Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Integration eines Finite-Domain-Constraint-Solvers in KiCS2

Jan Rasmus Tikovsky

August 2012

Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

betreut von
Prof. Dr. Michael Hanus
Björn Peemöller

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Kiel, den 2. August 2012

_____

Jan Rasmus Tikovsky

Zusammenfassung

Ein Constraint-Programming-System besteht üblicherweise aus zwei Komponenten: einer Modellierungssprache zur Spezifikation eines Constraint-Problems und einer Lösungskomponente, die durch Anwendung spezieller Algorithmen ein gegebenes Constraint-Modell löst. Die funktional-logische Programmiersprache Curry ist aufgrund ihres deklarativen Stils gut für die Einbettung einer Constraint-Modellierungssprache geeignet. Allerdings ist die direkte Realisierung eines zustandsbehafteten Constraint-Solvers in Curry aufgrund der Seiteneffektfreiheit dieser Sprache schwierig. Diese Arbeit beschreibt, wie man die KiCS2-Curry-Implementierung, die Curry-Programme in rein funktionale Haskell-Programme übersetzt, durch Integration der Solver eines funktionalen Constraint-Programming-Frameworks um eine Bibliothek zur Modellierung und Lösung von Finite-Domain-Constraints erweitern kann. In einem weiteren Entwicklungsschritt wird von den konkreten Finite-Domain-Constraints und -Solvern abstrahiert und gezeigt, wie man generische Schnittstellen zur Einbettung von Constraints und FD-Solvern in KiCS2 implementiert.

Inhaltsverzeichnis

1. Einführung **1**

2. Grundlagen **3**

2.1. Curry . . . 3

2.1.1. Datentypen . . . 3

2.1.2. Funktionen . . . 4

2.1.3. Nicht-deterministische Funktionen . . . 5

2.1.4. Freie Variablen und Unifikation . . . 6

2.1.5. Constraint-Programmierung . . . 8

2.1.6. Operationelle Semantik . . . 10

2.2. KiCS2 . . . 11

2.2.1. Darstellung des Nicht-Determinismus . . . 11

2.2.2. Unifikation . . . 14

2.2.3. Auswertung von (nicht-deterministischen) Ausdrücken . . . 17

2.3. Monadic-Constraint-Programming-Framework . . . 20

2.3.1. Allgemeines . . . 20

2.3.2. Finite-Domain-Schnittstelle . . . 24

3. Vorüberlegungen und Grundlegende Idee **30**

4. Implementierung **33**

4.1. Entwicklung einer Finite-Domain-Constraint-Bibliothek für KiCS2 . . . 33

4.1.1. KiCS2-CLPFD-Bibliothek . . . 33

4.1.2. Darstellung der Constraints in Haskell . . . 36

4.1.3. Einsammeln aller Finite-Domain-Constraints . . . 46

4.1.4. Kombination von Finite-Domain-Constraints mit der Curry-Unifikation . . . 50

4.2. Anschluss der FD-Constraint-Solver des MCP-Frameworks . . . 63

4.2.1. Übersetzung der Constraints in die MCP-Modellierungssprache . . . 63

4.2.2. Konstruktion und Lösung von MCP-Baummodellen . . . 70

4.2.3. Ausgabe der vom Solver bestimmten Lösungen durch KiCS2 . . . 75

4.3. Entwicklung generischer Schnittstellen zur Unterstützung weiterer Constraints und FD-Solver . . . 79

4.3.1. FD-Term-Interface . . . 79

4.3.2. FD-Solver-Interface . . . 81

4.3.3. Generische Constraint-Schnittstelle . . . 84

5. Evaluation **95**

6. Zusammenfassung und Ausblick **97**

A. Installation und Benutzung der KiCS2-CLPFD-Bibliothek **99**
A.1. Installation 99
A.2. Benutzung 100

B. Auszüge aus dem Monadic-Constraint-Programming-Framework **102**

C. MCP-Solver-Implementierung **110**

D. Beispiel für Sonderbehandlung der Curry-Bindungs-Constraints hinsichtlich des Wrappings **118**

E. Generate & Test-Realisierung des N-Damen-Problems **121**

Listings

1. Datentypdeklaration in Curry 3
2. Deklaration polymorpher Datentypen in Curry 3
3. Funktionsdefinition in Curry 5
4. Curry-Gleichheits-Constraint 6
5. Explizite Darstellung des Nicht-Determinismus in KiCS2 11
6. Propagieren des Nicht-Determinismus 12
7. Einführung eines Identifikators für `Choice`s 13
8. Realisierung von freien Variablen in KiCS2 13
9. Einführung eines `IDSupply` zur Erzeugung neue Identifikatoren 14
10. Erster Ansatz zur Implementierung der Unifikation in KiCS2 15
11. Unterscheidung von freien Variablen und Standard-`Choice`s 15
12. Narrowing von freien Variablen bei Funktionsanwendung 16
13. Definition von Bindungs-Constraints und -entscheidungen 16
14. Einführung von `Guard`-Ausdrücken 16
15. Verbesserte Implementierung der Unifikation 17
16. Beispiel: Repräsentation eines Bools in KiCS2 18
17. Generische `Try`-Struktur für das Pattern Matching 18
18. Transformation eines Bool in die generische `Try`-Struktur 18
19. Funktionen zum Zugriff auf den globalen Decision-Store 18
20. Auswertung eines nicht-deterministischen Ausdrucks mit der Tiefensuche 19
21. Repräsentation von Constraint-Modellen 21
22. Erweiterung monadischer Baummodelle 21
23. Syntaktischer Zucker für Baummodelle 22
24. Interface für Constraint-Solver im MCP-Framework 22
25. Auswertung von Constraint-Modellen 23
26. Repräsentation von FD-Variablen 25
27. Typen für Integer- und Listen-Ausdrücke sowie FD-Constraints im MCP-Framework 25

28. Konstruktorfunktionen für FD-Constraints und -Ausdrücke . . . 25
29. Beispiele für Implementierung der Constraint-Konstruktorfunktionen . . . 26
30. Constraint-Konstruktorfunktionen zur direkten Erzeugung eines MCP-Baummodell-Knotens . . . 26
31. MCP-`FDSolver`-Interface (Ausschnitt) . . . 27
32. Wrapper für MCP-FD-Solver . . . 27
33. `Solver`-Instanz für den FD-Solver-Wrapper `(FDInstance s)`(Ausschnitt) . . . 27
34. Modellierung des N-Damen-Problems mit dem MCP-Framework . . . 28
35. CLPFD-Bibliothek . . . 33
36. Addition auf FD-Termen (Curry) . . . 35
37. Domain-Constraint (Curry) . . . 35
38. Beipiel: 4-Damen-Problem . . . 36
39. Repräsentation von FD-Termen (Haskell) . . . 37
40. Repräsentation von FD-Constraints (Haskell) . . . 37
41. Erweiterung des Datentyps `Constraints` . . . 38
42. Repräsentation vom Curry-Typ `Int` in Haskell . . . 39
43. Umwandlung von Curry-FD-Termen in Haskell-FD-Terme . . . 39
44. Beispiel: `fromCurry` für `C_Bool` . . . 40
45. Repräsentation vom Curry-Typ `[a]` in Haskell . . . 40
46. Umwandlung von Curry-FD-Listen in Haskell-FD-Listen . . . 40
47. Addition auf FD-Termen (Curry) (angepasst) . . . 41
48. Addition auf FD-Termen (Haskell) . . . 41
49. Gleichheit auf FD-Termen (Haskell) . . . 42
50. Domain-Constraint (Haskell) . . . 43
51. Typklasse `GNFChecker` . . . 43
52. `GNFChecker`-Instanzen . . . 44
53. Addition auf FD-Termen (Haskell) - erweitert . . . 44
54. Beispiel: 2-Damen-Problem (Curry) . . . 45
55. Beispiel: Guard-Ausdrücke für 2-Damen-Problem (Haskell) . . . 45
56. Beispiel: Auswertung zu Guard-Ausdrücken . . . 46
57. Einsammeln der FD-Constraints . . . 47
58. Erweiterte Normalform-Berechnung . . . 49
59. Beispiel: N-Damen-Problem . . . 51
60. Wiederholung: Auswertung von Listen-Argumenten in Constraint-Funktionen . . . 52
61. Beispiel: SEND-MORE-MONEY (Curry) . . . 53
62. Repräsentation von $[S, E, N, D, M, O, R, Y] =:= l$ in Haskell . . . 54
63. SEND-MORE-MONEY (Haskell) - Ausschnitt . . . 54
64. `bind`-Implementierung für `C_Bool` . . . 55
65. `bind`-Implementierung für `OP_List a` . . . 56
66. Typklasse `FromDecisionTo` . . . 57
67. `lookupValue`-Funktion . . . 57

68. `FromDecisionTo`-Instanz für `C_Bool` . . . 57
69. `FromDecisionTo`-Instanz für `OP_List a` . . . 58
70. Funktion zur Aktualisierung von FD-Variablen . . . 60
71. Aktualisierung von FD-Constraints . . . 60
72. Bisherige Realisierung von `(&)` . . . 61
73. Angepasste Realisierung von `(&)` . . . 61
74. Translation State . . . 64
75. Übersetzung von FD-Termen . . . 65
76. Erzeugung einer MCP-Integer-Variable . . . 65
77. Übersetzung von Listen von FD-Termen . . . 66
78. Übersetzung der `FDConstraints` 1 . . . 66
79. Übersetzung der `FDConstraints` 2 . . . 67
80. Datenstruktur zur Zwischenspeicherung von Labeling Informationen . . . 68
81. Translation State (angepasst) . . . 68
82. Übersetzung der `FDConstraints` 3 . . . 68
83. Übersetzungsfunktion . . . 69
84. Beispiel: 2-Damen-Problem (MCP-Modellierungssprache) . . . 69
85. Umwandlung einer Liste von FD-Constraints in ein MCP-Baummodell . . . 70
86. Labeling Interface für FD-Solver (Ausschnitt) . . . 71
87. MCP-Baummodell: `Label`-Knoten . . . 72
88. Labeling von MCP-Baummodellen . . . 73
89. Aufruf der MCP-Solver . . . 73
90. Datenstruktur zur Speicherung von Lösungsinformationen . . . 74
91. Aufruf des Overton-Solvers . . . 74
92. Aufruf des Overton-Solvers (Auswertung mit Tiefensuche) . . . 77
93. Bindung der Constraint-Lösungen . . . 77
94. Bindung der Labeling-Variablen . . . 78
95. Interface für durch Constraints beschränkbare Typen . . . 80
96. Beispielinstanz zur Erzeugung von Integer-FD-Termen . . . 80
97. Funktion zur Aktualisierung von FD-Variablen (angepasst) . . . 80
98. Angepasste Typsignaturen für Funktionen zur Bindung von Constraint-Lösungen . . . 81
99. Interface zur Integration von FD-Solvern . . . 81
100. Interface zur Integration von FD-Solvern - associated types . . . 82
101. Interface zur Integration von FD-Solvern - Funktionen 1 . . . 82
102. Interface zur Integration von FD-Solvern - Funktionen 2 . . . 83
103. Integration der MCP-Solver über die FD-Solver-Schnittstelle . . . 83
104. Wrapper für Constraints . . . 85
105. Interface für erweiterbaren Constraint-Typ . . . 85
106. `cast`-Funktion und Default-Implementierung des Interface für erweiterbaren Constraint-Typ . . . 86
107. Anpassung des Datentyps `Constraints` . . . 87

108. Implementierung des `WrappableConstraint`-Interfaces für FD-Constraints 87
109. Externe FD-Constraint-Funktionen (angepasst) 87
110. Einsammeln von `WrappedConstraints` 88
111. Anpassung des FD-Solver-Interfaces 89
112. Solver für `WrappedConstraints` 90
113. Bislang unterstützte Constraint-Solver in KiCS2 90
114. Filtern der heterogenen Constraint-Liste 90
115. Lösung aller “eingepackten“ Constraints 91
116. Lösung aller “eingepackten“ Constraints während der KiCS2-Tiefensuche 92
117. Alternativer Typ für `Guard`-Ausdrücke 93
118. `WrappableConstraint`-Instanz für Curry-Bindungs-Constraints 93
119. Repräsentation von Integer-Ausdrücken 102
120. MCP-Collections 102
121. MCP-FD-Constraints 102
122. MCP-FD-Constraint-Konstruktorfunktionen 103
123. `FDSolver`-Typklasse 105
124. `EnumTerm`-Typklasse 107
125. MCP-`labelling`-Funktion 108
126. MCP-`assignments`-Funktion 108
127. MCP-Labeling-Strategien 108
128. MCP-Hilfsfunktionen zur Implementierung von `labelWith` 109
129. Einführung eines Identifiers für FD-Listen 110
130. Angepasste `FDConstraints` 110
131. Beispiel für Problem bei Kombination von Curry-Bindungs-Constraints und FD-Constraints 118
132. Generate & Test Realisierung des N-Damen-Problems 121

Abbildungsverzeichnis

1. Lösung für das 4-Damen-Problem 9
2. Vorgang zum Lösen von Finite-Domain-Constraints in KiCS2 32
3. Suchbaum **vor** Aufruf von `searchFDCs` 49
4. Suchbaum **nach** Aufruf von `searchFDCs` 50
5. Ablauf der Bindung und Rekonstruktion eines Wertes 59
6. Benchmarks: Unifikation in KiCS2 ohne und mit CLPFD-Erweiterung 95
7. Benchmarks: Performance der Solver im Vergleich 96
8. Ausschnitt des Suchbaums **vor** Aufruf von `searchWrappedCs` 119
9. Ausschnitt des Suchbaums **nach** Aufruf von `searchWrappedCs` (Wrapping der Curry-Bindungs-Constraints) 119
10. Ausschnitt des Suchbaums **nach** Aufruf von `searchWrappedCs` (separate Behandlung der Curry-Bindungs-Constraints) 119

1. Einführung

Constraint-Programming befasst sich mit der Lösung von Problemen durch Spezifikation von Eigenschaften und Bedingungen (= Constraints), die von möglichen Lösungen dieses Problems erfüllt werden müssen. Man definiert diese Bedingungen deklarativ durch das Aufstellen von Regeln für Variablen, die einen endlichen oder unendlichen Wertebereich besitzen. Die Modellierung eines Problems mit endlichem Wertebereich wird auch als Finite-Domain-Constraint-Programming bezeichnet.

Ein Constraint-Programming-System besteht im Wesentlichen aus zwei Elementen: einer Modellierungskomponente, mit deren Hilfe das zu lösende Problem beschrieben wird, und einer Lösungskomponente, einem sogenannten Constraint-Solver, der nach Lösungen für das beschriebene Problem sucht. Dazu speichert, kombiniert und vereinfacht der Solver die Constraints eines Modells mit Hilfe spezieller Algorithmen. Mit Hilfe der Labeling-Technik "probiert" er Variablenbelegungen aus und propagiert diese (Constraint-Propagierung), um auf diese Weise den Wertebereich weiterer Constraint-Variablen einzuschränken. Diese Techniken werden iterativ angewandt, bis entweder eine gültige Belegung für alle Constraint-Variablen gefunden wurde oder bis ein Widerspruch auftritt. In diesem Fall "springt" der Solver in einen konsistenten Zustand zurück und probiert eine andere Belegung aus (Backtracking).

Zu den Anwendungsgebieten des Constraint-Programmings zählen die Erstellung von Stunden-, Fahr- und Personaleinsatzplänen. Neben logischen Sprachen wie PROLOG, die spezielle Constraint-Programming-Bibliotheken zur Verfügung stellen, gibt es auch eine ganze Reihe von Softwaresystemen zur Modellierung und Lösung von Constraint-Problemen. Beispiele hierfür sind die auf C++-basierende Solver-Bibliothek Gecode (Generic Constraint Development Environment [3]) oder das in der Objekt-orientierten Programmiersprache Java realisierte TAILOR-Tool ([4]), das in der Solver-unabhängigen Modellierungssprache Essence' modellierte Constraint-Probleme mit Hilfe des Minion oder Gecode-Solvers löst.
Im Bereich der funktionalen Programmierung haben Tom Schrijvers, Peter Stuckey und Philip Wadler mit ihrem Monadic-Constraint Programming-Framework (kurz: MCP-Framework, [9]) gezeigt, wie man einen Finite-Domain-Constraint-Solver mit Hilfe von Monaden in der seiteneffektfreien Sprache Haskell implementieren kann.

Auf den ersten Blick ist auch die funktional-logische Programmiersprache Curry gut für die Einbettung einer Constraint-Modellierungssprache geeignet: So unterstützt Curry die Programmierung mit freien Variablen und bietet einen deklarativen Programmierstil. Allerdings gilt in Curry das Prinzip der referentiellen Transparenz, es handelt sich also um eine zustandslose, seiteneffektfreie Sprache. Dies erschwert die direkte Realisierung stark zustandsbehafteter Constraint-Solver in Curry.
Diese Arbeit hat sich zum Ziel gesetzt, die KiCS2-Curry-Implementierung, die funktional-logische Programme in rein funktionale Haskell-Programme übersetzt, um die Möglichkeit zur Programmierung mit Finite-Domain-Constraints zu erweitern. Dazu soll eine Finite-Domain-Constraint-

Bibliothek für KiCS2 entwickelt werden. Gelöst werden sollen diese Constraints mit Hilfe der FD-Solver des Monadic-Constraint-Programming-Frameworks ([9]), die daher in die KiCS2-Implementierung integriert werden sollen. Schließlich sollen die wichtigsten Erkenntnisse aus dieser Integration in die Entwicklung generischer Schnittstellen zur Einbindung weiterer Constraints und Constraint-Solver in KiCS2 einfließen. In einem letzten Schritt sollen diese Schnittstellen beispielhaft für die zuvor entwickelte FD-Constraint-Bibliothek und die MCP-FD-Solver implementiert werden.

Das nächste Kapitel liefert eine kurze Übersicht über die wichtigsten Grundlagen, welche zum Verständnis dieser Arbeit erforderlich sind. Dazu werden in einzelnen Unterkapiteln die Programmiersprache Curry, die KiCS2-Curry-Implementierung sowie das Monadic-Constraint-Programming-Framework vorgestellt. Kapitel 3 beschreibt die grundlegende Idee für die Implementierung von Finite-Domain-Constraints in KiCS2. In Kapitel 4 wird die Implementierung vorgestellt. Es ist in drei größere Unterabschnitte unterteilt, die sich mit der Entwicklung einer CLPFD-Bibliothek für KiCS2, der Integration der Finite-Domain-Solver des MCP-Frameworks sowie der Entwicklung generischer Schnittstellen zur Integration weiterer Constraints und Constraint-Solver befassen. Kapitel 5 evaluiert die hier vorgestellte Implementierung mit Hilfe geeigneter Benchmarks. Das letzte Kapitel liefert eine Zusammenfassung sowie einen Ausblick auf mögliche Weiterentwicklungen.

2. Grundlagen

2.1. Curry

Dieser Abschnitt liefert eine kurze Einführung in die funktional-logische Programmiersprache Curry. Dabei werden nur die wichtigsten und für das Verständnis dieser Arbeit relevanten Features dieser Sprache vorgestellt. Eine ausführlichere Einführung findet man in [6].

