4.6.4 Modellierung nebenläufiger Objekte

Wir wollen in einem kurzen Exkurs zeigen, wie man Residuation einsetzen kann, um die Programmierung mit nebenläufigen Objekten zu unterstützen. Als Beispiel betrachten wir ein Bankkonto als nebenläufiges Objekt. Dieses

- hat einen Zustand (den aktuellen Kontostand) und
- versteht Nachrichten (Einzahlen, Auszahlen, Stand)

In Curry können wir das Bankkonto als Prädikat mit einem Nachrichtenstrom und einem Kontostand als Parameter modellieren:

```
-- Type of messages for the bank account:
data Message = Deposit Int | Withdraw Int | Balance Int

account :: Int  → [Message]  → Bool
account n messages = case messages of
  []                  → True     -- no more messages
  (Deposit  a : ms)   → account (n+a) ms
  (Withdraw a : ms)   → account (n-a) ms
  (Balance  a : ms)   → a==n & account n ms
```

Durch Verwendung von `case` suspendiert das Prädikat `account` auf dem Nachrichtenstrom, solange keine Nachricht vorhanden ist.

Wir können das Bankkontoobjekt verwenden, in dem wir ein Bankkonto mit einer freien Variable als Nachrichtenstrom erzeugen und dieses nebenläufig mit einer Operation verknüpfen, die diesen Nachrichtenstrom mit Nachrichten belegt, wie z.B.

```
account 0 ms  &  ms =:= [Deposit 100, Withdraw 20, Balance b]
```

Dies führt zu der Bindung `b=80`, was den Kontostand nach den beiden Bankkontooperationen darstellt.
Hier repräsentiert `ms` ein Server-Objekt, das man durch Nachrichten aktivieren kann. In diesem Fall wurden alle Nachrichten durch die Unifikation des Gleichheitsconstraint gesendet, aber im Prinzip kann dies auch durch eine Berechnung erfolgen, die diesen Strom schrittweise instantiiert.

Diese Technik kann auch einen Rahmen zur Erweiterung (funktional-) logischer Sprachen um nebenläufige objektorientierte Programmierung bilden, wie dies schon in Oz [Smolka 95] und prototypisch für Curry [Hanus/Huch/Niederau 01] gemacht wurde.

4.7 Erweiterungen

In diesem Abschnitt wollen wir Erweiterungen der bisherigen vorgestellten Konzepte der logisch-funktionalen Programmierung vorstellen, die insbesondere bei der Anwendungsprogrammierung nützlich sind, wie wir später noch sehen werden.

4.7.1 Constraint-Programmierung

Bisher haben wir als einzige Constraints, d.h. Bedingungen zur Beschränkung des möglichen Lösungsraumes, nur das Gleichheitsconstraint "`=:=`" betrachtet. Mit diesem können wir Gleichungen über benutzerdefinierten Funktionen lösen. Zum Beispiel wird für den initialen Ausdruck

```
[1,2]++[x] =:= ys++[2,3]     where x,ys free
```

die einzige Lösung `{x=3, ys=[1]}` berechnet.

Dies funktioniert (mittels Narrowing), weil die beteiligten Funktionen durch explizite Regeln definiert sind. Allerdings funktioniert dies nicht für fest eingebaute Datentypen und ihre Operationen. Z.B. suspendiert (in PAKCS) die Auswertung des Ausdrucks

```
2+x =:= 5   where x free
```

an Stelle der Berechnung der Lösung `{x=3}`.

Hierzu wären spezielle Lösungsverfahren für arithmetische Constraints notwendig. So könnte man mit dem Gaußschen Eliminationsverfahren die Gleichung nach `x` auflösen und so eine Lösung berechnen:

```
2+x =:= 5  ⇝  x =:= 5-2  ⇝  x =:= 3
```

Dies führt zu dem Begriff der **Constraint-Programmierung**:

- erlaube in Programmen die Benutzung von Constraints für spezielle Datentypen, und
- stelle fest eingebaute Constraint-Löser (d.h. Lösungsalgorithmen) für diese Constraints bereit.

Konzeptuell ist dies leicht integrierbar in logische Sprachen, weil diese schon die Konzepte des Rechnens mit partieller Information und Lösungssuche unterstützen. In diesem Fall spricht man auch von **CLP** (Constraint Logic Programming).

Als erstes Beispiel betrachten wir die Constraint-Programmierung mit arithmetischen Constraints über reellen Zahlen:

CLP(R):

- Funktionen auf Gleitkommazahlen: `+ - * /`

- Constraints: `=:= < > <= >=`

  Die Argumente dieser Constraints sind Variablen, Gleitkommazahlen und Anwendungen der obigen Funktionen.

