4.5 Residuation

Die Auswertungsstrategie „Residuation“ basiert auf folgendem Prinzip:

> Funktionen werden nur deterministisch ausgerechnet, d.h. eine Funktionsanwendung wird nicht ausgerechnet (sondern „verzögert“), falls einige Argumente geraten werden müssen.

Falls man dennoch in einem Programm nach passenden Werten suchen will und daher Nichtdeterminismus erwünscht ist, erlaubt man die *nichtdeterministische Auswertung in Prädikaten* (d.h. boolesche Funktionen), aber nicht in anderen Funktionen.

Ein Problem ergibt sich allerdings noch. Wenn bestimmte Berechnungen verzögert werden, stellt sich die Frage, wo dann stattdessen weitergerechnet werden soll. Man benötigt also ein Konzept von nebenläufiger Auswertung in der Sprache. Zu diesem Zweck können wir beispielsweise einen speziellen Operator einführen, der seine Argumente nicht zwingend nacheinander, sondern nebenläufig auswertet, d.h. falls die Auswertung eines Arguments suspendiert, wird das andere ausgewertet. In Anlehnung an die Logikprogrammierung bietet sich hier ein Konjunktionsoperator “`&`” an, der verlangt, dass beide Argumente zu `True` ausgewertet werden, d.h. der wie folgt definiert ist:

```
(&) :: Bool  → Bool  → Bool
True & True = True
```

Da beide Argumente ausgewertet müssen, können diese auch nebenläufig oder parallel ausgewertet werden. Falls also ein Argument den Wert `True` hat, muss nur noch das andere zu `True` ausgewertet werden. Aus diesem Grund können wir bei der Berechnung auch folgenden Vereinfachungsregeln verwenden:

```
True & x  → x
x & True  → x
```

Beispiel: Betrachten wir die folgenden Regeln:

$$\begin{array}{rr} 0 + n \to n & \texttt{nat}(0) \to \texttt{True} \\ s(m) + n \to s(m + n) & \texttt{nat}(s(n)) \to \texttt{nat}(n) \end{array}$$

Hier ist also “+” eine Funktion und `nat` ein Prädikat. Somit werden bei Residuation Auswertungsschritte für “+” so lange verzögert, bis das erste Argument keine Variable ist.

Anfrage:	$\underbrace{x + 0}$ == $s(0)$	& $\underbrace{\texttt{nat}(x)}$
	nicht auswerten!	auswerten, Nichtdeterminismus zulässig
	$\leadsto_{\{x \mapsto s(x_1)\}}$	$\underbrace{s(x_1) + 0}$ == $s(0)$ & $\texttt{nat}(x_1)$
		jetzt auswerten
	$\to$	$s(x_1 + 0)$ == $s(0)$ & $\texttt{nat}(x_1)$
	$\to$	$\underbrace{x_1 + 0}$ == 0 & $\texttt{nat}(x_1)$
		nicht auswerten
	$\leadsto_{\{x_1 \mapsto 0\}}$	$\underbrace{0 + 0}$ == 0 & `True`
		auswerten
	$\to$	0 == 0
	$\to$	`True`

Die Vorteile von Residuation sind:

- Funktionen werden nur „funktional“ (deterministisch) verwendet.
- Residuation unterstützt nebenläufige Programmierung
  (Funktionen $\approx$ Konsumenten, Prädikate $\approx$ Erzeuger):
  Synchronisation über logische Variablen (einfaches Konzept, gestützt auf formale Logik, „deklarative Nebenläufigkeit“)
- Es ist ein einfacher Anschluss externer Funktionen möglich:
  Bsp.: Anschluss einer C-Bibliothek: Behandle externe Funktionen wie normale Funktionen, verzögere den Aufruf jedoch bis alle Argumente gebunden sind

  Bsp.: Arithmetik: Statt $0/s$-Terme und explizite Funktionsdefinitionen wie in den bisherigen Narrowing-Beispielen:

  – Fasse Zahlkonstanten als Konstruktoren (unendlich viele) auf.
  – Interpretiere $x + y$ als externe Funktion, die erst aufgerufen wird, wenn x und y an Konstanten gebunden sind.