Curry ist eine deklarative Programmiersprache. Im Unterschied zu imperativen Sprachen wird in deklarativen Sprachen nicht der Weg zur Lösung eines Problems, sondern vielmehr das Problem selbst beschrieben. Im direkten Vergleich sind deklarative Programme häufig deutlich kompakter, besser verständlich und damit auch weniger fehleranfällig als imperative Lösungen des gleichen Problems.
Curry wird auch als funktional-logische Sprache bezeichnet, da sie mit der funktionalen und logischen Programmierung zwei der wichtigsten deklarativen Programmierparadigmen vereint. So unterstützt Curry auf der einen Seite funktionale Konzepte wie u.a. Funktionen höherer Ordnung oder die bedarfsgesteuerte Auswertung von Ausdrücken (lazy evaluation), andererseits ermöglicht diese Sprache aber auch die Programmierung mit logischen Variablen, partiellen Datenstrukturen und nicht-deterministischen Funktionen.
Die Syntax von Curry-Programmen weist eine große Ähnlichkeit mit der der funktionalen Programmiersprache Haskell auf. Zusätzlich können in Curry-Ausdrücken jedoch auch noch freie (logische) Variablen verwendet werden. Ein Curry-Programm besteht im Allgemeinen aus einer Menge von Datentyp- und Funktionsdefinitionen.

2.1.1. Datentypen

Neue Datentypen werden in Curry mit dem Schlüsselwort **data** definiert:

```
data t = C1 t11 ... t1n1 | ... | Ck tk1 ... tknk
```

Listing 1: Datentypdeklaration in Curry

Mit der obigen Deklaration wird ein neuer Datentyp `t` mit k unterschiedlichen Konstruktoren $\texttt{C}_1, \ldots, \texttt{C}_k$ definiert. Die $\texttt{t}_{i1_i}, \ldots, \texttt{t}_{in_i}$ sind dabei mögliche Argumenttypen der Konstruktoren, das heißt, es gilt: $\texttt{C}_i$ `::` $\texttt{t}_{i1_i}$ `->` `...` `->` $\texttt{t}_{in_i}$ `->` `t` für alle $i \in \{1, ..., k\}$.
Neben diesen einfachen Datentypdeklarationen bietet Curry auch die Möglichkeit, polymorphe Datenstrukturen zu definieren:

```
data t a1 ... an = C1 t11 ... t1n1
                 | ...
                 | Ck tk1 ... tknk
```

Listing 2: Deklaration polymorpher Datentypen in Curry

Zur Definition von polymorphen Datenstrukturen verwendet man sogenannte Typkonstruktoren. In der obigen Deklaration ist `t` ein Typkonstruktor mit n Typvariablen $\texttt{a}_1$, `...`, $\texttt{a}_n$. Zur Konstruktion eines neuen Typs werden diese Typvariablen mit konkreten Typen belegt.
Im folgenden sollen einige Beispiele für Datentypdeklarationen betrachtet werden:

```
data Color      = Red | Blue | Yellow
data Maybe a    = Nothing | Just a
data BinTree a = Empty | Branch (BinTree a) a (BinTree a)
```

Das erste Beispiel führt einen einfachen Datentyp `Color` zur Repräsentation der drei Grundfarben ein. Hierbei handelt es sich um einen sogenannten Aufzählungstyp, da keiner der drei Konstruktoren Argumente erwartet.
Der polymorphe Datentyp `Maybe a` bietet die Möglichkeit, Typen mit optionalen Werten zu definieren. Beispielsweise könnte man durch die entsprechende Belegung der Typvariablen `a` den neuen Typ `Maybe Color` für optionale Farben einführen. Mögliche Werte dieses Typs sind dann unter anderem `Just Red` oder `Nothing`.
Das letzte Beispiel führt einen polymorphen Datentyp zur Repräsentation von Binärbäumen mit beliebigen Knotenelementen ein. Dabei ist `BinTree a` eine rekursive Datenstruktur, denn der Konstruktor `Branch` zur Definition einer Verzweigung im Baum erhält unter anderem wieder zwei Argumente vom Typ `BinTree a` zur Darstellung des linken bzw. rechten Kindbaums. Definiert man sich einen Binärbaum für `Int`-Werte, so könnte ein einfaches Beispiel mit drei Elementen wie folgt aussehen:

```
simpleIntTree :: BinTree Int
simpleIntTree =
  Branch (Branch Empty 2 Empty) 1 (Branch Empty 3 Empty)
```

Mit dem Schlüsselwort **type** können wie in Haskell Typsynonyme definiert werden. Damit hat man zum einen die Möglichkeit, komplexe Typbezeichnungen durch einfachere zu ersetzen und so den Code lesbarer zu gestalten. Zum anderen kann man aber auch einfachen Typen einen anderen Namen zuordnen, um auf diese Art die Verständlichkeit zu erhöhen. Die folgenden Beispiele verdeutlichen dies:

```
type IntTree = BinTree Int
type String = [Char]

type Name = String
type PhoneNumber = String
type PhoneBook = [(Name,PhoneNumber)]
```

2.1.2. Funktionen

Eine Funktionsdefinition in Curry hat allgemein die folgende Form:

```
f p1 ... pn = e
```

Listing 3: Funktionsdefinition in Curry

Hierbei ist `f` der Funktionsname, $\texttt{p}_1$`, ...,` $\texttt{p}_n$ sind formale Parameter und `e` ist der Funktionsrumpf. Mögliche Rumpfausdrücke sind unter anderem Zahlen, Basisoperationen wie die Addition (`1+2`), Funktionsanwendungen (`g` $\texttt{e}_1$ `...` $\texttt{e}_m$, wobei $\texttt{e}_1$ `...` $\texttt{e}_m$ selbst wieder Ausdrücke sind) oder bedingte Ausdrücke der Form `if b then` $\texttt{e}_1$ `else` $\texttt{e}_2$ (wobei `b` ein boolescher und $\texttt{e}_1$`,`$\texttt{e}_2$ einfache Ausdrücke sind).
Zusätzlich kann bei der Funktionsdefinition eine Typsignatur angegeben werden, um die Typen der Parameter und den Ergebnistyp festzulegen. Obwohl Curry eine streng getypte Sprache ist, ist die Angabe dieser Typsignaturen optional. Verzichtet man darauf, so wird der Typ durch einen Curry-Compiler wie KiCS2 oder PAKCS inferiert.
Als Beispiel für eine Funktionsdefinition wird im folgenden die Fakultätsfunktion betrachtet:

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

Wie Haskell bietet Curry auch die Möglichkeit, Funktionen mittels Pattern-Matching zu implementieren. Hierbei wird eine Funktion durch Angabe einer Menge von `k` definierenden Regeln der Form `f` $\texttt{pat}_{i1}$ `...` $\texttt{pat}_{in}$ `=` $\texttt{e}_i$ definiert mit $i \in \{1, ..., k\}$. Durch die Muster auf der linken Seite einer definierenden Regel spezifiziert man, für welchen Fall die entsprechende Regel und damit der Ausdruck auf der rechten Regelseite angewandt werden soll. Als Muster $\texttt{pat}_{ij}$ (mit $i \in \{1, ..., k\}$ und $j \in \{1, ..., n\}$) können unter anderem Variablen (matchen immer), "wild cards"der Form _ (Unterstrich) (matchen auch immer, allerdings ohne Bindung) oder Konstruktorterme (matchen auf den jeweiligen Fall) verwendet werden. Dabei ist allerdings zu beachten, dass nicht mehrfach auf die gleiche Variable in einer Regel gematcht werden darf und dass Regeln mit spezielleren Mustern vor Regeln mit allgemeineren Mustern definiert werden. Die letzte Regel sollte zudem immer so allgemein sein, dass sie alle Fälle auffängt, damit es später nicht zu Laufzeitfehlern kommt.
Diese sehr deklarative Form der Programmierung entspricht häufig der mathematischen Spezifikation eines Problems wie beispielsweise die Implementierung der Fakultätsfunktion mittels Pattern-Matching zeigt:

```
facPM :: Int -> Int
facPM 0 = 1
facPM n = n * facPM (n-1)
```

2.1.3. Nicht-deterministische Funktionen

Während es in Haskell vermieden werden sollte, Regeln mit überlappenden linken Regelseiten zu definieren, ist dies in Curry zulässig. Auf diese Weise kann man nicht-deterministische Funktionen definieren, die mehrere Lösungen liefern.

Das einfachste Beispiel für eine solche nicht-deterministische Funktion ist der `(?)`-Operator, der die Wahl zwischen zwei Möglichkeiten repräsentiert. Statt überlappender Regeln kann man auch diesen Operator verwenden, wenn für eine Regel mehrere Möglichkeiten auf der rechten Regelseite definiert werden sollen.
Die Funktion `insert` fügt ein Element an einer beliebigen Stelle in einer Liste ein. So liefert der Aufruf `insert 42 [1,2,3]` die folgende Lösungsmenge `{[42,1,2,3],[1,42,2,3],[1,2,42,3], [1,2,3,42]}`. Mit Hilfe dieser Funktion lässt sich dann als weiteres Beispiel sehr leicht die Permutation von Listen implementieren.

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y

insert :: a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = (x:y:ys) ? (y : insert x ys)

permut :: [a] -> [a]
permut [] = []
permut (x:xs) = insert x (permut xs)
```

2.1.4. Freie Variablen und Unifikation

Wie eingangs erwähnt unterstützt Curry auch Elemente der logischen Programmierung. Dazu gehören die Verwendung von freien Variablen und speziellen Constraints (z.B. für die Unifikation) in Programmen.
Zur Definition der Constraints wurde der Datentyp `Success` in Curry eingeführt. Dieser spezielle Typ hat keinerlei Werte, sondern repräsentiert nur den Erfolg eines Constraints. Ein Beispiel für ein solches Constraint ist das unten angegebene Gleichheits-Constraint `(=:=)` (auch Unifikationsoperator). Es entspricht der strikten Gleichheit auf Termebene, das heißt, der Ausdruck $e_1 =:= e_2$ wird genau dann zu `Success` ausgewertet, wenn e_1 und e_2 zu unifizierbaren Konstruktortermen reduzierbar sind.
Mit `(&)` lassen sich mehrere Constraints zu einer Konjunktion von Constraints verknüpfen:

```
data Success = Success

(=:=) :: a -> a -> Success
(&)   :: Success -> Success -> Success
```

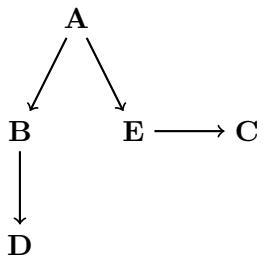
Listing 4: Curry-Gleichheits-Constraint

Das folgende kleine Beispiel mit einem gerichteten Graph soll nun verdeutlichen, wie das Programmieren mit Constraints und freien Variablen in Curry funktioniert:

```
data Node = A | B | C | D | E

edge :: Node -> Node
edge A = B
edge A = E
edge B = D
edge E = C

path :: Node -> Node -> Success
path x y = edge x =:= y
path x y = edge x =:= z & path z y
  where z free
```



Mit der Funktion `edge` werden die Kanten des Graphen definiert: `edge A = B` bedeutet, es gibt eine gerichtete Kante vom Knoten A zum Knoten B. Die Funktion `path` bestimmt, ob ein gerichteter Pfad zwischen zwei Knoten x und y existiert oder nicht. Ein solcher Pfad existiert genau dann, wenn es entweder eine direkte Kantenverbindung zwischen den beiden Knoten gibt oder wenn es eine Kante von x zu einem weiteren Knoten z gibt und gleichzeitig ein Pfad von z nach y existiert.

Wie man sieht, kann man mit Hilfe der Constraints und unter Verwendung einer freien Hilfsvariablen diese natürlichsprachliche Definition für einen Pfad fast eins zu eins im Code umsetzen: Falls es eine direkte Verbindung zwischen x und y gibt, so wird diese über das Constraint `edge x =:= y` gefunden. Ansonsten werden durch Belegung der freien Variablen z im Constraint `edge x =:= z` alle von x über eine Kantenverbindung aus erreichbaren Knoten bestimmt und es wird überprüft, ob es von dort einen Pfad zum Knoten y gibt (`path z y`).

Bei der Verwendung von freien Variablen ist zu beachten, dass diese explizit als frei deklariert werden müssen, zum Beispiel durch `let x free in ...` oder `... where x free`.

Es gibt jetzt mehrere Möglichkeiten, Anfragen an die `path`-Funktion zu stellen: Zum einen sind normale Aufrufe der Form `path A C` möglich, die nur dann `Success` zurückliefern, wenn die entsprechende Kantenverbindung im Graph existiert. Zum anderen können aber auch ein oder zwei konkrete Argumente der Funktion durch freie Variablen ersetzt werden. So lassen sich zum Beispiel alle vom Knoten A ausgehenden Pfade mittels `path A x where x free` ermitteln. Diese Anfrage liefert dann für die folgenden Bindungen von `x` einen `Success`: `{x=B,x=E,x=D,x=C}`. Nach dem gleichen Prinzip kann man sich mittels `path x E where x free` auch alle Startknoten für Pfade, die im Graphen zum Knoten E führen, ausgeben lassen.

2.1.5. Constraint-Programmierung

Das Programmieren mit Constraints (auch Constraint-Programming) ist ebenfalls Teil der deklarativen Programmierung. Es befasst sich im Allgemeinen mit dem Lösen von Problemen durch Beschreibung von Eigenschaften oder Bedingungen (Constraints), die für mögliche Lösungen eines Problems gelten müssen. Modelliert werden diese Eigenschaften durch die Definition von Regeln und Beziehungen für sogenannte Constraint-Variablen. Diese Variablen sind über einem endlichen oder unendlichen Wertebereich definiert. Die Modellierung eines Problems über einem endlichen Wertebereich wird auch als Finite-Domain-Constraint-Programming bezeichnet.

Aufgrund seines deklarativen Stils und der Unterstützung der Programmierung mit freien Variablen ist Curry gut für die Einbettung einer Constraint-Modellierungssprache geeignet. Dies soll nun an einem Anwendungsbeispiel für Finite-Domain-Constraints verdeutlicht werden, dem sogenannten N-Damen-Problem:
Bei diesem Problem ist es das Ziel, N-Damen auf einem NxN-Schachbrett so zu platzieren, dass keine Dame eine andere Dame schlagen kann. Hierbei gelten die vom Schach bekannten Züge für eine Dame. Es darf also keine Dame mit einer anderen Dame in derselben Spalte, Reihe oder Diagonalen des Schachbretts stehen.
Mit Curry lässt sich das N-Damen-Problem nun folgendermaßen modellieren (betrachtet wird hier die Modellierung mit der CLPFD-Bibliothek der PAKCS-Curry-Implementierung, die Curry-Programme nach PROLOG übersetzt):

```
queens options n l =
       gen_vars n =:= l &
       domain l 1 (length l) &
       all_safe l &
       labeling options l

all_safe [] = success
all_safe (q:qs) = safe q qs 1 & all_safe qs

safe :: Int -> [Int] -> Int -> Success
safe _ [] _ = success
safe q (q1:qs) p = no_attack q q1 p & safe q qs (p+#1)

no_attack q1 q2 p = q1 /=# q2 & q1 /=# q2+#p & q1 /=# q2-#p

gen_vars n = if n==0 then [] else var : gen_vars (n-1)
 where var free
```

Es wird eine Funktion `queens` definiert, die drei Argumente erhält: spezielle Optionen für das Labeling der Variablen (`options`), die Problemgröße (`n`) sowie die Liste der Constraint-Variablen

(`l`). Ein Aufruf dieser Funktion zur Berechnung des 4-Damen-Problems könnte dann beispielsweise so aussehen: `queens [] 4 l where l free`.
Die Funktion `gen_vars` erzeugt eine Liste von `n` freien Variablen, die per Unifikation an `l` gebunden wird. Mit Hilfe dieser `n` Variablen wird die Tatsache modelliert, dass jede Dame in einer anderen Spalte des Schachbretts positioniert werden soll. Eine solche Variable gibt zusammen mit ihrem Wert genau ein Feld auf dem Schachbrett an. Die Variable selbst bestimmt die Spalte, ihr Wert die Reihe (vergleiche Zeichnung für 4-Damen-Problem). Das heißt `q1 = 2` bedeutet, platziere eine Dame in der 1. Spalte des Schachbretts und zwar auf dem Feld in der 2. Reihe.

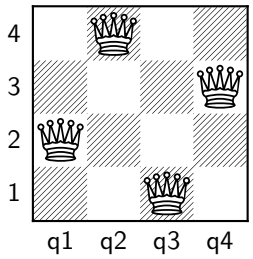


Abbildung 1: Lösung für das 4-Damen-Problem

Bei der Modellierung des Problems wird zunächst per `domain`-Constraint der Wertebereich der `n` Variablen auf $1, ..., n$ festgelegt. Mit den selbst definierten Hilfs-Constraints `all_safe`, `safe` und `no_attack` werden alle weiteren erforderlichen Beschränkungen für die Constraint-Variablen generiert: So erzeugt die Funktion `no_attack` für zwei Variablen alle Constraints, die nötig sind, um sicherzustellen, dass sich die durch sie repräsentierten Damen nicht gegenseitig schlagen können (`q1 /=# q2` $\Rightarrow$ Damen stehen nicht in der gleichen Reihe, `q1 /=# q2 +# p & q1 /=# q2 -# p` $\Rightarrow$ Damen stehen nicht auf der gleichen aufsteigenden bzw. abfallenden Diagonalen). `safe` überprüft durch Aufruf von `no_attack`, dass eine übergebene Damen-Position `q` sicher ist, also von keiner der anderen Damen angegriffen werden kann und `all_safe` wendet dieses `safe`-Constraint schließlich auf alle Constraint-Variablen an.
Abschließend wird durch Aufruf von `labeling` das Labeling der Constraint-Variablen angestoßen. Diese Technik, bei der verschiedene, gemäß Wertebereich gültige Variablenbelegungen durch einen Constraint-Solver ausprobiert werden, wird im Bereich der Finite-Domain-Constraint-Programmierung eingesetzt, um Lösungen zu finden.

Allgemein lässt sich ein FD-Constraint-Problem in Curry (PAKCS) also wie folgt angeben:

1. Festlegen des Wertebereichs der Constraint-Variablen mit dem `domain`-Constraint,
2. Modellieren des Problems durch Angabe weiterer Constraints (wie `(=#)`, `(/=#)`, `(<#)` etc.) über den Constraint-Variablen,
3. Anstoßen des Labelings der Constraint-Variablen mit `labeling`.

2.1.6. Operationelle Semantik

Currys operationelle Semantik kombiniert die aus der funktionalen Sprache Haskell bekannte bedarfsgesteuerte Auswertung (auch lazy evaluation) mit der Möglichkeit freie Variablen in Ausdrücken zu instanziieren. Das bedeutet, enthält ein auszuwertender Curry-Ausdruck keine freien Variablen, so wird er mittels **lazy evaluation** ausgewertet. Kommen hingegen freie Variablen vor, so ist es möglich, dass sich der Ausdruck für unterschiedliche Variablenbindungen zu verschiedenen Ergebniswerten reduzieren lässt. Bei der Verwendung von freien Variablen besteht ein gelöster sogenannter Antwortausdruck somit aus dem ausgewerteten Ausdruck selbst sowie der Belegung der freien Variablen, die zu dieser Auswertung geführt haben (Substitution).
Curry verwendet zwei unterschiedliche Techniken zur Instanziierung von freien Variablen während der Auswertung:

1. **Residuation:** Verzögert die Auswertung eines (Teil-)Ausdrucks mit freien Variablen, bis die freie Variable (z.B durch Auswertung anderer Teilausdrücke) an einen Wert gebunden wurde.

2. **Narrowing:** Falls der Wert einer freien Variable von den linken Regelseiten einer Funktion *benötigt* wird, so wird diese freie Variable nicht-deterministisch mit den erforderlichen Werten instanziiert und dann die Auswertung mit der passenden Regel durch Anwendung von Reduktionsschritten fortgesetzt.
   Der Wert einer freien Variable wird unter anderem bei einem Funktionsaufruf *benötigt*, falls auf der linken Seite einer definierenden Regel dieser Funktion ein Konstruktor an der gleichen Position steht wie die freie Variable im Aufruf.

2.2. KiCS2

Dieser Abschnitt befasst sich mit der KiCS2-Curry-Implementierung, die im Rahmen dieser Arbeit um eine Schnittstelle zur Integration von Constraint-Solvern erweitert werden soll.
KiCS2 (Kiel Curry System Version 2) ist ein Compiler für die funktional-logische Sprache Curry, der derzeit am Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion der Christian-Albrechts-Universität zu Kiel von Michael Hanus und seinen Mitarbeitern entwickelt wird.
Diese Arbeit beschränkt sich bei der Vorstellung von KiCS2 im Wesentlichen auf die Konzepte, die für die Entwicklung der Constraint-Solver-Erweiterung von Bedeutung sind.
Auf der KiCS2-Homepage des Lehrstuhls (http://www-ps.informatik.uni-kiel.de/kics2) kann man jedoch mehrere Artikel ([2], [1]) mit weiterführenden Informationen finden, sowie die aktuelle KiCS2-Distribution herunterladen.

KiCS2 kompiliert Curry-Programme in rein funktionale Haskell-Programme und nutzt den Glasgow Haskell Compiler als Back-End. Einerseits ermöglicht die Übersetzung nach Haskell die Wiederverwendung der Implementierung funktionaler Features wie bedarfsgesteuerte Auswertung (lazy evaluation) oder Funktionen höherer Ordnung. Auf der anderen Seite hat man das Problem, dass Haskell keinen Nicht-Determinismus unterstützt, während in Curry nicht-deterministische Ergebnisse in jedem Berechnungsschritt vorkommen können. Daher müssen nicht-deterministische Berechnungen in Haskell implementiert werden.

2.2.1. Darstellung des Nicht-Determinismus

Der KiCS2-Compiler löst dieses Problem, indem nicht-deterministische Werte explizit in Haskell-Datenstrukturen dargestellt werden. Dazu wird jeder Datentyp in Haskell um einen zusätzlichen Konstruktor `Choice` erweitert, der die Auswahl zwischen zwei möglichen Werten repräsentiert.
Funktional-logische Berechnungen können auch fehlschlagen. Im Unterschied zu rein funktionalen Programmen soll dann die Berechnung nicht abgebrochen werden, sondern ein Fehlschlag soll wie ein Ergebnisfall ohne sinnvolle Ergebnisse behandelt werden. Um diesen Fall auch in Haskell abbilden zu können, wird für jeden Datentyp zusätzlich der Konstruktor `Fail` eingeführt:

```
data Bool = True
          | False
          | Choice Bool Bool
          | Fail
```

Listing 5: Explizite Darstellung des Nicht-Determinismus in KiCS2

Der nicht-deterministische Curry-Ausdruck `aCurryBool = True ? False`, der die beiden Ergebnisse `True` und `False` liefert, lässt sich mit dieser Erweiterung direkt in Haskell darstellen und zwar durch `aHaskellBool = Choice True False`.

Wie weiter oben erwähnt, können nicht-deterministische Werte in Curry in jedem Berechnungsschritt vorkommen und somit auch als Funktionsargumente auftreten. Daher muss bei Funktionsdefinitionen über Pattern-Matching eine zusätzlich Regel für das Vorkommen von `Choice`-Argumenten eingeführt werden. Wie man im Beispiel sieht, wird dabei der Nicht-Determinismus von der Argument- auf die Ergebnisebene propagiert und die Funktion auf die beiden Möglichkeiten in der `Choice` angewandt. Die Möglichkeit des Fehlschlagens der Funktionsanwendung wird durch die letzte, immer passende Regel abgebildet, die ein `Fail` liefert (etwaige `Fail`-Werte auf Argumentebene werden mit dieser Regel ebenfalls propagiert):

```
not :: Bool -> Bool
not False          = True
not True           = False
not (Choice b1 b2) = Choice (not b1) (not b2)
not _              = Fail
```

Listing 6: Propagieren des Nicht-Determinismus

Die bisherige Implementierung des Nicht-Determinismus in Haskell hat allerdings noch einige Schwächen: Verwendet man nicht-deterministische Werte als Argumente, so kann dies zu einer semantischen Mehrdeutigkeit führen. So liefert der Ausdruck `xorSelf aHaskellBool` mit der unten angegebenen Implementierung von `xorSelf` beispielsweise das folgende Ergebnis:
`Choice (Choice False True) (Choice True False)`.
Das bedeutet, der Ausdruck liefert die Ergebnisse `False`, `True`, `True` und `False` und damit auch das nicht-gewollte Resultat `True`.

```
xorSelf :: Bool -> Bool
xorSelf x = xor x x

xor True x           = xor' x
xor False x          = x
xor (Choice x1 x2) x = Choice (xor x1 x) (xor x2 x)
xor _                = Fail

xor' False          = True
xor' True           = False
xor' (Choice x1 x2) = Choice (xor' x1) (xor' x2)
xor' _              = Fail
```

Bei einer strikten Auswertung der Argumente kommt das ungewünschte Ergebnis `True` hingegen nicht vor. Um zu verhindern, dass das Ergebnis von der gewählten Auswertungsstrategie abhängt, implementiert der KiCS2-Compiler die sogenannte **call-time-choice-Semantik**. Diese Semantik besagt, dass die Argumentwerte einer Funktion bestimmt werden müssen, bevor mit der Funktionsauswertung begonnen wird. Dabei können die Argumente trotzdem lazy ausgewertet werden.

Kommt ein nicht-deterministischer Wert mehrfach vor, so sorgt **Haskells Sharing** dafür, dass dieser bei jedem Vorkommen zum gleichen Wert reduziert wird.
Übertragen auf die KiCS2-Implementierung bedeutet dies, dass man eine Möglichkeit finden muss, das mehrfache Auftreten einer `Choice` zu erkennen, um eine konsistente Wertauswahl bei jedem Vorkommen sicherzustellen. In dem obigen Beispiel wird die einzelne `Choice` aus `aHaskellBool` mehrfach kopiert, so dass die gleiche `Choice` letztendlich dreimal im Ergebnisausdruck `Choice (Choice False True) (Choice True False)` vorkommt. Um dies auch bei der Auswertung des nicht-deterministischen Ergebnisses zu erkennen, wird jede `Choice` mit einem eindeutigen Identifier gekennzeichnet.

```
type ID = Integer

data Bool = ...
          | Choice ID Bool Bool
```

Listing 7: Einführung eines Identifikators für `Choice`s

Die Regeln für nicht-deterministische Argumente in Funktionsdefinitionen werden derart angepasst, dass beim Propagieren der `Choice`s der Identifier beibehalten wird:

```
xor (Choice i x1 x2) x = Choice i (xor x1 x) (xor x2 x)
```

Nun liefert der Ausdruck `xorSelf aHaskellBool` das folgende Ergebnis:
`Choice 1 (Choice 1 False True) (Choice 1 True False)` (die `ID` für `aHaskellBool` sei 1).
Somit ist sichergestellt, dass bei konsistenter Auswahl eines Wertes in allen Vorkommen einer `Choice` (also beispielsweise immer Auswahl des linken Werts) tatsächlich nur noch der erwartete Ergebniswert `False` herauskommen kann.

Curry als funktional-logische Sprache bietet auch die Möglichkeit, freie Variablen in Ausdrücken und Programmen zu verwenden. Diese freien Variablen ersetzt KiCS2 durch nicht-deterministische Generatoren in Haskell. Ein solcher Generator erzeugt lazy alle möglichen Werte des jeweiligen Variablentyps. Bereitgestellt wird dieser Generator von der Typklasse `Generable`, die im Folgenden zusammen mit einer (leicht vereinfachten) Beispielinstanz für `Bool` aufgeführt wird. In der Typklasse `NonDet` werden alle nicht-deterministischen Typen zusammengefasst, das heißt, alle Typen, die die Erweiterung um die `Choice`- bzw. `Fail`-Konstruktoren unterstützen:

```
class NonDet a => Generable a where
  generate :: IDSupply -> a

instance Generable Bool where
  generate s = Choice (thisID s) True False
```

Listing 8: Realisierung von freien Variablen in KiCS2

Um zu gewährleisten, dass während der Berechnung neue, bislang nicht verwendete `ID`s zur Kennzeichnung von `Choice`s generiert werden können, erhalten die Generatoren und andere nicht-deterministische Funktionen der KiCS2-Implementierung als zusätzliches Argument einen sogenannten `IDSupply`. Dieser stellt - konzeptuell betrachtet - einen unendlichen Vorrat von `ID`s während der Berechnung eines Ausdrucks zur Verfügung. Neben Funktionen zur Initialisierung des `IDSupply` und zum Zugriff auf eine neue `ID` gibt es auch noch die Operationen `leftSupply` und `rightSupply` zur Erzeugung disjunkter Teilmengen von frischen `ID`s, falls auf der rechten Regelseite mehr als eine nicht-deterministische Funktion aufgerufen wird, welche ihren eigenen individuellen `IDSupply` benötigt.

```
type IDSupply = Integer

initSupply :: IO IDSupply
initSupply = return 1

thisID :: IDSupply -> ID
thisID n = n

leftSupply, rightSupply :: IDSupply -> IDSupply
leftSupply n = 2 * n
rightSupply n = 2 * n + 1
```

Listing 9: Einführung eines `IDSupply` zur Erzeugung neue Identifikatoren

Neben der Realisierung mit unbeschränkten Integern gibt es weitere `IDSupply`-Implementierungen (z.B. mit Haskells Data.IORef- oder Unique-Modul). Diese verschiedenen Realisierungen können ausgetauscht werden, um für die jeweilige Anwendung die günstigste auszuwählen.
Zur Optimierung führt der KiCS2-Compiler vor der Übersetzung eine Determinismus-Analyse durch: Generiert die zu übersetzende Curry-Funktion keine nicht-deterministischen Werte - weder direkt noch indirekt, so kann auf das zusätzliche `IDSupply`-Argument verzichtet werden und der für sie erzeugte Code entspricht in etwa dem funktionalen Pendant dieser Funktion.

2.2.2. Unifikation

In logischen Sprachen werden Prädikate oder Constraints dazu eingesetzt, den Ergebnisraum nicht-deterministischer Berechnungen zu beschränken. Auch in Curry gibt es derartige Constraints wie beispielsweise `(=:=) :: a -> a -> Success`. Semantisch entspricht dieses Gleichheits-Constraint der strikten Gleichheit auf Termebene, das heißt, dass es nur erfüllt ist, wenn seine beiden Argumente zu unifizierbaren Konstruktortermen reduzierbar sind.
Bei der Übersetzung des Gleichheits-Constraints durch KiCS2 könnte man nun genauso vorgehen wie bei den bisherigen Funktionen und neue Regeln definieren, die auf `Choice`- und `Fail`-Argumente matchen.

```
[]           =:= []           = Success
(x:xs)       =:= (y:ys)       = x =:= y & xs =:= ys
Choice i l r =:= y            = Choice i (l =:= y) (r =:= y)
x            =:= Choice j l r = Choice j (x =:= l) (x =:= r)
...
```

Listing 10: Erster Ansatz zur Implementierung der Unifikation in KiCS2

Diese Implementierung ist zwar korrekt, sie kann aber bei Unifikationen mit freien Variablen als Argument zu unnötig großen Ergebnisräumen führen. Betrachtet man zum Beispiel den Curry-Ausdruck `x =:= [True] where x free`, so wird die freie Variable `x` in KiCS2 durch den folgenden nicht-deterministischen Generator für Listen über `Bool` repräsentiert:

```
boolListVar :: IDSupply -> [Bool]
boolListVar s = generate s
boolVar :: IDSupply -> Bool
boolVar s = generate s

instance Generable [Bool] where
  generate s = Choice (thisID s) []
    (boolVar (leftSupply s) : boolListVar (rightSupply s))
```

Dieser Generator baut nun unnötigerweise einen Suchraum für die Belegung der freien Variable `x` auf, obwohl keine Suche erforderlich ist, da das Gleichheits-Constraint nur für die Belegung mit `[True]` erfüllt ist. Für die Unifikation von zwei freien Variablen führt diese Implementierung gar zum Aufbau unendlicher Suchräume.
Daher wird die Unifikation bei der Übersetzung durch den KiCS2-Compiler anders behandelt als die übrigen Funktionen: Anstatt bei einer Unifikation über einer freien Variable einen Suchraum durch Aufzählung aller möglichen Belegungen aufzubauen, wird diese freie Variable an einen Term bzw. eine andere freie Variable **gebunden**. Da Haskell als rein funktionale Sprache seiteneffektfrei ist, werden diese Bindungen mit Hilfe spezieller Bindungs-Constraints ausgedrückt.
Zunächst ist es allerdings nötig, die Repräsentation von freien Variablen und Standard-`Choice`s in KiCS2 unterscheiden zu können. Dazu werden die Definition des Identifiertyps und die der Generatoren wie folgt angepasst:

```
data ID = ChoiceID IDSupply | FreeID IDSupply

instance Generable Bool where
  generate s = Choice (FreeID (thisID s)) True False
```

Listing 11: Unterscheidung von freien Variablen und Standard-`Choice`s

Falls der Wert einer freien Variable für die Anwendung einer Funktion benötigt wird, so wird die Darstellung der freien Variable in eine Standard-`Choice` umgewandelt, das heißt, die Regel für `Choice`-Argumente wird wie folgt angepasst:

```
narrow :: ID -> ID
narrow (FreeID i) = ChoiceID i
narrow i          = i

not (Choice i b1 b2) = Choice (narrow i) (not b1) (not b2)
```

Listing 12: Narrowing von freien Variablen bei Funktionsanwendung

Die oben erwähnten Bindungs-Constraints werden nun durch den folgenden Datentyp dargestellt:

```
data Constraint = ID :=: Decision

data Decision = ChooseLeft | ChooseRight | ChooseN Int Int |
                BindTo ID  | NoDecision
```

Listing 13: Definition von Bindungs-Constraints und -entscheidungen

Ein Bindungs-Constraint ist also ein Paar bestehend aus einer Variablen-ID und der Bindungsentscheidung für diese freie Variable. Eine freie Variable kann an den linken bzw. rechten Pfad einer `Choice` oder mittels `ChooseN` direkt an einen bestimmten Konstruktor gebunden werden. Daben geben die `Int-Argumente` der `ChooseN`-Entscheidung den Index des Konstruktors sowie die Anzahl der von ihm erwarteten Argumente an.
Mit `BindTo` kann man eine freie Variable an eine andere binden. Und `NoDecision` bildet schließlich den Fall ab, dass für die Variable bislang noch keine Bindungsentscheidung getroffen wurde.
Damit man Curry-Ausdrücke nach der Übersetzung mit solchen Bindungs-Constraints beschränken kann, wird der Repräsentation der Curry-Typen in Haskell ein weiterer Konstruktor hinzugefügt:

```
data Bool = ...
          | Guard Constraints Bool

data Constraints = StructConstr [Constraint]
```

Listing 14: Einführung von `Guard`-Ausdrücken

Ein `Guard`-Ausdruck `Guard cs e` repräsentiert einen durch Constraints beschränkten Wert und ist wie folgt zu interpretieren: Der Wert `e` ist nur dann gültig, wenn die Constraints in `cs` erfüllbar sind. Dabei liefert ein einzelnes `Constraint` immer nur die Bindungsentscheidung für einen Konstruktor. Um auch strukturierte Datentypen mit Konstruktoren mit mehreren Argumenten zu

unterstützen, erhalten `Guard`-Ausdrücke jedoch immer eine Liste von Constraints (vergleiche Datentyp `Constraints`). Diese Constraint-Liste enthält dann je ein Bindungs-Constraint für den äußersten Konstruktor sowie für dessen sämtliche Argumente.

Mit den neu eingeführten Bindungs-Constraints lässt sich nun die Implementierung des Unifikationsoperators `(=:=)` anpassen. Während die Implementierung für Standard-`Choice`s erhalten bleibt, werden für freie Variable neue Regeln definiert. Als Ergebnis liefern diese neuen Regeln einen `Guard`-Ausdruck, der - falls die Constraints erfüllbar sind - `Success` zurückliefert.
Welche Bindungs-Constraints die `Guard` enthält, ist davon abhängig, ob die freie Variable an einen Konstruktorterm oder eine andere freie Variable gebunden wird. Im Falle eines Konstruktors wird die gleiche Bindungsentscheidung getroffen wie die, die im erfolgreichen Pfad des zur freien Variable zugehörigen Generators getroffen werden würde. Bindet man die freie Variable hingegen an eine andere freie Variable, so wird - wie auch in der unten aufgeführten Beispielimplementierung für boolesche Variablen - eine `BindTo`-Bindungsentscheidung verwendet.

