Erweiterung von Needed Narrowing

Needed Narrowing ist nur für induktiv-sequentielle Termersetzungssysteme definiert. Dies bedeutet, dass *überlappende Regeln* bei Needed Narrowing nicht erlaubt sind. Als Beispiel betrachten wir hierzu noch einmal das „parallele Oder“:

```
R: True  ∨ x     → True
   x     ∨ True  → True
   False ∨ False → False
```

Das Problem bei dieser Definition ist, dass es kein eindeutiges Induktionsargument gibt, d.h. kein Argument ist “needed”, sodass bei der Auswertung von $t_1 \vee t_2$ unklar ist, ob man zuerst t_1 oder zuerst t_2 auswerten soll.

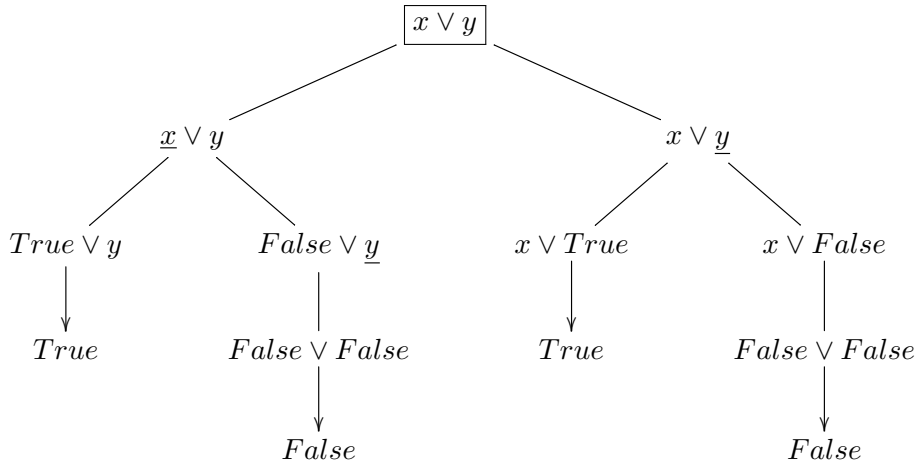
Weil allerdings solche überlappenden Definitionen in der Praxis durchaus vorkommen können, erweitern wir definierende Bäume, indem wir zusätzlich Oder-Knoten, d.h. Knoten für alternative Auswertungen, einführen:

Definition 4.11 *Ein **erweiterter definierender Baum** ist ein definierender Baum, der folgende Knoten mit einem Muster π enthalten kann:*

- *Regelknoten: wie bisher*
- *Verzweigungsknoten: wie bisher*
- ***Oder-Knoten** der Form $or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, wobei jedes $\mathcal{T}_i$ ein erweiterter definierender Baum mit dem Muster π ist.*

$\mathcal{T}$ heißt erweiterter definierender Baum für eine n-stellige Funktion f, falls $\mathcal{T}$ endlich ist und das Muster $f(x_1, \ldots, x_n)$ hat ($x_1, \ldots, x_n$ sind verschiedene Variablen), wobei jede Regel $f(t_1, \ldots, t_n) \to r$ aus R in $\mathcal{T}$ mindestens einmal vorkommt.

Beispiel: Ein erweiterter definierender Baum für das parallele Oder kann wie folgt graphisch dargestellt werden, wobei der oberste eingerahmte Wurzelknoten einen Oder-Knoten darstellt:



Wie man sieht, kommt die letzte Regel der Operation "$\vee$" in dem erweiterten definierenden Baum zweimal vor.

Anzumerken ist, dass für alle Operationen eines konstruktorbasierten Termersetzungssystems immer erweiterte definierende Bäume konstruiert werden können.

Die Erweiterung von Needed Narrowing bzgl. erweiterten definierenden Bäumen ist recht einfach, wodurch wir eine lazy Narrowing-Strategie für KB-SO Termersetzungssysteme erhalten:

Definition 4.12 (Weakly Needed Narrowing) *Die Erweiterung von Needed Narrowing bzgl. erweiterten definierenden Bäumen wird* ***Weakly Needed Narrowing*** *genannt und ist wie Needed Narrowing definiert, jedoch mit folgender Erweiterung von λ für Oder-Knoten:*

$$\lambda(t, or(\mathcal{T}_1, \ldots, \mathcal{T}_k)) \supseteq \lambda(t, \mathcal{T}_i) \qquad (i = 1, \ldots, k)$$

Somit probiert die Strategie λ bei Oder-Knoten alle Alternativen aus.