Üblicherweise gibt es keine CLP(R)-Programmiersprache, sondern die Benutzung dieser Constraints erfolgt in Programmiersprachen durch den Import einer passenden Bibliothek. Für die Curry-Implementierung PAKCS gibt es das CLP-Paket `clp-pakcs`, welches mit Hilfe des Curry Package Manager CPM[7] wie folgt installiert werden kann:

```
> cypm add clp-pakcs
```

Hierdurch wird z.B. die Bibliothek `CLP.R` bereitgestellt, die die notwendigen Definitionen enthält:

```
Prelude> :load CLP.R
...
CLP.R>
```

Insbesondere definiert sie einen Datentyp `CFloat`, der eine Instanz der numerischen Klasse `Num` und Constraint-Gleichungen lösen kann. Wenn man also Zahlen vom Typ `CFloat` statt `Float` verwendet, dann wird der Constraint-Löser verwendet:

```
CLP.R> 1.2 + x =:= 2.5   where x free
... Evaluation suspended! ...
CLP.R> 1.2 + x =:= (2.5::CFloat)  where x free
{x=1.3} True
CLP.R> y <= (3::CFloat) & 2 + x >= 0 & y - 5 >= x  where x,y free
{y=3.0, x=-2.0} True
```

Die Typannoation "`::CFloat`" dient also dazu, dass an Stelle der üblichen arithmetischen Operationen die entsprechenden Constraint-Operationen verwendet werden.

Man beachte, dass bei der Lösung der ersten Gleichung das Gaußsche Eliminationsverfahren zur Anwendung gekommen ist, während für die Lösung des letzten Ausdrucks, der Ungleichungen enthält, das Simplexverfahren zur Anwendung kommt. Als Constraint-Löser sind also schon recht komplexe Verfahren eingebaut.

Beispiel: **Hypothekenberechnung**
Für die Beschreibung aller nötigen Zusammenhänge beim Rechnen mit Hypotheken verwenden wir folgende Parameter:

- `p`: Kapital
- `t`: Laufzeit in Monaten

[7] `http://www.curry-lang.org/tools/cpm`

- `ir`: monatlicher Zinssatz
- `r`: monatliche Rückzahlung
- `b`: Restbetrag

Die Zusammenhänge lassen sich in Curry unter Verwendung der CLPR-Bibliothek wie folgt ausdrücken:

```
import CLP.R

mortgage :: CFloat → CFloat → CFloat → CFloat → CFloat → Bool

mortgage p t ir r b
  | t >= 0 & t <= 1  -- lifetime not more than 1 month?
  =  b =:= p * (1 + t * ir) - t * r

mortgage p t ir r b
  | t > 1               -- lifetime more than 1 month?
  =  mortgage (p * (1 + ir) - r) (t - 1) ir r b
```

Dann können wir z.B. die notwendige monatliche Rückzahlung zur Tilgung einer Hypothek berechnen:

```
> mortgage 100000 180 0.01 r 0.0      where r free
r = 1200.168
```

Wir können aber auch die Laufzeit zur Finanzierung einer Hypothek berechnen:

```
> mortgage 100000 time 0.01 1400 0     where time free
time = 125.901
```

Als weiteres Beispiel betrachten wir die Constraint-Programmierung über endlichen Bereichen:

CLP(FD):

- Variablen haben einen endlichen Wertebereich (Intervall ganzer Zahlen)
- Constraints: Gleichungen, Ungleichungen, Elementbeziehungen, kombinatorische Verknüpfungen
- Anwendung: Lösung schwieriger kombinatorischer Problem (z.B. Personalplanung, Fertigungsplanung, Stundenpläne), viele industrielle Anwendungen

Da die zu lösenden Probleme typischerweise NP-vollständig sind, benutzt man im Gegensatz zu CLP(R) keinen vollständigen Lösungsalgorithmus, sondern verwendet ein unvollständiges Verfahren, wobei man mit einer Heuristik Lösungen rät:

- Man verwendet OR-Methoden (Operations Research) zur lokalen Konsistenzprüfung.
- Die globale Konsistenz wird durch Aufzählung möglicher Werte sichergestellt (Motto: "constrain-and-generate").

Allgemeines Vorgehen hierbei:

1. Definiere den möglichen Wertebereich der beteiligten Variablen.
2. Beschreibe die Constraints, die diese Variablen erfüllen müssen.
3. Zähle die Werte im Wertebereich auf, d.h. binde die Variablen an konkrete Werte. Hierduch werden Constraints aktiviert, die dann den weiteren Suchraum stark einschränken.