  Dies ist die Grundlage der arithmetischen Operationen, die in Curry eingebaut sind. Somit sind in Curry folgende Berechnungen möglich:

  ```
  x==3 & y==5 & z==x*y   ~>   {x=3, y=5, z=15} True

  x==y+1 & y==2          ~>   {y=2, x=3} True
  ```

  Man beachte, dass bei der zweiten Berechnung die Auswertung der ersten Gleichheitsbedingung zunächst verzögert wird.

Residuation hat allerdings auch einige Nachteile:

- Residuation ist ein unvollständiges Lösungsverfahren. Betrachten wir z.B.

```
x+0 == s(0)
```

Die Auswertung wird mit Residuation verzögert und liefert somit kein Ergebnis, wogegen mit Narrowing die Lösung $\{\texttt{x} \mapsto \texttt{s(0)}\}$ ausgerechnet wird.

Mit Residuation wäre dieses Constraint oder auch das Constraint "`2 == x+1`" nur dann vollständig ausrechenbar, falls alle Funktionsargumente (wie `x`) im Berechnungsverlauf gebunden werden.

- Es kann bei Residuation durch die verzögerte Auswertung von Bedingungen sogar zu unendlichen Ableitungen kommen.

  Betrachten wir als Beispiel die folgende alternative Definition der Listenumkehrung `rev` (als Prädikat):

```
rev [] []     = True
rev zs (x:xs) | rx++[x]==zs & rev rx xs
              = True            where rx free
```

  Hier kommt es bei Anwendung der zweiten Klausel zu einer unendlichen Ableitung, weil die Auswertung von "`++`" verzögert wird:

```
rev [0] zs
⇝{zs↦x:xs}     rx++[x]==[0] & rev rx xs
⇝{xs↦x1:xs1}   rx++[x]==[0] & rx1++[x1]==rx & rev rx1 xs1
⇝{xs1↦x2:xs2}  ...
```

  Die Ursache für dieses ungünstige Verhalten liegt in der Erzeugung neuer Bedingungen im rekursiven Fall, die alle verzögert werden ($\approx$ "passive constraints").

  Wird jedoch "`++`" mit Lazy/Needed Narrowing ausgewertet, dann ist der Suchraum endlich, da bei Bindung von `rx` an zu lange Listen die Bedingungen fehlschlagen.

  Somit ist die Verzögerung des „Ratens" von Funktionsargumenten nicht immer besser als nichtdeterministisches Raten.

Als Zusammenfassung dieses Kapitels stellen wir noch einmal die Vor- und Nachteile von Narrowing und Residuation gegebenüber:

Narrowing:		**Residuation:**
+	vollständig	− unvollständig
−	Nichtdeterminismus bei Funktionen	+ Determinismus bei Funktionen
+	Optimalität bei induktiv-sequentiellen Funktionen	− ??
−	Anschluss externer Funktionen nicht möglich	+ Anschluss externer Funktionen trivial
		+ nebenläufige Programmierung

Aus diesem Grund ist es sinnvoll, beide Auswertungstechniken miteinander zu kombinieren, was die Grundlage des Berechnungsmodells der Sprache Curry ist, das wir im nächsten Kapitel vorstellen.

4.6 Ein einheitliches Berechnungsmodell für deklarative Sprachen

Wie wir gesehen haben, haben sowohl Narrowing als auch Residuation ihre Vorteile. Daher stellt sich die Frage, wie man beides sinnvoll kombinieren kann. Insbesondere muss man definieren, wann eine Berechnung verzögert wird und wo man in diesem Fall weiterrechnen soll.