Beispiel: Betrachten wir die Regeln für das parallele Oder "$\vee$" und außerdem die Regel

$$f(a) \;\rightarrow\; \texttt{True} \qquad (R_4)$$

Der auszurechnende Term sei $t = f(x) \vee f(y)$.
Dann gilt:

$$\lambda(t) = \{(1, R_4, \{x \mapsto a\}),\ (2, R_4, \{y \mapsto a\})\}$$

Daher sind

$$t \;\leadsto_{\{x \mapsto a\}}\; \texttt{True} \vee f(y)$$

und

$$t \;\leadsto_{\{y \mapsto a\}}\; f(x) \vee \texttt{True}$$

alle möglichen Weakly Needed Narrowing-Schritte für den Term t.

Weakly Needed Narrowing ist zwar nicht mehr optimal wie Needed Narrowing, aber es gilt:

Satz 4.6 ([Antoy/Echahed/Hanus 97]) *Weakly Needed Narrowing ist korrekt und vollständig für KB-SO Termersetzungssysteme bzgl. strikter Gleichheit.*

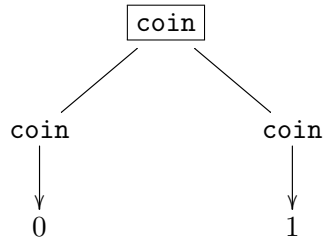
Darüber hinaus ist Weakly Needed Narrowing eine konservative Erweiterung, d.h. bei induktiv-sequentiellen Termersetzungssystemen verhält sich Weakly Needed Narrowing wie Needed Narrowing und ist damit auch optimal auf dieser Teilklasse.

4.4.3 Nichtdeterministische Operationen

Wie wir gesehen haben, ist die Voraussetzung für die Anwendung der (Weakly) Needed Narrowing-Strategie die Existenz erweiterter definierender Bäume. Dies bedeutet, dass die Eigenschaft „konstruktorbasiert“ der Termersetzungssysteme wichtig ist. Wie sieht es allerdings mit der Konfluenzeigenschaft aus? Betrachten wir dazu die schon früher vorgestellte Operation `coin`, die zu 0 oder 1 ausgewertet werden kann:

```
coin = 0
coin = 1
```

Da diese trivialerweise konstruktorbasiert ist, existiert hierzu auch ein erweiterter definierender Baum, wobei der Wurzelknoten ein Oder-Knoten ist:



Weil die Auswertung von `coin` die beiden unterschiedlichen Werte 0 oder 1 liefert, spricht man auch von einer **nichtdeterministischen Operation**.

Da ein erweiterter definierender Baum für `coin` existiert, kann man offensichtlich `coin` auch mit Weakly Needed Narrowing auswerten. Was spricht also dagegen, solche nichtdeterministischen Operationen (also solche, die nicht konfluent sind) zuzulassen? In [González-Moreno *et al.* 99] wurde gezeigt, dass nichts dagegen spricht, denn man kann auch nichtdeterministischen Operationen eine präzise mathematische Bedeutung geben. Wenn z.B. eine deterministische Funktion f eine Abbildung der Art

$$f : values \to values$$

ist, denn kann eine nichtdeterministische Operation g interpretiert werden als Abbildung der Art

$$g : values \to 2^{values}$$

d.h. g bildet einen Eingabewert auf eine Menge von Ausgabewerten ab. Die genaueren Details wollen wir hier nicht diskutieren. Stattdessen wollen wir zeigen, wie solche Operationen sinnvoll zur Programmierung eingesetzt werden können (was wir schon früher einmal kurz an einem Beispiel gesehen haben).

Grundsätzlich können nichtdeterministische Operationen verwendet werden, um Berechnungen mit unterschiedlichen Ergebnissen elegant zu formulieren. Zu beachten ist auch, dass die Verwendung nichtdeterministischer Operationen keinen Mehraufwand verursacht, da eine logisch-funktionale Sprache auf Grund der logischen Anteile schon Nicht-

determinismus anbietet. Z.B. könnte die Operation `coin` auch ohne überlappende Regeln unter Verwendung einer freien Variablen definiert werden:[4]

```
coin = zeroOne x
  where
    x free

    zeroOne 0 = 0
    zeroOne 1 = 1
```

[4]Tatsächlich wurde in [Antoy/Hanus 06] gezeigt, dass überlappende Regeln und freie Variablen gleich mächtige Konzepte sind, d.h. freie Variablen können mit überlappenden Regelsystemen eliminiert werden und umgekehrt.