```
Choice (FreeID i) _ _ =:= True
  = Guard (StructConstr [i :=: ChooseLeft]) Success
Choice (FreeID i) _ _ =:= False
  = Guard (StructConstr [i :=: ChooseRight]) Success
Choice (FreeID i) _ _ =:= Choice (FreeID j) _ _
  = Guard (StructConstr [i :=: BindTo j]) Success
```

Listing 15: Verbesserte Implementierung der Unifikation

2.2.3. Auswertung von (nicht-deterministischen) Ausdrücken

Abschließend soll nun noch die Auswertung der durch KiCS2 erzeugten Haskell-Ausdrücke betrachtet werden. Bei der Berechnung der Normalform eines Ausdrucks in KiCS2 wird jeglicher Nicht-Determinismus - dargestellt durch die `Choice`s - hochpropagiert. Das heißt, ein normalisierter Ausdruck entspricht entweder einem deterministischen Wert, der direkt ausgegeben werden kann, oder einem Suchbaum mit `Choice`s als innere Knoten und deterministischen Werten bzw. fehlgeschlagenen Berechnungen (`Fail`) als Baumblätter.
Um alle in einem solchen Suchbaum enthaltenen Ergebnisse zu bestimmen, stellt KiCS2 im Unterschied zu anderen Curry-Implementierungen eine Vielzahl unterschiedlicher Suchstrategien zur Verfügung. Unter anderem ist es möglich, den Suchbaum mittels Tiefen-, Breiten- oder paralleler Suche auszuwerten. Des Weiteren können alle Werte in einer Baum-ähnlichen Struktur gesammelt werden und mit Hilfe dieser Struktur eigene Auswertungsstrategien implementiert werden (= **eingekapselte Suche**).

Im Folgenden soll beispielhaft die Implementierung der Tiefensuche betrachtet werden.
Der Einfachheit halber wurden die durch KiCS2 neu eingeführten Konstruktoren bislang unabhängig von ihrem jeweiligen Typ stets mit `Choice`, `Fail` bzw. `Guard` bezeichnet. Die tatsächlichen

Konstruktorbezeichnungen enthalten zusätzlich einen Hinweis auf ihren jeweiligen Curry-Typ. Somit sieht beispielsweise die Haskell-Repräsentation des Curry-Typs `Bool` folgendermaßen aus:

```
data C_Bool = C_False
            | C_True
            | Choice_C_Bool ID C_Bool C_Bool
            | Fail_C_Bool
            | Guard_C_Bool Constraints C_Bool
```

Listing 16: Beispiel: Repräsentation eines Bools in KiCS2

Um nun bei der Implementierung der Suchstrategien das Pattern-Matching auf die unterschiedlichen Fälle möglichst einfach zu halten, führt man den folgenden generischen Datentyp ein:

```
data Try a = Val a | Choice ID a a | Fail | Guard Constraints a
```

Listing 17: Generische `Try`-Struktur für das Pattern Matching

Zusätzlich definiert man eine Funktion `try :: a -> Try a`. Hierbei handelt es sich um eine überladene Funktion; das heißt, sie wird von einer speziellen Typklasse zur Verfügung gestellt und für jede Haskelldarstellung eines Curry-Typs (wie `C_Bool`), die in den generischen `Try`-Datentyp konvertierbar ist, gibt es eine entsprechende Typklasseninstanz.

```
try (Choice_C_Bool i l r) = Choice i l r
try Fail_C_Bool           = Fail
try (Guard_C_Bool cs e)   = Guard cs e
try v                     = Val v
```

Listing 18: Transformation eines Bool in die generische `Try`-Struktur

Unter Verwendung dieser Hilfsfunktion lässt sich die Tiefensuche von KiCS2 dann definieren (siehe weiter unten): Die Fälle, in denen ein deterministischer Wert bzw. eine fehlgeschlagene Berechnung vorliegen, sind trivial. Im ersten Fall wird der Wert einfach ausgegeben, im Zweiten die Suche abgebrochen.
Um die Werte für Standard-`Choice`s und freie Variablen auszugeben, muss man wissen, welche (Bindungs-)Entscheidung für diese `Choice`s getroffen wurde. KiCS2 speichert bereits getroffene Entscheidungen unter der `ID` einer `Choice` in einem **globalen Decision-Store** ab. Realisiert wurde der Store mit Hilfe von Haskells `IO` Monade. Durch Implementierung der Typklasse `class Monad m => Store m` können aber auch andere Realisierungen definiert werden.
Über die beiden Funktionen `lookupDecision` und `setDecision` kann man auf gespeicherte Entscheidungen zugreifen bzw. neue Entscheidungen für eine `Choice` unter deren `ID` eintragen.

```
lookupDecision :: Store m => ID -> m Decision
setDecision :: Store m => ID -> Decision -> m ()
```

Listing 19: Funktionen zum Zugriff auf den globalen Decision-Store

Gelangt man bei der Auswertung des Suchbaums mit der Tiefensuche zu einem Knoten mit einer `Choice`, so wird überprüft, ob für sie schon eine Entscheidung im Decision-Store abgelegt wurde. Falls ja, wird die Auswertung im entsprechenden Zweig der `Choice` fortgesetzt.
Wurde hingegen noch keine Entscheidung getroffen, so werden nacheinander der linke und der rechte Zweig ausgewertet. Zuvor wird jeweils die getroffene Bindungsentscheidung in den Store eingetragen, um beim mehrfachen Vorkommen der gleichen `Choice` in demselben Auswertungszweig zu gewährleisten, dass auch wirklich überall die gleiche Entscheidung verfolgt wird. Abschließend wird die Entscheidung im Store wieder auf `NoDecision` zurückgesetzt, welches auch der initiale Wert für alle `Choice`s zu Beginn der Auswertung ist.
Um einen durch Constraints beschränkten Ausdruck, also einen `Guard`-Ausdruck, auszuwerten, müssen die Constraints zunächst gelöst und damit ihre Erfüllbarkeit überprüft werden. Hierfür implementiert KiCS2 einen Constraint-Solver, der - wenn möglich - die Bindungs-Constraints löst und gegebenenfalls getroffene Bindungsentscheidungen in den Decision-Store einträgt. Aufgerufen wird er über die Funktion `solve`. Konnten die Constraints gelöst werden, so wird die Auswertung fortgesetzt, wobei die getroffenen Entscheidungen am Ende zurückgesetzt werden.

```
dfs :: Try a -> IO ()
dfs (Val v) = print v
dfs Fail    = return ()
dfs (Choice i l r) = lookupDecision i >>= follow
  where follow ChooseLeft = dfs (try l)
        follow ChooseRight = dfs (try r)
        follow NoDecision = do setDecision i ChooseLeft
                               dfs (try l)
                               setDecision i ChooseRight
                               dfs (try r)
                               setDecision i NoDecision
dfs (Guard cs e) = solve cs (dfs (try e))
```

Listing 20: Auswertung eines nicht-deterministischen Ausdrucks mit der Tiefensuche

2.3. Monadic-Constraint-Programming-Framework

Das Monadic-Constraint-Programming-Framework (kurz: MCP) - entwickelt von Tom Schrijvers, Peter Stuckey, Phil Wadler und Pieter Wuille - ist ein Framework zur Modellierung und Lösung von Constraint-Problemen in Haskell. Diese Arbeit stützt sich in Bezug auf dieses Framework in erster Linie auf die Artikel [9] und [10].
Dieses Kapitel fasst die obigen Artikel grob zusammen und liefert somit einen knappen Überblick über die wichtigsten Features des MCP-Frameworks sowie über die im Rahmen dieser Arbeit verwendeten Schnittstellen. Im Zuge der Weiterentwicklung des MCP-Frameworks sind allerdings noch weitere Artikel erschienen: [11] und [12].
Das MCP-Framework stellt ein in Haskell implementiertes, generisches Constraint-Programming-System zur Verfügung. Für seine Realisierung wurde auf eine Vielzahl funktionaler Abstraktionen wie Monaden, Funktionen höherer Ordnung oder lazy evaluation zurückgegriffen.
Zu den vom Framework bereitgestellten Features zählen unter anderem:

- eine in Haskell eingebettete Sprache zur Modellierung von Constraint-Problemen (embedded domain specific language),
- zwei Constraint-Solver-Backends:
  - ein direkt in Haskell realisierter Solver
  - sowie ein Anschluss der C++-Solver-Bibliothek **Gecode**,
- eine Vielzahl von Auswertungsstrategien durch
  - Einsatz bekannter Suchalgorithmen wie Tiefen- und Breitensuche einerseits
  - sowie der Möglichkeit zur Definition komplexerer Suchstrategien mittels spezieller kombinierbarer Such-Transformer andererseits.

2.3.1. Allgemeines

Repräsentation von Constraint-Modellen als Bäume:
Constraint-Modelle werden im MCP-Framework mit Hilfe einer baumartigen Datenstruktur repräsentiert. Um Constraint-Modelle für beliebige Constraint-Solver und Ergebnisse darstellen zu können, wird der Baumdatentyp mit zwei Typvariablen - `solver` für den verwendeten Solver und `a` für den Ergebnistyp - parametrisiert. Durch die Bindung des Baummodells an einen konkreten Solver werden auch die Constraints (Typ `Constraint solver`) und Constraint-Terme (Typ `Term solver`) festgelegt, denn diese beiden Typen werden bei der Implementierung eines Solvers definiert (Vergleiche: Typklasse `Solver` unten).
Die Baumstruktur enthält verschiedene Knoten mit speziellen Anweisungen für die Auswertung des Modells durch den Solver. So gibt es beispielsweise Knoten zur Erzeugung einer neuen Constraint-Variable (`NewVar`) oder zum Hinzufügen eines Constraints zum Constraint-Speicher des Solvers

(`Add`). Um auch Disjunktionen wie zum Beispiel die Belegung einer Constraint-Variablen mit unterschiedlichen Werten darstellen zu können, gibt es den `Try`-Konstruktor, mit dem man Verzweigungen im Baum einführen kann. Ein Pfad im Baum kann entweder zu einer Lösung führen - repräsentiert durch `Return a` - oder eben nicht, was mit einem `Fail`-Blattknoten dargestellt wird.

```
data Tree solver a = Return a
                   | NewVar (Term solver -> Tree solver a)
                   | Add (Constraint solver) (Tree solver a)
                   | Try (Tree solver a) (Tree solver a)
                   | Fail
```

Listing 21: Repräsentation von Constraint-Modellen

Der Typkonstruktor `Tree solver` wird zu einer Instanz von Haskells `Monad`-Typklasse gemacht:

```
instance Monad (Tree solver) where
  return = Return
  (>>=) = extendTree

extendTree :: Tree solver a -> (a -> Tree solver b)
           -> Tree solver b
(Return x) `extendTree` k = k x
(NewVar f) `extendTree` k = NewVar (\v -> f v `extendTree` k)
(Add c t)  `extendTree` k = Add c (t `extendTree` k)
(Try l r)  `extendTree` k = Try (l `extendTree` k)
                                (r `extendTree` k)
Fail       `extendTree` k = Fail
```

Listing 22: Erweiterung monadischer Baummodelle

Durch den Aufruf von `return` wird ein triviales Baummodell erzeugt, das den übergebenen Wert als Lösung einkapselt. Der monadische Bindeoperator `(>>=)` wird durch eine Funktion `extendTree` implementiert, die ein Baummodell erweitert. Diese Erweiterung wird durch eine Funktion vollzogen, die angewandt auf den im Baummodell eingekapselten Wert ein neues Modell liefert. Diese Funktion wird bis zu den Blättern des Baummodells durchgereicht und auf den dortigen Wert angewandt (`Return`-Knoten) oder sie wird verworfen (`Fail`-Knoten).
Die Verwendung von monadischen Modellen hat den Vorteil, dass bei der Berechnung eines Werts als Seiteneffekt ein Modell erzeugt wird. Außerdem ermöglicht dies die Wiederverwendung einiger vordefinierter Funktionen für Monaden, wie die folgenden Funktionen zeigen, die als syntaktischer Zucker für die Modellierung vom Framework bereitgestellt werden:

```
(/\) = (>>)
(\/) = Try
conj = sequence
true = return ()
false = Fail
addC c = Add c true
exists = NewVar
```

Listing 23: Syntaktischer Zucker für Baummodelle

Constraint-Solver-Interface:
Constraint-Solver im MCP-Framework sind im Grunde genommen nur Interpreter für die zuvor vorgestellten Baummodelle. Sie werden durch die folgenden Typklassen definiert:

```
class Monad solver => Solver solver where
  type Constraint solver :: *
  type Label solver :: *

  add :: Constraint solver -> solver Bool
  mark :: solver (Label solver)
  goto :: Label solver -> solver ()
  run ::  solver a -> a

class Solver solver => Term solver t
  newvar :: solver t
```

Listing 24: Interface für Constraint-Solver im MCP-Framework

Die Solver-Typklasse setzt voraus, dass ein MCP-Solver eine Monade ist. Eine monadischer Wert wie `solver a` ist eine Abstraktion für eine (seiteneffektbehaftete) Berechnung `solver`, die ein Ergebnis vom Typ `a` zurückliefert. Ein Constraint-Solver hat üblicherweise einen internen Zustand, den Constraint-Speicher, welcher durch die monadische Realisierung vor dem Benutzer verborgen wird.
Bei der Instanziierung der Solver-Typklasse müssen unter anderem zwei Typen definiert werden:

- `Constraint solver`: Typ der Constraints, die dieser konkrete Solver interpretieren und lösen kann,
- `Label solver`: Label-Typ (Zustandsmarke), mit dem der interne Zustand des Solvers repräsentiert wird.

Neben diesen Typen, müssen noch die folgenden Funktionen implementiert werden:

- `add`: Hinzufügen eines Constraints zum Constraint-Speicher des Solvers,
- `mark`: Rückgabe des aktuellen Solver-Zustands (bzw. der aktuellen Zustandsmarke),
- `goto`: Rückversetzen des Constraint-Solvers in den zur übergebenen Zustandsmarke zugehörigen Zustand,
- `run`: Ausführen der Solver-Monade.

Jeder MCP-Solver muss außerdem über die Term-Typklasse einen Typ für Constraint-Terme implementieren. Dazu wird die Funktion `newvar` zur Erzeugung einer neuen Constraint-Variable definiert. Eine Term-Implementierung wird über ein Typklassen-Constraint einem konkreten Solver zugeordnet. Es ist möglich, mehr als einen Term-Typ für einen Solver anzugeben.

Auswertung:
Ein MCP-Solver löst ein Constraint-Problem nun durch Auswertung des zugehörigen Baummodells. Das heißt, beginnend bei der Wurzel wird jeder Knoten des Baumes betrachtet, die zum Knoten zugehörige Solver-Aktion durchgeführt und dann die Auswertung gemäß einer zuvor festgelegten primitiven Suchstrategie fortgesetzt.
Die folgende Implementierung wertet ein Baummodell gemäß der Tiefensuche aus, es wird also bei Verzweigungen im Baum stets zunächst der linke Pfad und danach der rechte Pfad ausgewertet. Dazu werden zwei Hilfsmittel eingesetzt: Zum einen eine Liste zur Speicherung des rechten Teilbaums einer Verzweigung zusammen mit dem aktuellen Solver-Zustand und zum anderen die Funktion `continue`, mit der die Auswertung in den zwischengespeicherten Teilbäumen durch Rücksetzen des Solver-Zustands fortgesetzt werden kann.
Falls das Ende eines Pfades im Baum erreicht wird, also ein `Return`- bzw. `Fail`-Knoten, so wird die Auswertung in den in der Liste zwischengespeicherten Teilbäumen fortgesetzt und - falls es sich um ein `Return`-Blattknoten handelt - die Lösung in die Ergebnisliste aufgenommen.
Bei einem `Add`-Knoten versucht der Solver, das entsprechende Constraint zu seinem Constraint-Speicher hinzuzufügen. Bleibt der Speicher auch nach dem Hinzufügen konsistent, so wird die Auswertung auf dem aktuellen Pfad fortgesetzt, sonst wird der nächste zwischengespeicherte Pfad ausgewertet.
Ein `NewVar`-Knoten enthält eine Funktion `f`, die angewandt auf eine Constraint-Variable ein neues Baummodell liefert. Daher erzeugt der Solver bei Auswertung eines solchen Knotens zunächst eine neue Constraint-Variable und führt dann die Auswertung auf dem durch die Anwendung von `f` generierten Baummodell fort.
Bei einer Verzweigung im Baum wird, wie schon erwähnt, der rechte Teilbaum zusammen mit dem aktuellen Solver-Zustand gespeichert und dann der linke Teilbaum weiter ausgewertet.

```
solve :: Solver solver => Tree solver a -> [a]
solve = run . eval

eval :: Solver solver => Tree solver a -> solver [a]
eval model = eval' model []
```

```
  where eval' (Return x) wl = do xs <- continue wl
                                 return (x:xs)
        eval' (Add c t) wl = do b <- add c
                                if b then eval' t wl
                                     else continue wl
        eval' (NewVar f) = do v <- newvar
                              eval' (f v) wl
        eval' (Try l r) = do now <- mark
                             eval' l ((now,r):wl)
        eval' Fail = continue wl

continue [] = return []
continue ((past,t):wl) = do goto past
                            eval' t wl
```

Listing 25: Auswertung von Constraint-Modellen

Neben der oben vorgestellten Implementierung der Auswertung wurden im MCP-Framework auch noch weitere primitive Suchstrategien wie die Breitensuche oder eine über heuristische Prioritäten gesteuerte Suche zur Auswertung von Baummodellen realisiert. Außerdem bietet das Framework die Möglichkeit, bei der Auswertung Transformationen auf die Baummodelle anzuwenden, mit deren Hilfe sich komplexere Suchalgorithmen umsetzen lassen. Beispiele für solche Such-Transformer sind das "Abschneiden" des Baumes ab einer bestimmten Knotentiefe (tiefenbeschränkte Suche) oder ab einer bestimmten Anzahl gefundener Lösungen (lösungsbeschränkte Suche) sowie das zufällige Vertauschen von Zweigen im Baum (zufallsgesteuerte Suche).

2.3.2. Finite-Domain-Schnittstelle

Für Finite-Domain-Constraints gibt es im MCP-Framework eine zusätzliche Abstraktionsschicht: Während im generischen Solver-Interface die verwendeten Constraints als Parameter übergeben werden, ist dies beim speziellen FD-Solver-Interface nicht nötig. Hier ist von vornherein festgelegt, dass FD-Constraints verwendet werden. Abstrahiert wird nur über den konkreten Finite-Domain-Solver und die von ihm verwendeten Techniken zur Constraint-Propagierung. Für diese verschiedenen Finite-Domain-Solver stellt das MCP-Framework eine gemeinsame Constraint-Modellierungssprache zur Verfügung, was die Entwicklung Solver-unabhängiger Modelltransformationen und -abstraktionen ermöglicht.

MCP-FD-Modellierungssprache:
Zur internen Darstellung von Finite-Domain-Constraints führt das Framework die folgenden Datentypen ein (Die vollständige Deklaration dieser Datentypen findet man im Anhang B):

- `Expr t c b`: Zur Repräsentation von arithmetischen Ausdrücken wie Addition, Subtraktion etc. über Integer-Termen,
- `ColExpr t c b`: Zur Darstellung von Listen (Collections) von Integer-Ausdrücken,
- `BoolExpr t c b`: Zur Repräsentation von FD-Constraints.

Alle drei Datentypen werden mit drei Typvariablen parametrisiert, mit denen der Typ für Integer- (`t`), Listen- (`c`) bzw. boolesche Terme (`b`) angegeben wird. Auf diese Weise können diese Datenstrukturen für unterschiedliche Term-Implementierungen wiederverwendet werden. Die folgende Term-Realisierung stellt das Framework zur Verfügung:

```
data ModelIntTerm  = ModelIntVar  Int
data ModelColTerm  = ModelColVar  Int
data ModelBoolTerm = ModelBoolVar Int
```

Listing 26: Repräsentation von FD-Variablen

Somit ist ein Term eine Constraint-Variable des entsprechenden Typs, welche durch eine `Int`-Referenz identifiziert wird. Mit den Term-Typen und den oben vorgestellen Ausdruckstypen werden dann die eigentlichen Typen zur Modellierung von Constraint-Problemen eingeführt: `ModelInt` für arithmetische Integer-Ausdrücke, `ModelCol` für Listen über Integer-Ausdrücken (auch als Collections bezeichnet) und `Model` für Finite-Domain-Constraints.

