damit können wir nun strikte Narrowing-Strategien definieren:

**Definition 4.6** *Eine Narrowing-Ableitung*

$$t_0 \leadsto_{p_1,\sigma_1} t_1 \leadsto_{p_2,\sigma_2} \ldots \leadsto_{p_n,\sigma_n} t_n$$

*heißt* ***innermost*** *(≈ strikt), falls $t_{i-1}|_{p_i}$ ein Muster ist ($i = 1, \ldots, n$).*

Beispiel: Betrachten wir wieder die Additionsoperation:

$$\begin{aligned} 0 + n &\to n \\ s(m) + n &\to s(m + n) \end{aligned}$$

Dann gibt es z.B. folgende Narrowing-Ableitungen:

$$x + (y + z) \doteq 0 \leadsto_{1\cdot 2,\{y \mapsto 0\}} x + z \doteq 0 \leadsto_{1,\{x \mapsto 0\}} z \doteq 0 \quad \textbf{innermost}$$

$$x + (y + z) \doteq 0 \leadsto_{1,\{x \mapsto 0\}} y + z \doteq 0 \leadsto_{1,\{y \mapsto 0\}} z \doteq 0 \quad \textbf{nicht innermost}$$

Innermost Narrowing entspricht der strikten Auswertung in funktionalen Sprachen und ist daher ähnlich effizient implementierbar. Der Nachteil ist allerdings, dass Innermost Narrowing nur eingeschränkt vollständig ist, und zwar aus zwei Gründen:

1. Die berechneten Lösungen sind evtl. zu speziell:

   $f(x) \to a$
   $g(a) \to a$

   Gleichung: $f(g(z)) \doteq a$

   Eine mögliche Lösung ist: $\{\}$ (Identität), denn es gilt: $f(g(z)) \to_R a$

   Aber: es existiert nur eine Innermost Narrowing-Ableitung:

   $$f(\underline{g(z)}) \doteq a \rightsquigarrow_{1\cdot 1,\{z \mapsto a\}} \underline{f(a)} \doteq a \rightsquigarrow_{a,\{\}} a \doteq a$$

   Somit berechnet Innermost Narrowing die speziellere Lösung $\{z \mapsto a\}$.

2. Bei partiellen Funktionen schlägt Innermost Narrowing evtl. fehl:

   $$\begin{array}{rcl} f(a,z) & \to & a \\ g(b) & \to & b \end{array}$$

   Gleichung: $f(x, g(x)) \doteq a$
   Innermost Narrowing resultiert in einem Fehlschlag:

   $$f(x, \underline{g(x)}) \doteq s \rightsquigarrow_{\{x \mapsto b\}} f(b,b) \doteq a$$

   Dagegen kann das allgemeine Narrowing eine Lösung berechnen:

   $$\underline{f(x, g(x))} \doteq a \rightsquigarrow_{\{x \mapsto a\}} a \doteq a$$

Es gibt allerdings eine eingeschränkte Vollständigkeitsaussage von Innermost Narrowing, wie der folgende Satz zeigt:

**Satz 4.3 ([Fribourg 85])** *Sei $R$ ein Termersetzungssystem, sodass $\to_R$ konfluent und terminierend ist. Weiterhin gelte für alle Grundterme $t$: wenn $t$ in Normalform ist, dann enthält $t$ nur Konstruktoren (d.h. alle Funktionen sind total definiert).*
*Dann ist Innermost Narrowing vollständig für alle „Grundlösungen“, d.h. es gilt: Falls $\sigma'$ eine Lösung von $s \doteq t$ ist mit der Eigenschaft, dass $\sigma(x)$ ein Grundterm ist für alle $x \in Var(s \doteq t)$, dann existieren eine Innermost Narrowing-Ableitung $s \doteq t \rightsquigarrow_\sigma^* s' \doteq t'$, ein mgu $\varphi$ für $s'$ und $t'$ und eine Substitution $\tau$ mit $\sigma'(x) \doteq \tau(\varphi(\sigma(x)))$, sodass $\forall x \in Var(s \doteq t)$ gültig ist.*

Viele Funktionen sind total definiert, sodass Innermost Narrowing dafür vollständig ist. Falls allerdings auch partielle Funktionen vorhanden sind, dann kann man Innermost Narrowing so erweitern, dass man bei partiellen Funktionen nicht fehlschlägt. Diese Erweiterung heißt **Innermost Basic Narrowing** und basiert auf folgender Idee:

- Erlaube das „Überspringen“ innerer Funktionsaufrufe

- Falls ein Aufruf übersprungen wurde, ignoriere diesen auch in allen nachfolgenden Schritten (diese Technik wurde auch unabhängig vom Innermost Narrowing als **Basic Narrowing** in [Hullot 80] vorgeschlagen).