4.6.1 Kombination von Narrowing und Residuation

Im folgenden werden wir sehen, dass man die auf den ersten Blick recht unterschiedlichen Berechnungsmodelle von Narrowing und Residuation recht einfach kombinieren kann. Dies kann man erreichen, indem definierende Bäume leicht erweitert werden. Hierzu fassen wir noch einmal die Kernpunkte von *Lazy Evaluation* zusammen:

- Vorteilhaft sowohl bei funktionaler und auch bei logischer Programmierung:
  - Rechnen mit unendlichen Datenstrukturen
  - Vermeidung unnötiger Berechnungen
  - normalisierende (vollständige) Strategie
- Präzise Beschreibung mit
  - case-Ausdrücken (funktionale Programmierung)
  - definierenden Bäumen (case-Ausdrücke + Oder-Knoten)

Definierende Bäume sind geeignet zur Beschreibung guter Narrowing-Strategien, aber können diese auch für Residuation genutzt werden?

Machen wir uns hierzu den Hauptunterschied zwischen Narrowing und Residuation klar:

> Falls ein verlangtes Funktionsargument eine freie Variable ist:
>
> - Narrowing: binde Variable
> - Residuation: verzögere Aufruf

Da sich verlangte Argumente immer auf branch-Knoten beziehen, kann man diesen Unterschied sehr einfach in branch-Knoten explizit machen. Hierzu erweitern wir branch-Knoten um ein Flag/Bit, das diesen Unterschied markiert. Von nun an hat ein branch-Knoten die Form

$branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$

wobei die Definition analog zu $branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ ist und zusätzlich

$r \in \{\texttt{rigid}, \texttt{flex}\}$

gilt. Hierbei steht `rigid` für das Verzögern und `flex` für das sofortige Binden von Variablen.

Formal können wir nun die Strategie λ auf diesen neuen Verzweigungsknoten definieren, indem für

- $r = \texttt{flex}$ die Definition wie bisher ist (d.h. identisch zu Needed Narrowing), und für
- $r = \texttt{rigid}$ das Ergebnis der Auswertung die spezielle Konstante `suspend`, die eine verzögerte Auswertung anzeigt.

Formal erweitern wir λ (vgl. Definition 4.10) um den folgenden Fall:

$$\lambda(t, \mathcal{T}) = \texttt{suspend} \quad \text{falls } \mathcal{T} = \text{branch}(\tau, p, rigid, \mathcal{T}_1, \ldots \mathcal{T}_n) \text{ und } t|_p \text{ ist Variable}$$

Die nächste Frage, die zu beantworten ist: Wo muss man weiterrechnen, falls das Ergebnis `suspend` ist? Betrachten wir noch einmal das obige Beispiel:

```
x+0 == s(0)  &  nat(x)
```

Da die Auswertung des ersten Constraint suspendiert wird, muss man hier mit der Auswertung von `nat(x)` weitermachen.

Dies können wir durch folgende Erweiterung von λ auf `&`-Termen definieren:

$$\lambda(e_1 \texttt{ \& } e2) \quad = \quad \begin{cases} 1 \cdot \lambda(e_1) & \text{falls } \lambda(e_1) \neq \texttt{suspend} \\ 2 \cdot \lambda(e_2) & \text{falls } \lambda(e_1) = \texttt{suspend} \text{ und } \lambda(e_2) \neq \texttt{suspend} \\ \texttt{suspend} & \text{sonst} \end{cases}$$

wobei die Konkatenation einer Position i zu einer Narrowing-Tripelmenge S wie folgt definiert ist:

$$i \cdot S = \{(i \cdot p, r, \sigma) \mid (p, r, \sigma) \in S\}$$

Durch diese Definition wird eine Priorität für die Berechnung des linken Constraint festgelegt, aber prinzipiell kann man auch eine faire (indeterministische) Auswahl beider Constraints erlauben.

4.6.2 Auswertungsstrategie von Curry

Nun haben wir alle Elemente gesehen, um die konkrete Auswertungsstrategie der Sprache Curry zu beschreiben:

- **Auswertungsstrategie / Pattern Matching:**
  Für jede Funktion wird ein erweiterter definierender Baum erzeugt. Diese Erzeugung passiert ähnlich wie das Pattern Matching in Haskell (vgl. Kapitel 2.3), allerdings wird für induktiv-sequentielle Funktionen immer ein definierender Baum erzeugt, sodass die Auswertung mit Pattern Matching in diesem Fall optimal ist.