```
type ModelInt = Expr     ModelIntTerm ModelColTerm ModelBoolTerm
type ModelCol = ColExpr  ModelIntTerm ModelColTerm ModelBoolTerm
type Model    = BoolExpr ModelIntTerm ModelColTerm ModelBoolTerm
```

Listing 27: Typen für Integer- und Listen-Ausdrücke sowie FD-Constraints im MCP-Framework

Über vom Framework bereitgestellte Konstruktorfunktionen können dann FD-Constraint-Probleme modelliert werden (nur ein Auszug, vollständiges FD-Interface mit allen Konstruktorfunktionen im Anhang B):

```
(@+), (@-), (@*)  :: Expr t c b -> Expr t c b -> Expr t c b
(@=), (@/=)       :: Expr t c b -> Expr t c b -> BoolExpr t c b
xsum              :: ColExpr t c b -> Expr t c b
allDiff           :: ColExpr t c b -> BoolExpr t c b
forall, forany    :: ColExpr t c b
  -> (Expr t c b -> BoolExpr t c b) -> BoolExpr t c b
```

Listing 28: Konstruktorfunktionen für FD-Constraints und -Ausdrücke

Neben arithmetischen und relationalen Operatoren auf Integer-Ausdrücken gibt es unter anderem auch Funktionen zur Bestimmung der Summe aller Elemente einer Liste (`xsum`), zur Einschränkung von Listen auf paarweise verschiedene Elemente (`allDiff`) sowie spezielle Constraints höherer

Ordnung (`forall, forany`) zur Überprüfung der Gültigkeit eines Prädikats für alle bzw. für mindestens ein Listenelement. Bei der Konstruktion werden solche Ausdrücke durch Aufruf der `simplify`-Funktionen gegebenenfalls noch vereinfacht. Hierbei handelt es sich um eine der oben erwähnten Solver-unabhängigen Modelltransformationen. Beispiele für derartige Vereinfachungen sind die Berechnung von konstanten Teilausdrücken sowie die Normalisierung von Ausdrücken durch Vertauschung von Argumenten.

```
a @+ b      = simplify $ a `Plus` b
a @= b      = boolSimplify $ Rel a EREqual b
forall c f = boolSimplify $ BoolAll f c
```

Listing 29: Beispiele für Implementierung der Constraint-Konstruktorfunktionen

Um ein Constraint-Problem mit den vorgestellten FD-Constraints zu modellieren, muss ein entsprechendes MCP-Baummodell generiert werden. Das heißt, für jeden Constraint-Term, der durch eine der obigen Constraint-Funktionen generiert wird, muss ein `Add`-Knoten im Baummodell konstruiert, werden, der dieses Constraint einem (trivialen) Baummodell hinzufügt.
Das MCP-Framework stellt dazu in einem separaten Modul eine zweite Variante der obigen Constraint-Funktionen zur Verfügung, die direkt einen solchen `Add`-Knoten erzeugt:

```
addM :: (Constraint solver ~ Either Model q) =>
        Model -> Tree solver ()
addM m = addC $ Left m

(@=), (@<) :: (Constraint solver ~ Either Model q) =>
              ModelInt -> ModelInt -> Tree solver ()
(@=) a b = addM $ (Sugar.@=)  a b
(@<) a b = addM $ (Sugar.@<) a b

allDiff :: (Constraint solver ~ Either Model q) =>
           ModelCol -> Tree solver ()
allDiff = addM . Sugar.allDiff
...
```

Listing 30: Constraint-Konstruktorfunktionen zur direkten Erzeugung eines MCP-Baummodell-Knotens

Die Funktion `addM` konstruiert für ein gegebenes FD-Constraint vom Typ `Model` durch Aufruf von `addC` einen `Add`-Knoten, der das Constraint in einem triviales Baummodell einträgt. Die mit diesem Baummodell verknüpften Solver verwenden den Typ `Either Model q` als Constraint-Typ (ausgedrückt durch `Constraint solver ~ Either Model q`). Daher wird das FD-Constraint vor der Konstruktion des `Add`-Knotens durch Aufruf von `Left` noch in einen entsprechenden `Either`-Typ "gepackt".
Die Constraint-Funktionen, die direkt einen `Add`-Knoten zurückliefern, werden dann mit Hilfe von

`addM` und der Konstruktorfunktion für den passenden Constraint-Term aus dem Modul *Sugar.hs* implementiert.

FD-Solver-Schnittstelle:
Im Folgenden soll noch kurz auf die Einbettung von Finite-Domain-Constraint-Solvern in das MCP-Framework eingegangen werden. Da eine ausführliche Vorstellung den Rahmen dieser Arbeit sprengen würde, wird nur ein knapper Überblick über die FD-Solver-Schnittstelle des Frameworks gegeben. Weitere Informationen hierüber findet man in [10].
Neben der allgemeinen Typklasse für Solver gibt es im MCP-Framework auch eine Typklasse speziell für FD-Solver (vollständige Angabe im Anhang B):

```
class (Solver s, Term s (FDIntTerm s), Term s (FDBoolTerm s)) =>
  FDSolver s where
    type FDIntTerm s    :: *
    type FDBoolTerm s   :: *

    type FDIntSpec s    :: *
    type FDBoolSpec s   :: *
    type FDColSpec s    :: *
    ...
```

Listing 31: MCP-`FDSolver`-Interface (Ausschnitt)

Diese Typklasse macht wie auch schon die MCP-Solver-Typklasse Gebrauch von der Spracherweiterung der *TypeFamilies*. So muss man bei der Instanziierung dieser Typklasse für einen bestimmten FD-Solver die Typen für Integer-Terme und boolesche Terme angeben, die dieser FD-Solver verwendet. Des Weiteren müssen Typen für Integer-, Integer-Array- und boolesche Ausdrücke definiert werden. Die Typklasse stellt außerdem eine Reihe von Funktionen zur Implementierung bereit, darunter unter anderem Konstruktorfunktionen für die zuvor definierten Ausdruckstypen und Funktionen zum Hinzufügen von Gleichheits-Constraints.
Weiterhin definiert das Framework einen generischen Solver `FDInstance s`. Dieser Solver dient als Wrapper für konkrete FD-Solver, kapselt also konkrete FD-Solver ein.

```
newtype FDSolver s =>
  FDInstance s a = FDInstance { unFDInstance :: s a }
```

Listing 32: Wrapper für MCP-FD-Solver

Dieser Typ wird im MCP-Framework zu einer Instanz der `Solver`-Typklasse gemacht:

```
instance FDSolver s => Solver (FDInstance s) where
  type Constraint (FDInstance s) = Either Model (Constraint s)
  ...
```

Listing 33: `Solver`-Instanz für den FD-Solver-Wrapper `(FDInstance s)`(Ausschnitt)

Als Constraints legt man dabei die FD-Constraints der MCP-Modellierungssprache (`Model`) fest.

Mit Hilfe des FD-Solver-Wrappers ist es also möglich, durch die bloße Angabe einer `FDSolver`-Instanz verschiedene FD-Solver an die allgemeine Solver-Schnittstelle des MCP-Frameworks - die `Solver`-Typklasse - anzuschließen. Somit wird nur noch vom konkreten FD-Solver und dessen Lösungstechniken (z.B. Implementierung der Constraint-Propagierung) abstrahiert. Die Constraints, über denen ein (Baum-)Modell definiert wird, sind hingegen für alle FD-Solver gleich.
Beim Aufruf eines bestimmten FD-Solvers übersetzt die FD-Schicht des MCP-Frameworks die Solver-unabhängigen *high-level* Constraints der MCP-Modellierungssprache intern in *low-level* Constraints, die von dem aufgerufenen Solver unterstützt werden. Dies ermöglicht die Entwicklung von Solver-unabhängig Modellen und Modelltransformationen. Gleichzeitig können konkrete FD-Solver die ihnen bekannten Constraints effizient verarbeiten, ohne sich um die Modellierung kümmern zu müssen. Dieser Übersetzungsvorgang, der als Zwischendatenstruktur einen sogenannten *constraint network graph* erzeugt, wird in [11] ausführlich erläutert.

Modellierung des N-Damen-Problems:
Abschließend soll nun das MCP-Modell für das N-Damen-Problem betrachtet werden. Mit den gerade vorgestellten Constraint-Funktionen kann man direkt ein entsprechendes MCP-Baummodell beschreiben. Bei der Beschreibung kann man Haskells do-Notation verwenden, da die Baummodelle monadisch sind.

```
noattack i j qi qj = do
  qi      @/=  qj
  qi  +  i  @/=  qj  +  j
  qi  -  i  @/=  qj  -  j

queens :: (FDSolver solver) => ModelInt -> Tree solver ModelCol
queens n = exists $ \p -> do
  size p @= n
  p `allin` (cte 0,n-1)
  loopall (cte 0,n-2) $ \i ->
    loopall (i+1,n-1) $ \j ->
      noattack i j (p!i) (p!j)
  return p
```

Listing 34: Modellierung des N-Damen-Problems mit dem MCP-Framework

Ein MCP-Modell beginnt üblicherweise mit dem Aufruf von `exists`. Diese Funktion erwartet eine Funktion, die als Argument einen Solver-Term (hier die Collection `p`) entgegennimmt und ein neues Baummodell liefert. Die Größe der MCP-Collection wird dann durch das `size`- und das `(@=)`-Constraint auf die übergebene Problemgröße `n` festgelegt. Das `allin`-Constraint gibt den Wertebereich für alle Constraint-Variablen in der Collection vor. Mit Hilfe zweier Schleifen und des selbst definierten `noattack`-Constraints werden alle notwendigen Beschränkungen erzeugt, so

dass keine Dame mit einer anderen in der gleiche Reihe oder Diagonale des Schachbretts steht. Abschließend wird ein `Return`-Knoten mit der Collection `p` erzeugt. Über den in diesem Knoten angegebenen Constraint-Variablen führen die FD-Solver beim Lösen des Modells das Labeling durch.
Bei der Modellierung kommen außerdem zwei Hilfsfunktionen zum Einsatz: Mit `cte :: Integral a => a -> ModelInt` kann man ganze Zahlen in einen MCP-Integer-Ausdruck umwandeln. Und mit `(!) :: ColExpr t c b -> Expr t c b -> Expr t c b` kann man auf ein bestimmtes Feld einer MCP-Collection zugreifen.

Weitere Informationen findet man in den wissenschaftlichen Artikeln der Entwickler, zum Beispiel über die Integration der Gecode-Solver-Bibliothek in das MCP-Framework ([10]) oder über die Implementierung der primitiven Suchstrategien und Such-Transformer ([9]). Wer das Framework selbst ausprobieren möchte, findet das entsprechende Package in der HackageDB:
http://hackage.haskell.org/package/monadiccp

3. Vorüberlegungen und Grundlegende Idee

Das Ziel dieser Arbeit ist die Integration von Constraint-Solvern in die funktional-logische Sprache Curry. Genauer gesagt sollen generische Schnittstellen zum Einbinden von Constraint-Bibliotheken und zum Anschluss von Constraint-Solvern für die KiCS2-Curry-Implementierung entwickelt werden. Getestet werden sollen diese Schnittstellen durch die Entwicklung eines Finite-Domain-Constraint-Programming-Systems bestehend aus einer Finite-Domain-Constraint-Bibliothek und der beispielhaften Integration der Solver des Monadic-Constraint-Programming-Frameworks in KiCS2.

Ein Constraint-Programming-System besteht in der Regel aus einer Modellierungseinheit zur Beschreibung des Problems und einer Komponente zur Lösung des modellierten Problems. Ein Problem wird dabei durch Angabe von Regeln, Eigenschaften oder Beschränkungen (Constraints) für sogenannte Constraint-Variablen beschrieben. Diese Problembeschreibung erfolgt häufig mit Hilfe einer speziellen Modellierungssprache. Die Solver-Komponente versucht dann, durch Einsatz verschiedener Techniken wie Constraint-Propagierung oder Labeling von Variablen (= "Ausprobieren" von Variablenbelegungen) Lösungen für das modellierte Problem zu finden.
Curry ist aufgrund seines deklarativen Stils und der Unterstützung logischer Features wie der Programmierung mit freien Variablen gut für die Einbettung einer Sprache zur Modellierung von Constraints geeignet. Die direkte Implementierung eines Constraint-Solvers in Curry wird jedoch durch das Prinzip der referentiellen Transparenz erschwert. Dieses besagt, dass der Wert eines Ausdrucks nur von dessen Umgebung und nicht vom Zeitpunkt oder der Reihenfolge der Auswertung abhängt. Anders formuliert: Curry ist eine zustandslose Sprache ohne Seiteneffekte.
Dies wird bei der Implementierung eines Constraint-Solvers zum Problem, denn ein solcher verwendet üblicherweise einen sogenannten Constraint-Speicher zur Verwaltung der Constraints. Solange alle Constraints im Speicher erfüllbar sind, befindet sich der Solver in einem konsistenten Zustand. Durch das Eintragen weiterer Constraints kann sich dieser Zustand aber jederzeit ändern. Diese zustandsorientierten Berechnungen sind in einer seiteneffektfreien Sprache jedoch nicht oder wenn, dann nur unter Einsatz spezieller Konzepte implementierbar.
Beispielsweise gibt es in der funktionalen Sprache Haskell, die ebenfalls referentiell transparent ist, das Konzept der Monade zur Realisierung seiteneffektbehafteter Berechnungen.
Durch Haskells lazy evaluation ist die Auswertungsreihenfolge relativ unvorhersehbar. Für bestimmte Anwendungen wie zum Beispiel das Lesen einer Datei, die Ausgabe eines Textes auf der Kommandozeile oder eben die Aktualisierung eines Constraint-Speichers ist jedoch eine feste Ausführungsabfolge entscheidend. Mit Monaden kann man die sequentielle Ausführung derartiger seiteneffektbehafteter Berechnungen garantieren. Wie Tom Schrijvers, Peter Stuckey und Phil Wadler gezeigt haben, ist es mit Hilfe von Monaden auch möglich, einen Constraint-Solver direkt in Haskell zu implementieren [9].
Effizientere und besser optimierte Solver-Bibliotheken wurden allerdings in imperativen Sprachen wie C++ oder Java realisiert, z.B. Gecode [3] oder TAILOR [4]. Daher ist das Ziel dieser Arbeit nicht die direkte Realisierung eines Curry-Constraint-Solvers, sondern vielmehr die Entwicklung

einer generischen Schnittstelle, mit deren Hilfe man extern implementierte Solver in Curry integrieren kann.
Wie bereits erwähnt wird für die Entwicklung die KiCS2-Curry-Implementierung verwendet, die funktional-logische Curry-Programme in rein funktionale Haskell-Programme übersetzt. Das bedeutet, KiCS2 bildet alle Datentypen und Funktionen eines Curry-Programms auf entsprechende Haskell-Typen und -Funktionen ab. Weiterhin bietet KiCS2 dem Programmierer die Möglichkeit, Curry-Funktionen extern, also durch Angabe einer Haskell-Implementierung, zu definieren. Weite Teile dieser Arbeit wurden auf diese Weise in Haskell implementiert.
Zur Realisierung der Modellierungskomponente wird KiCS2 zunächst um eine Bibliothek für Finite-Domain-Constraints erweitert. Diese Constraints sollen dann durch Anbindung der FD-Solver des Monadic-Constraint-Programming-Frameworks an KiCS2 gelöst werden. Die Frage ist nun, wie man ein in Curry mit Hilfe der Finite-Domain-Constraint-Bibliothek formuliertes Constraint-Problem in Haskell repräsentiert und diese Haskell-Repräsentation an den Solver übergibt.
KiCS2 unterstützt bereits Gleichheits-Constraints der Form $\texttt{e}_1$ `=:=` $\texttt{e}_2$, durch die zwei Ausdrücke $\texttt{e}_1$ und $\texttt{e}_2$ unifiziert werden. Da es derartige Constraints in Haskell nicht gibt, wird die Haskell-Darstellung eines jeden Curry-Datentyps in KiCS2, um einen sogenannten `Guard`-Konstruktor erweitert, mit dem man die Möglichkeit hat, Werte durch Constraints zu beschränken. Allgemein hat ein solcher Guard-Ausdruck die folgende Form: `Guard cs e`. Hierbei ist `e` ein Wert eines bestimmten Curry-Datentyps und `cs` die Haskell-Darstellung der Curry-Constraints, durch die `e` beschränkt wird. Somit ist ein Ausdruck der Form `Guard cs e` so zu interpretieren, dass der Wert `e` nur dann gültig ist, wenn die Constraints `cs` erfüllbar sind.
Bei der Auswertung eines `Guard`-Ausdrucks wird also ebenfalls eine Lösungskomponente aufgerufen, die die Erfüllbarkeit der Constraints prüft. Weitere Details und Beispiele zur Implementierung des Gleichheits-Constraints sowie zur Auswertung in KiCS2 findet man im entsprechenden Grundlagen-Kapitel.
Die Idee ist nun, dass man auch für die Finite-Domain-Constraints der Curry-Bibliothek eine entsprechende Darstellung in Haskell definiert und diese Haskell-Repräsentation der Constraints mit Hilfe des `Guard`-Konstrukts nach dem gleichen Schema durch die Implementierung durchreicht, wie dies auch mit den bisherigen Curry-Constraints geschieht. Bei Aufruf einer Constraint-Funktion aus der Bibliothek soll also in Haskell ein `Guard`-Ausdruck passenden Typs, der die zugehörige Haskell-Darstellung des Constraints enthält, erzeugt werden. Da die Constraints eines Modells dem Solver nicht einzeln übergeben werden sollen, werden vor der Auswertung alle erzeugten `Guard`-Ausdrücke zu einem einzigen `Guard`-Ausdruck zusammengefasst, der eine Liste aller Haskell-Constraints enthält. Bei der Auswertung eines solchen `Guard`-Ausdrucks durch KiCS2 werden schließlich alle Haskell-Constraints in dieser Liste in Constraints der MCP-FD-Modellierungssprache übersetzt und das resultierende Modell durch einen MCP-Solver gelöst. Die nachfolgende Grafik verdeutlicht diesen Vorgang nochmals.

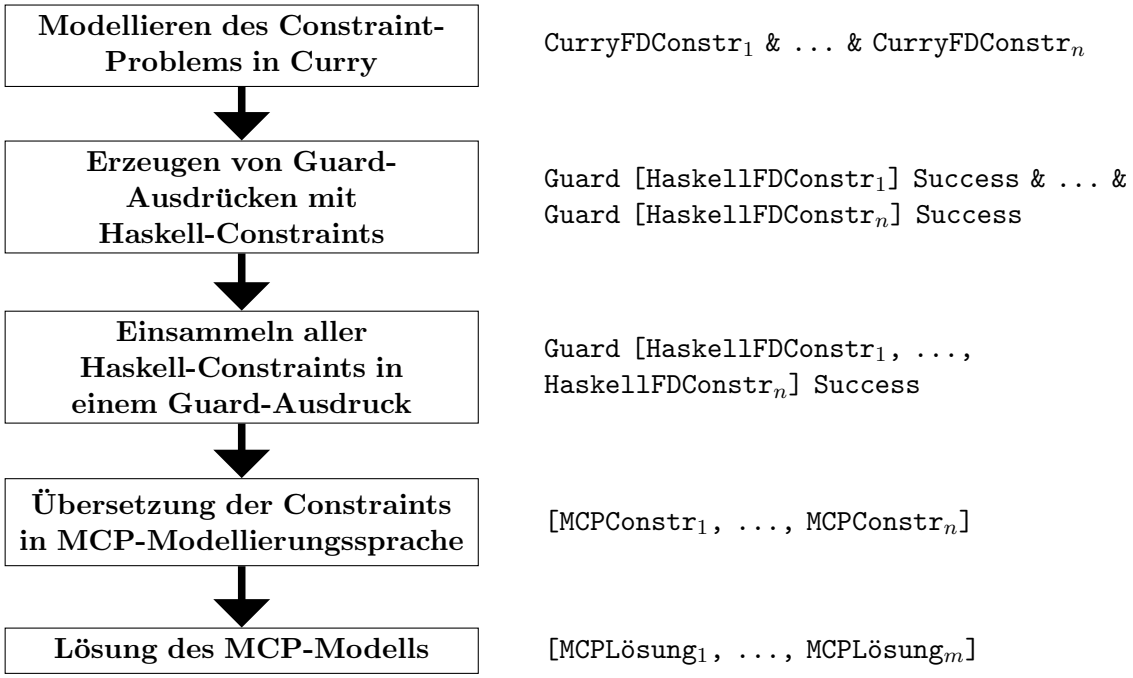


Abbildung 2: Vorgang zum Lösen von Finite-Domain-Constraints in KiCS2

4. Implementierung

Die ausführliche Beschreibung der Implementierung in diesem Kapitel folgt genau dem Vorgehen bei der Entwicklung der generischen Constraint-Solver-Schnittstellen für KiCS2. Das heißt, der erste Teilabschnitt beschreibt die Erweiterung von KiCS2 um eine Bibliothek für Finite-Domain-Constraints, wie man diese Constraints in Haskell repräsentiert und wie man sie mit den bereits vorhandenen Gleichheits-Constraints kombinieren kann.
Der zweite Unterabschnitt behandelt dann ausführlich das Vorgehen zum Anschluss der Constraint-Solver des Monadic-Constraint-Programming-Frameworks. Es wird unter anderem erklärt, wie die FD-Constraints in die MCP-Modellierungssprache übersetzt werden, wie daraus dann ein spezielles MCP-(Baum-)Modell für einen konkreten Solver generiert wird und wie die vom Solver gefundenen Lösungen in KiCS2 ausgegeben werden können.
Im letzten Abschnitt des Implementierungskapitels wird schließlich von der speziellen Finite-Domain-Constraint-Implementierung abstrahiert, indem zunächst die wichtigsten Ideen und Konzepte aus dieser konkreten Realisierung abgeleitet und diese dann zur Entwicklung generischer Schnittstellen zur Integration beliebiger Constraint-Solver verwendet werden. Abschließend wird gezeigt, wie man die bisherige Integration der FD-Constraints und der MCP-Solver auch über das generische Interface an KiCS2 anschließen kann.

4.1. Entwicklung einer Finite-Domain-Constraint-Bibliothek für KiCS2

Dieser Abschnitt beschreibt die Entwicklung einer FD-Constraint-Bibliothek für KiCS2. Zunächst werden die von der Bibliothek bereitgestellten Constraints vorgestellt. Dann wird auf ihre externe Implementierung in Haskell eingegangen. Abschließend wird erklärt, wie man die FD-Constraints mit der Curry-Unifikation bei der Modellierung von Constraint-Problemen kombinieren kann.

4.1.1. KiCS2-CLPFD-Bibliothek

Die Finite-Domain-Constraint-Bibliothek für KiCS2 (CLPFD-Bibliothek) orientiert sich stark an der entsprechenden PAKCS-Bibliothek. Sie definiert die folgenden Funktionen:

```
(=#),(/=#),(<#),(<=#),(>#),(>=#) :: Int -> Int -> Success
(+#),(-#),(*#)    :: Int -> Int -> Int
domain            :: [Int] -> Int -> Int -> Success
allDifferent      :: [Int] -> Success
sum               :: [Int] -> Int
genVars           :: Int -> [Int]
labeling          :: [Int] -> Success
labelingWith      :: LabelingStrategy -> [Int] -> Success
```

Listing 35: CLPFD-Bibliothek

Wie man sieht, sind die Constraints über einem `Int`-Wertebereich definiert und verwenden mit Ausnahme der arithmetischen Operationen den speziell für Constraints in Curry eingeführten Typ `Success` für den Rückgabewert. Auf diese Weise können bei der Modellierung eines Problems mehrere Constraints mit dem vordefinierten Konkatenationsoperator `(&)` verknüpft werden. Neben den bekannten arithmetischen Operationen und den relationalen Vergleichsoperatoren, stellt die CLPFD-Bibliothek noch einige Hilfsfunktionen und komplexere Constraints zur Verfügung:

- Mit `domain vs min max` beschränkt man alle Finite-Domain-Variablen in der Liste `vs` auf den Wertebereich gegeben durch einen minimal möglichen Wert `min` und einen maximal möglichen Wert `max`.

- Das Constraint `allDifferent vs` erzwingt, dass die Werte der Constraint-Variablen in der Liste `vs` paarweise verschieden sind.

- `sum vs` liefert die Summe der Werte aller Constraint-Variablen in der Liste `vs`.

- Mit `genVars n` kann man eine Liste von `n` freien Variablen erzeugen. Falls man einer Constraint-Funktion eine solche Liste als Argument übergibt, so werden die freien Variablen KiCS2-intern auf eine entsprechende Liste von Constraint-Variablen abgebildet. Diese Abbildung von freien Variablen auf Constraint-Variablen wird im nächsten Abschnitt dieses Kapitels beschrieben.

- Durch `labeling vs` kann man angeben, dass das Labeling, also das Ausprobieren von Variablenbelegungen gemäß gültigem Wertebereich, über allen Constraint-Variablen in der Liste `vs` durchgeführt werden soll. Während bei der einfachen `labeling`-Funktion das Labeling der Variablen in der gegebenen Reihenfolge durchgeführt wird, kann man der Funktion `labelingWith` zusätzlich eine Labeling-Strategie übergeben. Die CLPFD-Bibliothek unterstützt bislang die folgenden vier Strategien:

  - `InOrder`: Labeling in der ursprünglichen Reihenfolge,

  - `FirstFail`: Beim Labeling wird die Constraint-Variable mit dem kleinsten (dem am weitesten eingeschränkten) Wertebereich bevorzugt,

  - `MiddleOut`: Beginnt die Belegung in der Mitte der Liste aller Labeling-Variablen,

  - `EndsOut`: Beginnt die Belegung abwechselnd von beiden Enden der Liste aller Labeling-Variablen.

Bis auf `genVars` und `labeling` werden alle Funktionen der CLPFD-Bibliothek extern implementiert. Das bedeutet, die eigentliche Definition dieser Funktionen erfolgt nicht in Curry sondern in Haskell. Dazu werden die entsprechenden Funktionen im Curry-Code mit dem Schlüsselwort `external` gekennzeichnet. Die Implementierung der so deklarierten Funktionen in Haskell erfolgt

dann in einer Datei *External_<Curry-Modulname>.hs* im gleichen Verzeichnis. Die vom Programmierer implementierten Haskell-Definitionen aus diesem Modul werden vom KiCS2-Compiler letztendlich an den bei der Übersetzung generierten Haskell-Code angehängt.
Im Folgenden soll die Curry-Implementierung der Funktionen `(+#)` und `domain` beispielhaft näher betrachtet werden.

```
(+#) :: Int -> Int -> Int
x +# y = ((prim_FD_plus $!! x) $!! y)

prim_FD_plus :: Int -> Int -> Int
prim_FD_plus external
```

Listing 36: Addition auf FD-Termen (Curry)

Die Addition zweier FD-Constraint-Terme wird durch die externe Funktion `prim_FD_plus` realisiert. Der `($!!)`-Operator sorgt dafür, dass die beiden Argumente `x` und `y` strikt zur Normalform ausgewertet werden, bevor `prim_FD_plus` auf sie angewandt wird. Durch diese strikte Auswertung müssen später bei der externen Implementierung von `prim_FD_plus` beim Pattern-Matching weniger Fälle betrachtet werden (siehe unten).

Bei Constraint-Funktionen mit Listen-Argumenten, wie beispielsweise `domain`, wird durch Aufruf der in der *Curry-Prelude* definierten Hilfsfunktion `ensureSpine` zunächst überprüft, ob die übergebene Liste in eine sogenannte *spine*-Form überführbar ist. Dabei wird sichergestellt, dass die Liste endlich ist und keine freien Variablen als Teillisten enthält (durch den Aufruf von `ensureNotFree`). Die Listenelemente werden bei diesem Vorgehen zunächst nicht weiter betrachtet. Bevor jedoch `prim_domain` angewandt wird, werden sie mit Hilfe des `($!!)`-Operators genauso zur Normalform ausgewertet wie die Argumente für die untere und obere Begrenzung des Wertebereichs:

```
domain :: [Int] -> Int -> Int -> Success
domain vs l u = ((prim_domain $!! (ensureSpine vs)) $!! l) $!! u

prim_domain :: [Int] -> Int -> Int -> Success
prim_domain external

ensureSpine :: [a] -> [a]
ensureSpine l = ensureList (ensureNotFree l)
 where ensureList []     = []
       ensureList (x:xs) = x : ensureSpine xs
```

Listing 37: Domain-Constraint (Curry)

Die übrigen Funktionen der Bibliothek werden nach dem gleichen Prinzip implementiert. Das heißt, die Argumente der Constraint-Funktionen werden normalisiert, bevor sie an die jeweilige

extern implementierte Funktion übergeben werden. Bei Listen-Argumenten wird zusätzlich die Überführbarkeit in die *spine*-Form überprüft.

Als Anwendungsbeispiel für die vorgestellte CLPFD-Bibliothek betrachten wir nun erneut das im Grundlagenkapitel über Curry vorgestellte N-Damen-Problem. Da bislang noch nicht darauf eingegangen wurde, ob und inwieweit die FD-Constraints der Bibliothek mit der Curry-Unifikation durch `(=:=)` kombinierbar sind, wird im Folgenden die Modellierung einer konkreten Instanz des N-Damen-Problems gezeigt, nämlich das 4-Damen-Problem:

```
fourQueens = let queens = [q1,q2,q3,q4]
             in domain queens 1 4 &
                all_safe queens &
                labeling queens
 where q1,q2,q3,q4 free

all_safe [] = success
all_safe (q:qs) = safe q qs 1 & all_safe qs

safe :: Int -> [Int] -> Int -> Success
safe _ [] _ = success
safe q (q1:qs) p = no_attack q q1 p & safe q qs (p+#1)

no_attack q1 q2 p = q1 /=# q2 & q1 /=# q2 +# p & q1 /=# q2 -# p
```

Listing 38: Beipiel: 4-Damen-Problem

Dabei verwendet man die gleichen Hilfs-Constraints (`all_safe`,`safe` und `no_attack`), die auch schon bei der Modellierung des N-Damen-Problems mit PAKCS zum Einsatz gekommen sind. Daher wird für eine genauere Erklärung dieses Beispiels auf die entsprechende Beschreibung im Grundlagenkapitel über Curry verwiesen.

4.1.2. Darstellung der Constraints in Haskell

Nachdem KiCS2 um eine Bibliothek für Finite-Domain-Constraints erweitert wurde, beschreibt der folgende Abschnitt die Implementierung der externen Funktionen, mit deren Hilfe diese Bibliothek realisiert wird.
Zur Erinnerung: Die grundlegende Idee ist es, jeden Aufruf einer Constraint-Funktion aus der CLPFD-Bibliothek auf einen `Guard`-Ausdruck mit dem entsprechenden Constraint in Haskell abzubilden. Bevor nun eine hierfür geeignete Haskell-Repräsentation von Finite-Domain-Constraints definiert wird, soll aber zunächst eine Darstellung für FD-Terme in Haskell eingeführt werden.

Finite-Domain-Constraints werden üblicherweise über Termen eines bestimmten Typs definiert. Dieser Typ wird durch den Wertebereich, über dem die Constraint-Variablen definiert werden, vorgegeben. In der Regel verwendet man boolesche oder Integer-Terme. Neben Constraint-Variablen zählen auch Konstanten zu den Basistermen. Durch die Zusammenfassung in Listen oder Anwendung arithmetischer Operationen können aus solchen (Basis-)Termen komplexere Termausdrücke konstruiert werden.
Da die Constraints der CLPFD-Bibliothek über einer `Int`-Domain definiert wurden, ist also ein Typ zur Repräsentation von Integer-FD-Termen in Haskell gesucht. Hierfür wird die folgende polymorphe Datenstruktur eingeführt:

```
data FDTerm a = Const a
              | FDVar ID
```

Listing 39: Repräsentation von FD-Termen (Haskell)

Ein Wert vom Typ `FDTerm a` ist nun entweder eine Konstante mit einem Wert vom Typ `a` oder eine FD-Variable. Zur Identifikation einer FD-Variablen wird der Typ `ID` wiederverwendet. KiCS2 benutzt diesen `ID`-Typ zur eindeutigen Identifikation von `Choice`s, mit deren Hilfe unter anderem die freien Variablen aus Curry in Haskell repräsentiert werden. Die Wiederverwendung dieses Typs erleichtert später die Zuordnung von freien Variablen in einem Curry-Constraint-Modell zu deren FD-Variablen-Repräsentation im entsprechenden Haskell-Modell.

Nun, da man mit dem Typ `FDTerm Int` Integer-FD-Terme repräsentieren kann, ist es möglich, eine Datenstruktur zu definieren, mit deren Hilfe man die Constraints aus der CLPFD-Bibliothek in Haskell darstellen kann:

```
data FDConstraint
  = FDRel RelOp (FDTerm Int) (FDTerm Int)
  | FDArith ArithOp (FDTerm Int) (FDTerm Int) (FDTerm Int)
  | FDSum [FDTerm Int] (FDTerm Int)
  | FDAllDifferent [FDTerm Int]
  | FDDomain [FDTerm Int] (FDTerm Int) (FDTerm Int)
  | FDLabeling LabelingStrategy [FDTerm Int] ID

data ArithOp = Plus | Minus | Mult
data RelOp   = Equal | Diff | Less | LessEqual
```

Listing 40: Repräsentation von FD-Constraints (Haskell)

Wie man sehen kann, besitzt der Datentyp `FDConstraint` für jede Constraint-Funktion der oben vorgestellten Bibliothek einen dazu passenden Konstruktor wie beispielsweise `FDDomain` für die `domain`-Funktion. Analog zu den `Int` und `[Int]`-Argumenten der CLPFD-Bibliothek erhalten diese Konstruktoren Integer-FD-Terme und/oder Listen solcher Terme als Argumente.

Die relationalen Vergleichsoperatoren wie `(=#)`, `(/=#)` usw. werden durch einen `FDRel`-Konstruktorterm mit passendem `RelOp`-Wert dargestellt. Für `(>#)` und `(>=#)` gibt es nicht extra einen `RelOp`-Wert, da sich diese Fälle durch Vertauschung der Argumente auch mit Hilfe von `Less` und `LessEqual` ausdrücken lassen. Nach dem gleichen Prinzip werden die Addition, Subtraktion und Multiplikation auf FD-Termen durch `FDArith`-Konstruktorterme mit dem entsprechenden arithmetischen Operator `ArithOp` repräsentiert.
Einige Konstruktoren wie `FDArith` oder `FDSum` erhalten mehr Argumente als ihre zugehörige Funktion in der CLPFD-Bibliothek. Dies hängt damit zusammen, dass komplexe arithmetische Ausdrücke in der Haskell-Darstellung "flachgeklopft" werden sollen. Betrachtet man beispielsweise den folgenden Curry-Ausdruck `(x +# 7) *# 2 where x free`, so kann dieser durch Einführung von Hilfsvariablen, an die die (Zwischen-)Ergebnisse der einzelnen arithmetischen Operationen gebunden werden, wie folgt "flachgeklopft" werden: `y = x +# 7`, `z = y *# 2`. Der Konstruktor `FDArith` drückt nun genau dies aus: Er erhält als drittes Argument eine neue FD-Variable, an die das Ergebnis des arithmetischen Ausdrucks gebunden wird. Somit wird der Beispielausdruck durch die beiden folgenden `FDConstraint`-Konstruktorterme in Haskell abgebildet:

```
FDArith Plus (FDVar <id_x>) (Const 7) (FDVar <id_y>)
FDArith Mult (FDVar <id_y>) (Const 2) (FDVar <id_z>)
```

Für das Summen-Constraint - in Haskell dargestellt durch den `FDSum`-Konstruktor - gilt dies genauso.
Das "Labeling-Constraint" erhält eine unbenutzte `ID` als zusätzliches Argument, die für die Ausgabe der durch den Solver gefundenen Lösungen benötigt wird (vergleiche hierzu Kapitel 4.2.3). Es handelt sich hierbei nicht um ein Constraint im eigentlich Sinne, sondern es dient vielmehr dazu, die für das Labeling benötigten Informationen bis zum Aufruf der Constraint-Solver durch die Implementierung durchzureichen.

Bevor man nun mit Hilfe des `FDConstraint`-Datentyps die externen Funktionen der CLPFD-Bibliothek implementieren kann, sind noch zwei Dinge notwendig: Zum einen muss der Constraint-Typ von KiCS2, der in den `Guard`-Ausdrücken verwendet wird, derart erweitert werden, dass auch Constraints vom Typ `FDConstraint` unterstützt werden.
Zum anderen benötigt man Funktionen, mit denen man die Haskell-Repräsentation eines Curry-Int (`C_Int`) bzw. einer Curry-Int-Liste `OP_List C_Int` in einen Integer-FD-Term bzw. eine Liste von Integer-FD-Termen umwandeln kann.
Die Erweiterung des Constraint-Typs zur Unterstützung von `FDConstraints` lässt sich relativ einfach umsetzen. Man erweitert den in KiCS2-`Guard`-Ausdrücken verwendeten `Constraints`-Typ (vergleiche Kapitel 2.2.2) um einen Konstruktor für Finite-Domain-Constraints:

```
data Constraints
  = ...
  | FDConstr [FDConstraint]
```

Listing 41: Erweiterung des Datentyps `Constraints`

Man verwendet eine Liste vom Typ `FDConstraint`, damit mehr als ein FD-Constraint in einem `Guard`-Ausdruck durch die Implementierung gereicht werden kann. Das Ziel soll es später sein, die FD-Constraints aus verschiedenen `Guard`-Ausdrücken in einem einzelnen solchen Ausdruck zu sammeln, bevor ein geeigneter Solver aufgerufen wird.

Für die Definition einer Funktion `toFDTerm :: C_Int -> FDTerm Int` zur Transformation der Haskell-Darstellung eines Curry-Ints in einen FD-Term soll zunächst der Ausgangstyp `C_Int` näher betrachtet werden:

```
data C_Int
     = C_Int Int#
     | C_CurryInt BinInt
     | Choice_C_Int ID C_Int C_Int
     | Choices_C_Int ID ([C_Int])
     | Fail_C_Int
     | Guard_C_Int Constraints C_Int
```

Listing 42: Repräsentation vom Curry-Typ `Int` in Haskell

Mit den beiden ersten Konstruktoren kann man konstante Integer-Werte einführen und zwar einmal über unboxed Ints (Konstruktor `C_Int`) und einmal über eine binäre Codierung mit `Nat`-Werten (Konstruktor `C_CurryInt`). Die übrigen Konstruktoren tauchen mit entsprechend angepasstem Namen bei jeder Datentypdefinition eines Curry-Typs in Haskell auf (vergleiche Typ `C_Bool` im Grundlagenkapitel über KiCS2). Sie dienen zur Darstellung des Nicht-Determinismus (`Choice_C_Int` und `Choices_C_Int`) oder einer fehlgeschlagenen Berechnung (`Fail_C_Int`) sowie dem Hinzufügen von Constraints (`Guard_C_Int`) in der Haskell-Übersetzung eines Curry-Ausdrucks vom Typ `Int`.
Bei der Implementierung der Übersetzungsfunktion `toFDTerm` wird nun allerdings nicht auf alle diese Fälle gematcht. Da die Argumente der Constraint-Funktionen normalisiert werden, bevor sie an die zugehörige externe Constraint-Funktion übergeben werden (siehe Erklärung des `($!!)`-Operators weiter oben), wird davon ausgegangen, dass bei Aufruf von `toFDTerm` das Argument bereits in Normalform ist. Somit wird das Pattern-Matching bei der Implementierung von `toFDTerm` nur für die Konstruktoren einer Integer-Konstanten bzw. -Variablen durchgeführt:

```
toFDTerm :: C_Int -> FDTerm Int
toFDTerm (Choices_C_Int i@(FreeID _) _) = FDVar i
toFDTerm x                              = Const (fromCurry x)
```

Listing 43: Umwandlung von Curry-FD-Termen in Haskell-FD-Terme

Eine freie Integer-Variable - in Haskell repräsentiert durch den Konstruktor `Choices_C_Int` mit einer `FreeID` als ID-Argument - wird in eine neue FD-Variable mit der gleichen ID `i` transformiert. In allen übrigen Fällen muss es sich bei dem Argument gemäß der obigen Annahme um eine Integer-Konstante handeln und somit wird ein konstanter FD-Term mit dem entsprechenden

Integer-Wert eingeführt. Dazu wird die überladene Hilfsfunktion `fromCurry` verwendet, die die Haskell-Darstellung eines Curry-Typs in den zugehörigen Haskell-Typ konvertiert. Beispielsweise ist `fromCurry` auf dem Typ `C_Bool` folgendermaßen definiert:

```
fromCurry C_True  = True
fromCurry C_False = False
```

Listing 44: Beispiel: `fromCurry` für `C_Bool`

Bei der Implementierung von `toFDTerm` wird `fromCurry` benutzt, um einen Wert vom Typ `C_Int` in einen entsprechenden Wert vom Haskell-Typ `Int` zu überführen.

Nun fehlt noch eine Funktion, mit der man eine Liste von Integern in Curry (`OP_List C_Int`) in eine Liste von FD-Termen in Haskell (`[FDTerm Int]`) übersetzen kann. Zunächst betrachtet man den polymorphen Typ, mit dem Curry-Listen in Haskell repräsentiert werden:

```
data OP_List a
     = OP_List
     | OP_Cons a (OP_List a)
     | Choice_OP_List ID (OP_List a) (OP_List a)
     | Choices_OP_List ID ([OP_List a])
     | Fail_OP_List
     | Guard_OP_List Constraints (OP_List a)
```

Listing 45: Repräsentation vom Curry-Typ `[a]` in Haskell

Dabei ist `OP_List` der Konstruktor für eine leere Liste und `OP_Cons` der Konstruktor, um ein neues Element vom Typ `a` vor eine Liste vom Typ `OP_List a` zu hängen. Man definiert nun eine Übersetzungsfunktion `toFDList` wie folgt:

```
toFDList :: OP_List C_Int -> [FDTerm Int]
toFDList OP_List         = []
toFDList (OP_Cons x xs) = toFDTerm x : toFDList xs
```

Listing 46: Umwandlung von Curry-FD-Listen in Haskell-FD-Listen

Eine leere Curry-Liste wird in eine leere Haskell-Liste überführt. Bei einer nicht-leeren Liste wird das aktuelle Listenelement per `toFDTerm` in einen FD-Term transformiert und per Haskell-Cons-Operator vor die durch rekursiven Aufruf von `toFDList` übersetzte Restliste eingefügt.

Externe Implementierung der CLPFD-Bibliothek:
Nun ist man in der Lage, die Funktionen der CLPFD-Bibliothek extern zu implementieren. Beispielhaft wird zunächst die Implementierung der Additionsfunktion `prim_FD_plus` aus dem Curry-Modell *CLPFD.curry* betrachtet. Wie bereits erwähnt erfolgt die externe Implementierung dieser Funktion in einem Modul *External_CLPFD.hs*. In diesem Haskell-Modul müssen nun beispielsweise Regeln für eine Funktion `external_d_C_prim_FD_plus` angegeben werden. Durch den Präfix

`external_d_C_` werden alle deterministischen externen Funktionen in KiCS2 gekennzeichnet. Entsprechend wird für nicht-deterministische Funktionen der Präfix `external_nd_C_` verwendet. Eine Funktion in KiCS2 gilt als nicht-deterministisch, wenn sie nicht-deterministische Werte, also z.B. neue freie Variablen einführt. Dazu erhalten nicht-deterministische Funktion durch ein zusätzliches `IDSupply`-Argument einen quasi unendlichen Vorrat unbenutzter `ID`s.
Wie weiter oben beschrieben sollen arithmetische Ausdrücke "flachgeklopft" werden. Das heißt, jedes Zwischenergebnis einer einfachen arithmetischen Operation soll an eine neue Constraint-Variable gebunden werden. Zur Erzeugung einer neuen Constraint-Variablen wird allerdings eine unverbrauchte `ID` benötigt, welche man beispielsweise durch Einführung einer neuen Curry-Variablen erhält. Man könnte `prim_FD_plus` nun wie zuvor beschrieben mit Hilfe einer nicht-deterministische externen Funktion implementieren, die über ihren `IDSupply` neue `ID`s zur Verfügung stellt. Durch eine kleine Anpassung der Curry-Implementierung muss sich der Programmierer aber gar nicht selbst um die Erzeugung dieser freien Variablen in Haskell kümmern:

```
(+#) :: Int -> Int -> Int
x +# y = ((prim_FD_plus $!! x) $!! y) result where result free

prim_FD_plus :: Int -> Int -> Int -> Int
prim_FD_plus external
```

Listing 47: Addition auf FD-Termen (Curry) (angepasst)

Man übergibt einer Funktion wie `prim_FD_plus` einfach eine freie Variable namens `result` als zusätzliches Argument. An diese freie Variable soll dann später das Ergebnis der Addition gebunden werden. Dadurch wird die Einführung des Nicht-Determinismus auf die Ebene der `(+#)`-Funktion verschoben. Bei der Übersetzung dieser Funktion sorgt KiCS2 nun selbst für die Erzeugung einer neuen freien Variablen, so dass man sich darum nicht mehr selbst kümmern muss.
Nach dem gleichen Prinzip wird auch der Code für die anderen arithmetischen Constraint-Funktionen (u.a. `sum`) in Curry angepasst.
Nach diesen Anpassungen kann man `external_d_C_prim_FD_plus` nun wie folgt definieren:

```
external_d_C_prim_FD_plus :: C_Int -> C_Int -> C_Int
                          -> ConstStore -> C_Int
external_d_C_prim_FD_plus x y result _ =
  let c = [newArithConstr Plus x y result]
  in Guard_C_Int (FDConstr c) result

newArithConstr :: ArithOp -> C_Int -> C_Int -> C_Int
               -> FDConstraint
newArithConstr arithOp x y result =
  FDArith arithOp (toFDTerm x) (toFDTerm y) (toFDTerm result)
```

Listing 48: Addition auf FD-Termen (Haskell)

Wie man sieht, werden die Curry-Int-Argumente bei der Implementierung einer externen Funktion durch ihre entsprechende Haskell-Darstellung (also hier `C_Int`-Argumente) ersetzt. Außerdem wird als zusätzliches Argument ein sogenannter `ConstStore` übergeben. Dieser `ConstStore` ist eine Optimierung für den Zugriff auf Currys Bindungs-Constraints und spielt keine Rolle für die hier vorgestellte Implementierung der FD-Constraint-Bibliothek. Daher wird dieses Argument nur bei Aufruf von Hilfsfunktionen aus der *Curry-Prelude* durchgereicht (siehe unten) und ansonsten ignoriert.
Zur Implementierung der arithmetischen Operationen wird die Hilfsfunktion `newArithConstr` definiert. Diese erzeugt für den jeweils übergebenen arithmetischen Operator und dessen Argumente einen passenden `FDArith`-Konstruktorterm, indem die Argumente mittels `toFDTerm` in Integer-FD-Terme übersetzt werden.
Die Funktion `external_d_C_prim_FD_plus` erzeugt durch Aufruf von `newArithConstr` ein Additions-Constraint, das die Summe der FD-Terme von `x` und `y` an die FD-Term-Darstellung von `result` bindet. Mit Hilfe des weiter oben eingeführten `FDConstr`-Konstruktors wird schließlich ein neuer `Guard`-Ausdruck konstruiert, welcher die freie Variable `result` vom Typ `C_Int` mit dem zuvor erzeugten Additions-Constraint beschränkt.
Der erzeugte `Guard`-Ausdruck ist also so zu interpretieren, dass die freie Variable für das Ergebnis der Addition (`result`) erst dann ausgewertet werden kann, wenn das sie beschränkende Additions-Constraint gelöst wurde. Die übrigen arithmetischen Operatoren sowie das Summen-Constraint werden nach dem gleichen Prinzip realisiert.

Als weiteres Beispiel soll die externe Implementierung von `prim_FD_equal` betrachtet werden:

```
external_d_C_prim_FD_equal :: C_Int -> C_Int -> ConstStore
                           -> C_Success
external_d_C_prim_FD_equal x y _ =
  let c = [newRelConstr Equal x y]
  in Guard_C_Success (FDConstr c) C_Success

newRelConstr :: RelOp -> C_Int -> C_Int -> FDConstraint
newRelConstr relOp x y = FDRel relOp (toFDTerm x) (toFDTerm y)
```

Listing 49: Gleichheit auf FD-Termen (Haskell)

Analog zur Funktion `newArithConstr` definiert man eine Hilfsfunktion `newRelConstr`, welche für den übergebenen relationalen Operator und dessen Argumente ein passendes `FDRel`-Constraint erzeugt. Durch Aufruf dieser Hilfsfunktion führt `external_d_C_prim_FD_equal` dann ein Gleichheits-Constraint über den Argumenten `x` und `y` ein und konstruiert schließlich einen neuen `Guard`-Ausdruck vom Typ `C_Success`. Dieser `Guard`-Ausdruck wird zu `C_Success` reduziert, falls das Gleichheits-Constraint erfüllbar ist.

Abschließend wird die externe Implementierung von `prim_domain` beschrieben:

```
external_d_C_prim_domain :: OP_List C_Int -> C_Int -> C_Int
                         -> ConstStore -> C_Success
external_d_C_prim_domain vs l u _ =
  let c = [FDDomain (toFDList vs) (toFDTerm l) (toFDTerm u)]
  in Guard_C_Success (FDConstr c) C_Success
```

Listing 50: Domain-Constraint (Haskell)

Auch `external_d_C_prim_domain` erzeugt zunächst eine passende Haskell-Repräsentation des Domain-Constraints in Form eines `FDDomain`-Konstruktorterms, wobei die Argumente mittels `toFDTerm` bzw. `toFDList` in ihre Term- bzw. Term-Listen-Darstellung überführt werden. Anschließend verwendet man das erzeugte Wertebereichs-Constraint zur Konstruktion eines `Guard`-Ausdrucks.
Nach diesem Schema werden auch die übrigen externen Constraint-Funktionen implementiert, so dass sie letztendlich alle einen `Guard`-Ausdruck mit dem passenden `FDConstraint` zurückliefern.

Optimierung der Implementierung für konstante Argumente:
Die bisherige Implementierung hat einen Nachteil: Auch für Constraint-Ausdrücke wie `5 +# 7`, `0 <# 2` oder `allDifferent [1,2,3,2]` werden entsprechende `Guard`-Ausdrücke generiert, obwohl man sie direkt berechnen und das Ergebnis zurückgeben könnte. Um diesen unnötigen Overhead zu vermeiden, soll eine zusätzliche Fallunterscheidung in alle extern implementierten Funktionen eingebaut werden: Sind alle Argumente einer solchen Funktion Konstanten, so ruft man eine passende Funktion aus der *Curry-Prelude* auf, um den Constraint-Ausdruck direkt zu berechnen. Andernfalls wird wie zuvor ein passender `Guard`-Ausdruck erzeugt.
Da das Matching auf die verschiedenen Konstruktoren für konstante `C_Int`-Werte relativ umständlich wäre, wird eine neue Typklasse eingeführt, die einem hilft zu erkennen, ob ein Wert eines bestimmten Typs in Grundnormalform (ground normal form, kurz: GNF) ist:

```
class NonDet a => GNFChecker a where
  gnfCheck :: a -> Bool
  gnfCheck x = gnfCheck' (try x)
    where gnfCheck' (Val _) = True
          gnfCheck' _       = False
```

Listing 51: Typklasse `GNFChecker`

Die Typklasse `GNFChecker a` stellt ein Prädikat `gnfCheck :: a -> Bool` zur Verfügung. Hierfür gibt es eine Default-Implementierung: Der übergebene Wert wird mit Hilfe der in KiCS2 vordefinierten Funktion `try :: NonDet a => a -> Try a` in eine generische `Try`-Struktur überführt (vergleiche Kapitel 2.2.3). Anschließend wird durch Matching auf den Konstruktor `Val` festgestellt, ob es sich um einen konstanten Wert in GNF handelt oder nicht.
Zur Anpassung der externen Implementierung der Constraint-Funktionen wird ein `GNFChecker` für Integer- und einer für Listen-Argumente benötigt:

```
instance GNFChecker C_Int

instance GNFChecker a => GNFChecker (OP_List a) where
  gnfCheck OP_List         = True
  gnfCheck (OP_Cons x xs) = gnfCheck x && gnfCheck xs
```

Listing 52: `GNFChecker`-Instanzen

Für Integer-Werte kann die vordefinierte Default-Implementierung verwendet werden. Bei Listen muss hingegen nicht nur getestet werden, ob der oberste Listenkonstruktor konstant ist, sondern zusätzlich müssen auch alle Listenelemente in Grundnormalform sein. Das bedeutet, man benötigt einen `GNFChecker` für den Typ der Listenelemente. Dies wird durch ein entsprechendes Typklassen-Constraint ausgedrückt.
Schließlich wird die Default-Implementierung der Funktion `gnfCheck` überschrieben: Bei einer leeren Liste (`OP_List`) wird direkt `True` zurückgegeben, sonst (`OP_Cons`) werden das aktuelle Listenelement und die Restliste getestet.
Mit diesem Hilfsmittel lässt sich nun beispielsweise die externe Implementierung der Addition von FD-Termen folgendermaßen erweitern:

```
external_d_C_prim_FD_plus :: C_Int -> C_Int -> C_Int
                          -> ConstStore -> C_Int
external_d_C_prim_FD_plus x y result cs
  | gnfCheck x && gnfCheck y = d_OP_plus x y cs
  | otherwise                =
    let c = [newArithConstr Plus x y result]
    in Guard_C_Int (FDConstr c) result
```

Listing 53: Addition auf FD-Termen (Haskell) - erweitert

Anstatt wie oben beschrieben direkt einen `Guard`-Ausdruck zu erzeugen, wird nun überprüft, ob die beiden Argumente des Additions-Constraints `x` und `y` in GNF sind: Falls ja, so wird die von KiCS2 in Haskell generierte Funktion `d_OP_plus` für die Curry-Addition von Integer-Werten aus der *Curry-Prelude* aufgerufen. Andernfalls wird, wie gehabt, ein `Guard`-Ausdruck erzeugt.
Für die übrigen Constraint-Funktionen wird nach dem gleichen Schema ebenfalls eine solche Fallunterscheidung eingebaut. Bei den relationalen Vergleichsoperatoren verwendet man zum Beispiel den entsprechenden booleschen Operator aus der *Curry-Prelude* und gibt dann je nach dem booleschen Rückgabewert dieses Operators `C_Success` oder `Fail_C_Success` zurück.

KiCS2-Interne Repräsentation des 2-Damen-Problems:
Zum Abschluss dieses Kapitels soll nun noch einmal ein Beispiel-Modell in Curry und dessen Repräsentation mit `Guard`-Ausdrücken in Haskell betrachtet werden. Damit das Beispiel vor allem in der Haskell-Darstellung übersichtlich bleibt, wird hier das einfachere 2-Damen-Problem betrachtet. Dieses lässt sich wie folgt mit der CLPFD-Bibliothek modellieren:

```
twoQueens =
  domain [q1,q2] 1 2 &  -- Festlegung des Wertebereichs
  q1 /=# q2 &           -- Reihen,
  q1 /=# q2 +# 1 &      -- aufsteigende Diagonalen und
  q1 /=# q2 -# 1 &      -- abfallende Diagonalen unterschiedlich
  labeling [q1,q2]      -- Labeling in gegebener Reihenfolge
 where q1,q2 free
```

Listing 54: Beispiel: 2-Damen-Problem (Curry)

In diesem Beispiel wurde auf die Hilfsfunktionen verzichtet. Alle erforderlichen Constraints aus der Bibliothek wurden explizit hingeschrieben. Auf diese Weise kann man leichter die Repräsentation des Modells in Curry mit der in Haskell vergleichen. In Haskell wird das obige Modell auf die unten angegebene Konjunktion von `Guard`-Ausdrücken über `FDConstraints` abgebildet. Dabei handelt es sich um `Guard`-Ausdrücke vom Typ `C_Success`. Der Übersichtlichkeit halber wird unten auf den genauen Bezeichner für den Konstruktor verzichtet (`Guard` statt `Guard_C_Success`) und der `FDConstr`-Wrapper um die Constraint-Listen vom Typ `[FDConstraint]` wird ebenfalls weggelassen. Weiterhin wird angenommen, dass die `ID`s der freien Variablen `q1` und `q2` `1` bzw. `2` seien. Zusätzlich eingeführte Hilfsvariablen erhalten entsprechend fortlaufende `ID`s, also `3,4,...`):

```
1 Guard [FDDomain [FDVar 1, FDVar 2] (Const 1) (Const 2)] C_Success &
2 Guard [FDRel Diff (FDVar 1) (FDVar 2)] C_Success &
3 Guard [FDArith Plus (FDVar 2) (Const 1) (FDVar 3)]
4   Guard [FDRel Diff (FDVar 1) (FDVar 3)] C_Success &
5 Guard [FDArith Minus (FDVar 2) (Const 1) (FDVar 4)]
6   Guard [FDRel Diff (FDVar 1) (FDVar 4)] C_Success &
7 Guard [FDLabeling InOrder [FDVar 1, FDVar 2] 5] C_Success
```

Listing 55: Beispiel: Guard-Ausdrücke für 2-Damen-Problem (Haskell)

Wie man sieht, wird für jede der sieben im Modell verwendeten Constraint-Funktionen aus der CLPFD-Bibliothek (`domain`, dreimal `(/=#)`, `(+)`, `(-)` und `labeling`) ein `Guard`-Ausdruck mit dem entsprechenden `FDConstraint`-Konstruktorterm erzeugt.
Auffällig sind die verschachtelten `Guard`-Ausdrücke in den Zeilen 3 und 4 sowie 5 und 6, die zur Abbildung der Constraint-Ausdrücke `q1 /=# q2 +# 1` bzw. `q1 /=# q2 -# 1` in Haskell konstruiert werden. Diese Verschachtelung resultiert aus der Anwendung des `($!!)`-Operators bei der Implementierung von `(/=#)`.
Zur Erinnerung: Dieser Operator sorgt dafür, dass die Argumente einer Funktion vor der Funktionsanwendung normalisiert werden. Somit wird unter anderem `q2 +# 1` zur Normalform ausgewertet. Das bedeutet, es wird der `Guard`-Ausdruck `Guard_C_Int (FDConstr [FDArith Plus (FDVar 2)(Const 1)(FDVar 3)]) r` erzeugt, wobei `3` die `ID` der Hilfsvariable `r` sei, an die das Ergebnis der Addition gebunden wird.

Bei der Normalisierung wird dieser `Guard`-Ausdruck vom Typ `C_Int` nun allerdings auf die Ergebnisebene der Funktion `(/=#)` propagiert. Das heißt, es wird ein `Guard`-Ausdruck vom Typ `C_Success` konstruiert, der das Ergebnis des Ausdrucks `q1 /=# r` mit dem Additions-Constraint `(FDConstr [FDArith Plus (FDVar 2)(Const 1)(FDVar 3)])` beschränkt. Durch Auswertung dieses Ungleichheits-Constraints erhält man letztendlich den inneren `Guard`-Ausdruck.