Für eine genaue formale Beschreibung ist es notwendig, zwischen auszuwertenden und übersprungenen Funktionsaufrufen zu unterscheiden. Hierzu stellen wir Gleichungen als Paar

$$\langle E;\ \sigma \rangle$$

dar, wobei das „Skelett" $E$ die auszuwertenden und die „Umgebung" $\sigma$ die übersprungenen Funktionsaufrufe enthält. Das Paar $\langle E;\ \sigma \rangle$ steht dabei für den Ausdruck $\sigma(E)$.

Intuitiv wird diese Paardarstellung wie folgt verwendet:

- Eine initiale Gleichung $E$, die gelöst werden soll, wird als Paar $\langle E; [] \rangle$ dargestellt.
- Auswertung: hier werden nur Aufrufe in $E$ ausgewertet
- „Überspringen": verschiebe einen Aufruf von $E$ nach $\sigma$

Beispiel: Betrachten wir das oben angegebene Termersetzungssystem $R$ mit einer partiellen Funktion:

$$\begin{array}{rcl} f(a,z) & \to & a \\ g(b) & \to & b \end{array}$$

Gleichung: $f(x, g(x)) \doteq a$

Darstellung als Paar:

$$\begin{array}{lll} & \langle f(x,g(x)) \doteq a; \{\} \rangle & \\ \leadsto & \langle f(x,y) \doteq a; \{y \mapsto g(x)\} \rangle & \text{„überspringen"} \\ \leadsto_{\{x \mapsto a\}} & \langle a \doteq a; \{y \mapsto g(a), x \mapsto a\} \rangle & \text{„Innermost Narrowing im Skelett"} \end{array}$$

Damit haben wir die Lösung: $\{x \mapsto a\}$ berechnet, d.h. mittels des "Überspringens" ist Innermost Basic Narrowing vollständig.

Wir wollen nun diese Strategie formal beschreiben: Innermost basic Narrowing besteht aus zwei alternativen Ableitungsschritten:

**Narrowing:**

- $p \in \mathcal{P}os(E)$ mit $E|_p$ ist ein Muster
- $l \to r$ ist eine neue Variante einer Regel
- $\sigma'$ ist ein mgu für $\sigma(E|_p)$ und $l$

Dann ist

$$\langle E; \sigma \rangle \ \leadsto\ \langle E[r]_p; \sigma' \circ \sigma \rangle$$

ein „Innermost Basic Narrowing-Schritt"

**Innermost reflection:**

- $p \in Pos(E)$ mit $E|_p$ ist ein Muster
- $x$ ist eine neue Variable mit $\sigma' = \{x \mapsto \sigma(E|_p)\}$

Dann ist

$$\langle E; \sigma \rangle \ \leadsto \ \langle E[x]_p; \sigma' \circ \sigma \rangle$$

ein „Überspringen"-Schritt.

Anmerkungen:

- In beiden Fällen kann $p$ auch die linkeste Position sein.
- Innermost basic Narrowing ist vollständig für konfluente und terminierende Termersetzungssysteme.

Reine innermost Narrowing-Verfahren haben trotz dieser Verbesserung einen Nachteil (im Unterschied zu innermost Reduktionstrategien!): Sie laufen häufig in Endlosschleifen, wenn rekursive Datenstrukturen verwendet werden. Dies passiert auch, wenn das Termersetzungssystem terminierend ist!

Beispiel:

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

Dieses Termersetzungssystem ist konfluent, terminierend und total definiert, d.h. es hat alle für Innermost Narrowing wünschenswerten Eigenschaften. Allerdings gibt es schon bei einfachen Gleichungen endlose Innermost Narrowing-Ableitungen:

$$x + y \doteq 0 \ \leadsto_{\{x \mapsto s(x_1)\}} \ s(x_1 + y) \doteq 0 \ \leadsto_{\{x_1 \mapsto s(x_2)\}} \ s(s(x_2 + y)) \doteq 0 \ \leadsto \ \ldots$$

Der letzte und alle weiteren Narrowing-Schritte sind überflüssig, da die linke und rechte Seite der Gleichung nie gleich werden kann, da $s$ und 0 verschiedene Konstruktoren sind.

Eine Verbesserung kann erreicht werden, wenn wir prüfen (vor einem Narrowing-Schritt!), ob bei einer Gleichung verschiedene Konstruktoren außen stehen. Hierzu definieren wir:

$$\mathcal{H}ead(t) = f \ \ :\Leftrightarrow \ \ t = f(t_1, \ldots, t_n)$$

**Rückweisung:** („rejection"):
Falls $s \doteq t$ die zu lösende Gleichung ist und $p \in Pos(s) \cap Pos(t)$ mit $Head(s|_p) \neq Head(t|_p)$ und $Head(s|_{p'}), Head(t|_{p'}) \in C$ für alle Positionen $p' \leq p$ (hierbei ist $C$ wie immer die Menge der Konstruktoren), dann hat $s \doteq t$ keine Lösung, d.h. wir können die Narrowing-Ableitung sofort mit einem Fehlschlag abbrechen.