Beispiel: **Krypto-arithmetisches Puzzle**
Gesucht ist eine Zuordnung von Buchstaben zu Ziffern, sodass verschiedene Buchstaben verschiedenen Ziffern entsprechen und die folgende Rechnung stimmt:

```
  send
+ more
------
 money
```

Das Curry-Paket `clp-pakcs` enthält auch die Bibliothek `CLP.FD`, mit der wir dieses Problem wie folgt lösen können:

```
import CLP.FD

-- "send more money" puzzle in Curry with FD constraints:
smm :: [Int]
smm =
 let xs@[s,e,n,d,m,o,r,y] = take 8 (domain 0 9)
  in solveFD [] xs $
        s ># 0 /\
        m ># 0 /\
        allDifferent xs /\
                       1000 * s + 100 * e + 10 * n + d
                     + 1000 * m + 100 * o + 10 * r + e
        =# 10000 * m + 1000 * o + 100 * n + 10 * e + y
```

Damit können wir nun eine Lösung sehr schnell berechnen:

```
> smm
[9,5,6,7,1,0,8,2]
```

Anmerkungen:

- FD-Variablen haben keine `Int`-Zahlen als Werte, sondern endliche Bereiche, d.h. sie haben den Typ `FDExpr`, der allerdings eine Instanz von `Num` ist, sodass man die üblichen arithmetischen Funktionen verwenden kann.

- Zur Erzeugung von FD-Variablen eines bestimmten Wertebereichs gibt es den Generator `domain min max`, welcher eine unendliche Liste von FD-Variablen mit dem Wertebereich `[min..max]` liefert.

- Zur Unterscheidung von FD-Relationen von den üblichen arithmetischen Relationen werden die FD-Relationen mit dem Suffix `#` aufgeschrieben. Außerdem steht `/\` für die Konjunktion von FD-Relationen.

- Kombinatorische Constraints haben spezielle Lösungsalgorithmen zur Einschränkung des Suchraumes. Z.B. steht

  ```
  allDifferent xs
  ```

  für das Constraint „alle Variablen der Liste `xs` haben verschiedene Werte“. Prinzipiell wäre dies auch ausdrückbar durch eine Liste von „paarweise verschieden“-Constraints, aber dieses Constraint implementiert eine bessere (globalere) Einschränkung.

- Der Constraint-Löser wird durch die Operation `solveFD` aufgerufen, die eine Liste von Optionen (s.u.), eine Liste von FD-Variablen, für die Lösungen berechnet werden sollen, und ein zu lösendes FD-Constraint als Eingabe erhält und nichtdeterministisch eine Lösung, d.h. eine Liste von Werten für die FD-Variablen, zurückgibt.

  Die Optionen beinhalten Strategien zur Belegung von FD-Variablen mit Werten. Beispielsweise wird durch die Option `FirstFail` die Variable mit dem kleinsten noch möglichen Wertebereich zuerst an einen ihrer Werte gebunden.

Als weiteres Beispiel betrachten wir das **Lösen eines Su Doku-Puzzles**:
Hierbei ist z.B. das folgende 9×9-Feld gegeben:

9			2			5		
	4			6			3	
		3						6
			9			2		
				5			8	
		7			4			3
7						1		
	5			2			4	
		1			6			9

Die Aufgabe ist, die leeren Felder mit den Ziffern 1 bis 9 so zu füllen, dass alle Zeilen, Spalten und 3×3-Felder keine doppelten Vorkommen haben.

Dies können wir in Curry lösen, in dem wir die Su Doku-Matrix als Liste von Listen (Typ `[[Int]]`) darstellen und das Constraint `allDifferent` auf die entsprechenden Teillisten anwenden:

```
import CLP.FD
import Data.List(transpose)

-- Solving a Su Doku puzzle represented as a matrix of numbers (possibly free
-- variables):
sudoku :: [[FDExpr]] → [Int]
sudoku m = solveFD [FirstFail] (concat m) $
 allC allDifferent m  /\             -- all rows contain different digits
 allC allDifferent (transpose m) /\  -- all columns have different digits
 allC allDifferent (squares m)       -- all 3x3 squares are different
 where
  -- translate a matrix into a list of small 3x3 squares
  squares :: [[a]] → [[a]]
  squares []            = []
  squares (l1:l2:l3:ls) = group3Rows [l1,l2,l3] ++ squares ls

  group3Rows l123 = if null (head l123) then [] else
   concatMap (take 3) l123 : group3Rows (map (drop 3) l123)
```

Hierbei ist `allC` die Konjunktion aller Anwendungen eines FD-Constraints auf eine Liste von Werten, die folgt vordefiniert ist:

```
allC c xs = foldr (/\) true (map c xs)
```

Damit können wir ein konkretes Su Doku durch Angabe der gegebenen Felder und Variablen für die unbekannten Felder lösen:

```
> sudoku [[9,x12,x13,2,x15,x16,5,x18,x19],...
x12=6
x13=8
⋮
```

Schöner ist es, ein konkretes Su Doku in einer Textmatrixdarstellung zu lesen und die Lösung formatiert auszugeben, was aber in Curry auch in vier Codezeilen erledigt werden kann.