Beispiel: Berechnung von Permutationen

Wir haben schon in einem früheren Beispiel gesehen, wie man Permutationen einer Liste berechnen kann. Nun wollen wir eine andere Variante kennenlernen, die auf einer nichtdeterministischen Operation `insert` zum Einfügen eines Elements an einer beliebigen Stelle einer Liste beruht:

```
insert :: a → [a] → [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

Mit dieser Operation ist die Definition einer Permutation weitgehend trivial:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Somit liefert `perm` *eine* beliebige Permutation der Eingabeliste. In einer Programmierumgebung, die alle Werte eines Ausdruck ausgibt, würde also z.B. `perm [1,2,3]` zu den Werten

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

ausgerechnet werden.

Eine mögliche Anwendung von Permutationen ist das Sortieren einer Liste durch Aufzählung aller Permutationen. Hierzu definieren wir eine Funktion, die die Identität auf sortierten Listen ist (und auch nur für diese definiert ist):

```
sorted []              = []
sorted [x]             = [x]
sorted (x:y:ys) | x<=y = x : sorted (y:ys)
```

Damit kann die Sortierung einer Liste einfach definiert werden als Permutation, die sortiert ist:

```
sort xs = sorted (perm xs)
```

Man sollte beachten, dass diese Definition zwar kein effizienter Sortieralgorithmus ist. Es ist aber eine (ausführbare) Spezifikation des Sortierens, die man z.B. verwenden kann, um effizientere Sortieralgorithmen zu testen.

Diese Definition hat in einer nicht-strikten Sprache wie Curry einen interessanten operationalen Effekt. Wegen der lazy Auswertung wird das Argument `(perm xs)` nicht voll-

ständig ausgewertet, sondern nur bedarfsgesteuert. Hierdurch werden nicht alle Permutationen berechnet. Betrachten wir z.B. den Ausdruck

```
sort [6,5,4,3,2,1]
```

Dieser wird zunächst reduziert zu

```
sorted (perm [6,5,4,3,2,1])
```

Wegen der "needed" Strategie wird die Permutation nicht vollständig ausgerechnet, sondern nur soweit, wie es durch `sorted` verlangt wird, d.h. nur bis zu den beiden ersten Listenelementen (wobei zunächst jeweils die erste Regel für `insert` angewendet wird):

```
sorted (6 : 5 : perm [4,3,2,1])
```

Da allerdings `6<=5` nicht erfüllt ist, führt diese Berechnungsalternative zu einem Fehlschlag, sodass die Werte von "`perm [4,3,2,1]`" überhaupt nicht berechnet werden. Durch die bedarfsgesteuerte Auswertung werden hier also schon 4! Berechnungen eingespart.

Somit können wir festhalten:

> Die Kombination nichtdeterministischer Operationen und bedarfsgesteuerter Auswertung führt zu einer bedarfsgesteuerten Suche.

Diesen Effekt haben wir bei rein logischen Programmen nicht, da erst durch die Benutzung von Funktionen die bedarfsgesteuerte Auswertung sinnvoll ermöglicht wird.

Es gibt allerdings auch ein semantisches Problem bei der Anwendung von nichtdeterministischen Operationen mit nicht-strikten Strategien, denn es gibt zwei mögliche Optionen, wie nichtdeterministische Argumente einer Funktion interpretiert werden sollen:

- Die nichtdeterministische Auswahl findet zum Zeitpunkt des Funktionsaufrufs statt (dies wird auch **call-time choice** genannt).
- Die nichtdeterministische Auswahl findet zum Zeitpunkt der Auswertung der Funktionsargumente statt (dies wird auch **run-time choice** genannt).

Ein kleines Beispiel soll diesen Unterschied verdeutlichen. Wir betrachten die Operation

```
double x = x + x
```

und den Ausdruck "`double (1?2)`".

Call-time choice: Hier wird vor dem Aufruf von `double` der Wert des Arguments `(1?2)` (d.h. `1` oder `2`) festgelegt und dieser Wert dann innerhalb des Aufrufs verwendet. Somit sind dann `2` oder `4` die möglichen Ergebniswerte.

Run-time choice: Hier werden die möglichen Werte des Arguments `(1?2)` erst dann gewählt, wenn dieses Argument berechnet wird. Da das Argument zweimal im Rumpf vorkommt, können dann auch zweimal unabhängig die Werte gewählt werden. Somit gibt es bei run-time choice also die folgenden mögliche Ableitungen:

```
double (1?2) → (1?2) + (1?2) → 1 + (1?2) → 1 + 1 → 2
double (1?2) → (1?2) + (1?2) → 1 + (1?2) → 1 + 2 → 3
double (1?2) → (1?2) + (1?2) → 2 + (1?2) → 2 + 1 → 3
double (1?2) → (1?2) + (1?2) → 2 + (1?2) → 2 + 2 → 4
```

Somit sind in diesem Fall `2`, `3` und `4` die möglichen Ergebniswerte.

Welche Semantik sinnvoll ist, kann man nicht absolut sagen. Run-time choice liefert zwar alle Werte bezüglich einer Auswertung mittels Termersetzung, allerdings sind dies nicht immer die intuitiv erwarteten. Zum Beispiel würde man nicht erwarten, dass bei einer Operation wie `double` als Ergebnis überhaupt ungerade Zahlen berechnet werden.

Letztendlich muss beim Entwurf einer Programmiersprache eine Entscheidung getroffen werden. Bei Curry (wie auch bei der ähnlichen Sprache TOY [López-Fraguas/Sánchez-Hernández 99]) wird die call-time choice Semantik gewählt. Diese Entscheidung ist u.a. durch folgende Gründe motiviert:

- Call-time Choice erlaubt eine effiziente Implementierung mittels "sharing" (was bei nicht-strikten Sprachen für eine optimale Auswertung sowieso gemacht wird). Zum Beispiel gibt es mit "sharing" nur die beiden folgenden Ableitungen des obigen Ausdrucks:

```
double (1?2) → ·+·   → ·+· → 2
               ↓ ↓      ↓↓
               (1?2)     1

double (1?2) → ·+·   → ·+· → 4
               ↓ ↓      ↓↓
               (1?2)     2
```

- Call-time Choice passt daher gut zu der Strategie der lazy Auswertung, die auf Sharing basiert.
- Call-time Choice führt zu kleineren Suchräumen, da es, wie oben gezeigt, weniger Ableitungsmöglichkeiten gibt.
- Call-time Choice hat häufig die intendierte Bedeutung, wie das obige `double`-Beispiel zeigt.
- Falls man in einer Anwendung tatsächlich bestimmte Argumente an eine Operation mittels Run-time Choice übergeben will, kann man dies mit einer Darstellung dieser Argumente als Operationen erreichen, denn Operationen werden erst dann ausgewertet, wenn sie auf konkrete Argumente angewendet werden. Betrachten wir hierzu noch einmal die Definition von `double`:

```
double :: Int → Int
double x = x + x

d12 = double (1 ? 2)
```

Wenn nun das Argument an `double` mittels Run-time Choice übergeben will, stellen wir das Argument als Operation dar, die bei jedem Vorkommen des Argumentes im Rumpf von `double` zu allen ihren Werten ausgewertet wird:

```
double :: (() → Int) → Int
double rx = rx () + rx ()

d12 = double (\() → 1 ? 2)
```

Durch diese einfache Transformation erhalten wir dann die Werte `2`, `3` und `4` als Ergebnis von `d12`.

4.4.4 Zusammenfassung

Nachdem wir nun wichtige Narrowing-Strategien vorgestellt haben (dies sind aber noch lange nicht alle: in dem Überblicksartikel [Hanus 94] werden über 20 verschiedene Narrowing-Strategien erläutert), wollen wir nun noch einen zusammenfassenden Überblick geben.

Allgemein ist Narrowing die Erweiterung von Reduktion um die Instantiierung freier Variablen, wodurch diese (unter bestimmten Einschränkungen) zu einer vollständigen Auswertungsstrategie wird, d.h. sie findet immer alle oder allgemeinere Lösungen unter der Voraussetzung, dass alle Ableitungswege untersucht werden.

Bezüglich der erlaubten Klasse von Programmen/Termersetzungssystemen und den zu lösenden initialen Ausdrücken/Prädikate gibt es generell zwei Möglichkeiten:

1. Reflexive Gleichheit (wie in der Mathematik üblich) sowie terminierende, konfluente und konstruktorbasierte Termersetzungssysteme: dann kann man Gleichungen mittels Innermost (Basic) Narrowing und Normalisierung lösen.

2. Strikte Gleichheit (d.h. man ist an der Berechnung von Datenobjekten interessiert) und konstruktorbasierte, schwach orthogonale Termersetzungssysteme: dann kann man Gleichungen mittels
   - Needed Narrowing (bei induktiv-sequentiellen Termersetzungssystemen)
   - Weakly Needed Narrowing (bei nicht induktiv-sequentiellen Termersetzungssystemen)

   lösen. Hier ist das "sharing" von Argumenten aus zwei Gründen wichtig:
   a) Zur Vermeidung doppelter Berechnungen und damit zur optimialen Auswertung bei induktiv-sequentiellen Termersetzungssystemen.
   b) Zur Vermeidung nicht-intendierter Ergebnisse, die z.B. bei Run-time Choice ohne Sharing entstehen könnten.

Die wesentliche Charakteristik von Narrowing (im Gegensatz zur Reduktion) ist

- das Raten von Werten für freie Variablen
- die eventuell nichtdeterministische Auswertung von Funktionen ($\leadsto$ unendlich viele Möglichkeiten)

Dies ist allerdings nicht die einzige Möglichkeit, logisch-funktionale Programme auszuwerten. Als mögliche Alternative wurde auch vorgeschlagen, Auswertungen von Funktionen zu verzögern, statt mögliche Werte für unbekannte Argumente zu raten. Diese Strategie, genannt „Residuation", wollen wir im nachfolgenden Kapitel erläutern.