Allerdings gibt es auch mit der Rückweisungsregel noch viele überflüssige Ableitungen:

$$\begin{array}{lll} \underline{(x+y)} + z \doteq 0 & \leadsto_{\{x \mapsto s(s_1)\}} & s(\underline{x_1 + y}) + z \doteq 0 \\ & \leadsto_{\{x_1 \mapsto s(x_2)\}} & s(s(\underline{x_2 + y})) + z \doteq 0 \\ & \leadsto \ldots & \end{array}$$

Bei dieser Ableitung erfolgt keine Rückweisung, da + eine definierte Funktion und kein Konstruktor ist!

Eine weitere Verbesserung können wir dadurch erreichen, dass wir *vor* jedem Narrowing-Schritt den Term zur Normalform reduzieren. Dieses erweiterte Verfahren wird auch als **normalisierendes Innermost Narrowing** bezeichnet [Fay 79, Fribourg 85].

Betrachten wir erneut das obige Beispiel und führen es mit normalisierendem Innermost Narrowing aus:

$$\begin{array}{lll} \underbrace{(x+y) + z \doteq 0}_{in\ Normalform} & \leadsto_{\{x \mapsto s(x_1)\}} & s(x_1 + y) + z \doteq 0 \\ & \rightarrow & s((x_1 + y) + z) \doteq 0 \\ & & \text{Rückweisung: Fehlschlag!} \end{array}$$

Normalisierendes Innermost Narrowing hat einige interessante Eigenschaften:

- Normalisierung ist ein deterministischer Prozess (für konfluente Termersetzungssysteme), da wir der Normalisierung Priorität geben (diese findet ja *vor* dem Narrowing statt), führt dies zu einer kürzeren Laufzeit und geringerem Speicherplatzverbrauch.

  Beispiel:

  $$\begin{array}{lll} 0 * n & \rightarrow & 0 \\ n * 0 & \rightarrow & 0 \end{array}$$

  Gleichung: $x * 0 \doteq 0$
  Innermost Narrowing erlaubt zwei verschieden Ableitungen:

  $$\begin{array}{lll} x * 0 \doteq 0 & \leadsto_{\{x \mapsto 0\}} & 0 \doteq 0 \\ x * 0 \doteq 0 & \leadsto_{\{\}} & 0 \doteq 0 \end{array}$$

  Innermost Narrowing mit Normalisierung ist dagegen deterministisch, weil nur eine Ableitung möglich ist:

  $$x * 0 \doteq 0 \quad \rightarrow \quad 0 \doteq 0$$

- Vermeidung unnötiger Ableitungen
- Es ist effizient implementierbar, indem man ähnliche Implementierungstechniken wie für funktionale oder logische Programmiersprachen verwendet (vgl. [Hanus 90, Hanus 91, Hanus 92])

- Durch diese Technik können logisch-funktionale Programme effizienter abgearbeitet werden als rein logische Programme, wie nachfolgend gezeigt wird.

Nachteil rein logischer Programme:

- Sie haben eine flache Struktur.
- Sie haben keine expliziten funktionalen Abhängigkeiten.

Beispiel: Betrachten wir die Implementierung des obigen Beispiels in Prolog:

```
add(0, N, N).  % hier ist N ein Extraargument für das Funktionsergebnis
add(s(M), N, s(Z)) :- add(M, N, Z).
```

Die Anfrage ist:

```
add(X, Y, R), add (R, Z, 0)
```

Hierbei ist `R` eine Hilfsvariable, die notwendig ist zur Abflachung des Terms `((x+y)+z)`. Zu dieser Anfrage gibt es die folgende endlose Ableitung:

```
add(X, Y, R), add (R, Z, 0)
```
$\leadsto_{\{X \mapsto s(X1), R \mapsto s(R1)\}}$ `add(X1, Y, R1), add(s(R1), Z, 0)`
$\leadsto_{\{X1 \mapsto s(X2), R1 \mapsto s(R1)\}}$ `add(X2, Y, R2), add(s(s(R2)), Z, 0)`
$\leadsto \cdots$

Die Voraussetzung für alle Innermost-Strategien und auch für die Normalisierung ist die Terminierung des Termersetzungssystems. Diese Voraussetzung ist allerdings sehr streng, denn

- sie ist schwer zu prüfen (unentscheidbar!)
- sie verbietet bestimmte funktionale Programmiertechniken (unendliche Datenstrukturen, Kap. 2.6)

Bei den Reduktionsstrategien hatten wir gesehen, dass outermost oder lazy Strategien bei nichtterminierenden Termersetzungssystemen vorteilhaft sind. Aus diesem Grund wollen wir nachfolgend diskutieren, ob dies auch bei Narrowing-Strategien so ist.