  Betrachten wir z.B. das Programm

  ```
  g 0 []     = 0
  g _ (x:xs) = x

  h x = h x
  ```

  und den zu berechnenden Ausdruck

  ```
  (g (h 0) [1])
  ```

  Diese Berechnung terminiert in Haskell nicht (wegen des in Kapitel 2.3 dargestellten strikten Links-Rechts-Pattern-Matching). In Curry wird dagegen das Ergebnis `1` geliefert, denn die Funktion `g` ist durch Fallunterscheidung über das zweite Argument induktiv-sequentiell.

- **Unterscheidung zwischen flexiblen und rigiden Berechnungen:**
  Die branch-Knoten haben bei den erzeugten definierenden Bäumen immer das Flag `flex`, d.h. alle benutzerdefinierten Operationen sind flexibel. Externe Funktionen, die nicht in Curry selbst definiert sind (z.B. arithmetische Operationen wie "`+`", "`*`",...), sind dagegen rigid.

  Will man als Benutzer selbst rigide Funktionen definieren (was aber nur selten notwendig ist), dann gibt es dazu zwei Möglichkeiten:

  1. Explizit verwendete `case`-Ausdrücke (und auch die damit definierte Operation `if-then-else`) sind rigid. Zum Beispiel wird mit

     ```
     rnot x = case x of True  → False
                        False → True
     ```

     eine Operation `rnot` definiert, für die der Ausdruck "`rnot x`" (wobei `x` eine freie Variablen ist) suspendiert.

  2. Expliziter Verzögerungsoperator

     ```
     ensureNotFree :: a → a
     ```

     Der Ausdruck "`ensureNotFree` e" suspendiert, so lange e zu einer ungebundenen Variablen ausgewertet wird, ansonsten wird der Wert von e (d.h. in Kopfnormalform) zurückgeliefert. Mittels dieses primitiven Operators werden externe Funktionen suspendiert.

- **Bedingte Regeln:** l `|` c `=` r
  Die generelle Strategie zur Anwendung einer bedingten Regel ist:
  1. Pattern Matching der linken Seite l.
  2. Beweise die Bedingung c, d.h. reduziere diese zu `True`.
  3. Falls beide Schritte erfolgreich sind, ersetze den zu l passenden Teilausdruck durch r.

  Wie schon früher erläutert, kann diese Bedeutung durch eine Transformation in eine unbedingte Regel definiert werden, d.h. die bedingte Regel

  l `|` c `=` r

  wird transfomiert in

  l `=` c `&>` r

  wobei die Operation "`&>`" ("bedingter Ausdruck") durch

  ```
  (&>) :: Bool → a → a
  True &> x = x
  ```

  definiert ist.

  Hierbei ist c eine boolesche Bedingung. Ähnlich wie in Haskell kann man auch eine Folge von Bedingungen angeben, was allerdings einem sequentiellen if-then-else entspricht (und nicht, wie bei mehreren Regeln, einem nichtdeterministischen Ausprobieren aller Bedingungen!). Beispielsweise ist die Definition

  ```
  f x | x>0 = 1
      | x<0 = 0
  ```

  äquivalent zu

  ```
  f x = if x>0 then 1 else
         if x<0 then 0 else failed
  ```

- **Funktionen höherer Ordnung:**
  Dies ist analog zur rein funktionalen Programmierung (vgl. Kapitel 2.4), wobei der (nicht sichtbare) Applikationsoperator rigide ist, d.h. ein Ausdruck der Form "`x 2`" suspendiert, wenn `x` eine freie Variable ist. Als Alternative könnte man auch passende Funktionen raten (z.B. [Antoy/Tolmach 99]), was aber leicht zu großen Suchräumen führt.