```
   q1 /=# q2 +# 1
→ (prim_FD_notequal $!! q1) $!! q2 +# 1
→ (prim_FD_notequal $!! q1) $!!
     (((prim_FD_plus $!! q2) $!! 1) result where result free)
...
→ (prim_FD_notequal $!! q1) $!!
     (Guard_C_Int (FDConstr
       [FDArith Plus (FDVar 2) (Const 1) (FDVar 3)]) result)
→ Guard_C_Success (FDConstr
     [FDArith (FDVar 2) (Const 1) (FDVar 3)])
       ((prim_FD_notequal $!! q1) result)
...
→ Guard_C_Success (FDConstr
     [FDArith (FDVar 2) (Const 1) (FDVar 3)])
       (Guard_C_Success (FDConstr
         [FDRel Diff (FDVar 1) (FDVar 3)]) C_Success)
→ Guard [FDArith Plus (FDVar 2) (Const 1) (FDVar 3)]
     Guard [FDRel Diff (FDVar 1) (FDVar 3)] C_Success
```

Listing 56: Beispiel: Auswertung zu Guard-Ausdrücken

4.1.3. Einsammeln aller Finite-Domain-Constraints

Mit der bislang vorgestellten Erweiterung der KiCS2-Implementierung ist es möglich, Finite-Domain-Constraints in Curry auf `Guard`-Ausdrücke in Haskell abzubilden. Die Frage ist nun, wie man die Constraints, die mit diesen `Guard`-Ausdrücken durch die Implementierung gereicht werden, letztendlich an einen Constraint-Solver weitergibt.
Wie im Grundlagenkapitel über KiCS2 bereits erwähnt liefert die Normalisierung eines Ausdrucks in KiCS2 entweder einen deterministischen Wert, der direkt ausgegeben werden kann, oder einen Suchbaum über nicht-deterministischen `Choice`s. Mit Hilfe von Suchalgorithmen kann ein solcher Baum nach weiteren (deterministischen) Ergebniswerten durchsucht werden. Auch die `Guard`-Ausdrücke kommen als Knoten im Suchbaum vor. Bei der Auswertung eines solchen Knotens durch einen Suchalgorithmus wie die Tiefensuche werden die im `Guard`-Ausdruck transportierten Constraints an einen Solver weitergegeben. Dieser Solver, der speziell zur Lösung der Curry-Bindungs-Constraints entwickelt wurde, versucht die übergebenen Constraints zu lösen. Falls die

Constraints erfüllbar sind, gibt der Solver zum einen eine Funktion zurück, mit der die beim Lösen getroffenen Bindungsentscheidungen zurückgesetzt werden können (`reset`), zum anderen wird der Teil des Suchbaums zurückgegeben, in dem die Auswertung zunächst fortgesetzt werden soll (`e'`). Ergibt die Prüfung des Solvers hingegen, dass die Constraints nicht lösbar sind, so wird die Auswertung in diesem Pfad des Baumes abgebrochen:

```
...
dfsGuard cs e = solve cs e >>= \mbSltn -> case mbSltn of
  Nothing          -> mnil
  Just (reset, e') -> dfs cont e' |< reset
...
```

Die Idee ist nun, dass man die Implementierung der Suchalgorithmen um einen Fall für `Guard`-Ausdrücke mit FD-Constraints erweitert. Erreicht die Auswertung einen solchen Ausdruck, so soll ein geeigneter FD-Solver aufgerufen werden, der die Constraints dieser `Guard` löst und mögliche Lösungen in geeigneter Form zurückgibt, so dass die Auswertung fortgesetzt werden kann:

```
...
dfsGuard (FDConstr fdCs) e = let solutions = runFDSolver cs e
                             in dfs cont solutions
...
```

Ein Problem hierbei ist, dass ein `Guard`-Ausdruck bislang nur ein einzelnes Finite-Domain-Constraint enthält. Zur Lösung eines Constraint-Problems benötigt ein Finite-Domain-Solver aber alle zur Modellierung des Problems verwendeten Constraints. Das bedeutet, man muss vor der KiCS2-Auswertung und damit auch vor dem Aufruf eines konkreten FD-Solvers die FD-Constraints aus allen `Guard`-Ausdrücken einsammeln und dann mit den gesammelten Constraints einen einzelnen neuen `Guard`-Ausdruck konstruieren. Mit der Funktion `searchFDCs` wird dies erreicht:

```
searchFDCs :: NormalForm a => a -> [FDConstraint] -> a
searchFDCs x fdCs =
  match sfChoice sfNarrowed choicesCons failCons sfGuard sfVal x
 where
  sfChoice i x1 x2       = choiceCons i (searchFDCs x1 fdCs)
                                        (searchFDCs x2 fdCs)
  sfNarrowed i xs        = choicesCons i
                             (map (\x' -> searchFDCs x' fdCs) xs)
  sfGuard (FDConstr c) e = searchFDCs e (fdCs ++ c)
  sfGuard c e            = guardCons c (searchFDCs e fdCs)
  sfVal v | null fdCs    = v
          | otherwise    = guardCons (FDConstr fdCs) v
```

Listing 57: Einsammeln der FD-Constraints

Bei der Implementierung dieser Funktion werden einige bislang noch nicht vorgestellte Funktionen und Typklassen von KiCS2 verwendet:

- In der Typklasse `NormalForm` fasst KiCS2 alle Typen zusammen, die sich normalisieren lassen. Da die Funktion `searchFDCs` unmittelbar nach der Normalisierung eines Ausdrucks und vor dessen Auswertung durch eine Suchstrategie aufgerufen werden soll, wird ihr Argumenttyp durch das entsprechende Typklassen-Constraint eingeschränkt.

- Die Funktion `match` stellt eine Alternative zur bereits vorgestellten `try`-Funktion dar. Anstatt die Haskell-Darstellung eines Curry-Datentyps zunächst durch Aufruf von `try` in einer generische `Try`-Struktur zu überführen und dann auf dieser mittels Pattern-Matching eine Funktion zu definieren, kann man mit Hilfe von `match` auch direkt die entsprechende Funktion definieren. Dazu muss man sechs Funktionen - eine für jeden Konstruktor der `Try`-Struktur (`Val`, `Choice`, `Fail` etc.) - mit der gewünschten Funktionalität an `match` übergeben. Sowohl `match` als auch `try` sind für verschiedene Typen überladen.

- Die Funktionen `choiceCons`, `failCons`, `guardCons` etc. sind überladene Konstruktorfunktionen. Das heißt, abhängig vom gerade erforderlichen Typ liefert beispielsweise der Aufruf von `failCons` den Wert `Fail_C_Int`, `Fail_C_Bool` etc.

Die Funktion `searchFDCs` hat zwei Argumente: einen normalisierten, nicht-deterministischen Ausdruck und eine anfangs leere Liste zum Einsammeln der FD-Constraints. Ziel ist es, alle `Guard`-Ausdrücke mit FD-Constraints aus dem gegebenen nicht-deterministischen Ausdruck zu entfernen, die Constraints in der Liste zu sammeln und eine einzelne `Guard` für sie zu erzeugen. Man implementiert `searchFDCs` mit der Funktion `match` durch Angabe der Hilfsfunktionen:

- `sfChoice`: Einsammeln der FD-Constraints in beiden Zweigen durch rekursiven Aufruf von `searchFDCs`. Die `Choice` wird über den resultierenden Teilbäumen erneut konstruiert.

- `sfNarrowed`: Rekursiver Aufruf von `searchFDCs` zum Einsammeln der FD-Constraints in allen Zweigen. Die n-äre `Choice` wird über den resultierenden Teilbäumen wieder aufgebaut.

- `sfGuard`: Hier unterscheidet man, ob es sich um einen `Guard`-Ausdruck mit einer Liste von `FDConstraint`s handelt oder nicht. Falls ja, so wird die entsprechende Constraint-Liste an die Liste zum Sammeln aller FD-Constraints (`fdCs`) angehängt. Danach wird der aktuelle Zweig durch rekursiven Aufruf nach weiteren Constraints durchsucht, wobei der `Guard`-Knoten nicht wieder aufgebaut und somit aus dem Suchbaum entfernt wird.
  Enthält der `Guard`-Ausdruck hingegen keine FD-Constraints, so wird zwar auch der aktuelle Zweig weiter durchsucht, allerdings wird die `Guard` in diesem Fall mit denselben Constraints über dem durch `(searchFDCs e fdCs)` gelieferten Teilbaum wieder aufgebaut.

- `sfVal`: Erreicht man einen konstanten Wert `v`, so hat man einen Pfad im Suchbaum vollständig abgelaufen. Das heißt, es kann ein neuer `Guard`-Ausdruck erzeugt werden, der `v` mit den gesammelten FD-Constraints `fdCs` beschränkt.
  Falls keine FD-Constraints eingesammelt wurden, so bleibt der Wert `v` unbeschränkt.

Gelangt man beim Einsammeln der FD-Constraints mit `searchFDCs` in einem Ausdruck zu einem Knoten, der eine freie Variable oder eine fehlgeschlagene Berechnung repräsentiert, so bleibt dieser Knoten erhalten und die Suche nach FD-Constraints wird in diesem Zweig nicht weiter fortgesetzt. Daher wird in diesen Fällen die entsprechende Konstruktorfunktion aufgerufen.

Aufgerufen wird die Funktion `searchFDCs` unmittelbar nach der Normalisierung eines Ausdrucks durch die KiCS2-Funktion `getNormalForm`. Ein zu normalisierender Ausdruck wird durch eine Funktion vom Typ `NonDetExpr a` repräsentiert, die einen `IDSupply` und einen `ConstStore` als Argumente erwartet. Der `IDSupply` ist notwendig, falls der Ausdruck Nicht-Determinismus einführt und das `ConstStore`-Argument wird für eine Optimierung beim Lösen von Curry-Bindungs-Constraints benötigt. Nachdem der zu normalisierende Ausdruck mit einem frischen `IDSupply` `s` und einem leeren `ConstStore` `emptyCs` appliziert wurde, wird die Normalform durch Aufruf des `($!!)`-Operators aus der Typklasse `NormalForm` berechnet.
Anstatt den resultierenden Suchbaum nun direkt zurückzugeben, werden zuvor durch Aufruf von `searchFDCs` alle Finite-Domain-Constraints in einem einzelnen `Guard`-Ausdruck gesammelt:

```
type NonDetExpr a = IDSupply -> ConstStore -> a
-- Ergänzt um Aufruf von 'searchFDCs' zum Einsammeln
-- aller FD-Constraints
getNormalForm :: NormalForm a => NonDetExpr a -> IO a
getNormalForm goal = do
  s <- initSupply
  let normalForm = const $!! goal s emptyCs $ emptyCs
  return $ searchFDCs normalForm []
```

Listing 58: Erweiterte Normalform-Berechnung

Die beiden folgenden Zeichnungen verdeutlichen das Verhalten der Funktion `searchFDCs`:

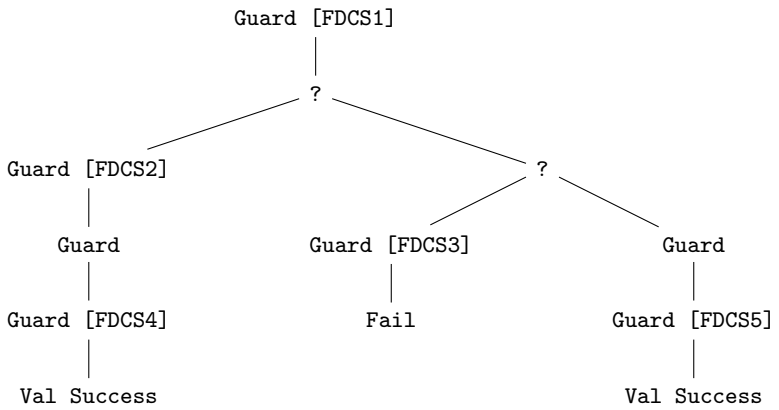


Abbildung 3: Suchbaum **vor** Aufruf von `searchFDCs`

Wie man sieht, enthält der obige Suchbaum fünf `Guard`-Ausdrücke mit Finite-Domain-Constraints. Beim Aufruf von `searchFDCs` werden die einzelnen Pfade durch den Suchbaum abgelaufen. Dabei werden alle `Guard`-Ausdrücke mit FD-Constraints entfernt und ihre Constraints eingesammelt. Die übrigen inneren Knoten wie Standard-`Guards` oder `Choices` (hier dargestellt durch `?`) bleiben unverändert. Erreicht die Funktion am Ende eines Pfades einen deterministischen Wert (`Val`-Knoten), so wird dieser durch Konstruktion eines neuen `Guard`-Ausdrucks mit den bis dahin gesammelten Constraints beschränkt. Beispielsweise werden beim Ablaufen des linken Pfades im obigen Suchbaum nacheinander die Constraints `FDCS1`, `FDCS2` und `FDCS4` gesammelt und in einem neuen `Guard`-Ausdruck zusammengefasst. Endet der Pfad hingegen mit einer fehlgeschlagenen Berechnung, so werden die gesammelten Constraints einfach verworfen und der `Fail`-Knoten bleibt erhalten (Vergleiche mittlerer Pfad im Beispiel-Suchbaum).

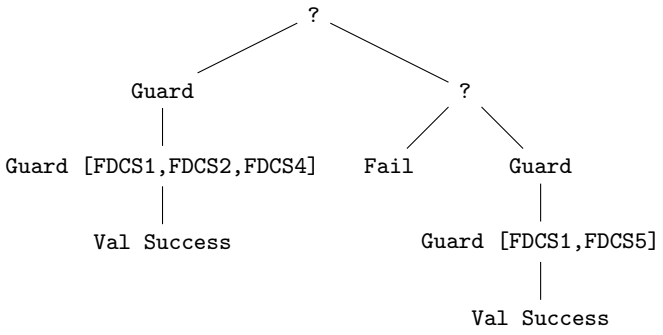


Abbildung 4: Suchbaum **nach** Aufruf von `searchFDCs`

4.1.4. Kombination von Finite-Domain-Constraints mit der Curry-Unifikation

Nun, da man ein in Curry beschriebenes FD-Constraint-Modell auf einen einzelnen `Guard`-Ausdruck mit einer Haskell-Repräsentation dieses Modells abbilden kann, ist man prinzipiell soweit, einen konkreten Constraint-Solver in KiCS2 einzubinden. Zuvor soll aber noch auf die Möglichkeit eingegangen werden, die Finite-Domain-Constraints aus der CLPFD-Bibliothek mit den Curry-Gleichheits-Constraints bei der Modellierung eines Problems zu kombinieren.

KiCS2 stellt in seiner bisherigen Form bereits ein Constraint zur Verfügung, nämlich ein Gleichheits-Constraint der Form `(=:=) :: a -> a -> Success` (auch als Unifikationsoperator bezeichnet). Dabei ist ein Constraint-Ausdruck der Form `e1 =:= e2` genau dann erfüllbar, wenn `e1` und `e2` zu unifizierbaren Konstruktortermen reduzierbar sind. Ausdrücke, die keinen Wert haben wie `head []`, sind bezüglich `(=:=)` **nicht** gleich.

Die Frage ist nun, in welchen Fällen es sinnvoll ist, dieses Gleichheits-Constraint bei der Modellierung eines Problems mit den Finite-Domain-Constraints aus der neu eingeführten CLPFD-Bibliothek zu kombinieren. Zur Klärung dieser Frage betrachtet man erneut das N-Damen-Problem. Bislang wurde für die einzelnen Instanzen des N-Damen-Problems stets ein eigenes Modell definiert. Es wäre jedoch wünschenswert, dass man ein Modell findet, mit dem man durch

Übergabe der Problemgröße als Parameter beliebige Instanzen beschreiben kann.
Dafür ist es erforderlich, dass man zur Laufzeit die notwendigen Constraint-Variablen und Constraints dynamisch in Abhängigkeit von der übergebenen Problemgröße erzeugen kann. Bei der Formulierung des 4-Damen-Problems wurden bereits Constraint-Funktionen verwendet, die abhängig von der Anzahl der Constraint-Variablen die erforderlichen Constraints generiert haben (`all_safe`, `safe` und `no_attack`). Somit benötigt man nur noch eine Funktion, mit der man eine entsprechende Liste von Constraint-Variablen erzeugen kann. Eine solche stellt die CLPFD-Bibliothek aber bereits in Form der Funktion `genVars :: Int -> [Int]` zur Verfügung.
Ein Problem verbleibt jedoch noch: Die Liste der Constraint-Variablen zur Repräsentation der N Damen wird als Parameter verschiedener Constraint-Funktionen verwendet. Man kann jedoch nicht überall, wo ein solcher Parameter verlangt wird, `genVars` aufrufen, da diese Funktion jedes Mal eine neue Liste von Constraint-Variablen erzeugt, was dazu führen würde, dass die Constraints über unterschiedlichen Variablen definiert würden.
Lösen lässt sich dieses Problem durch Verwendung des Curry-Gleichheits-Constraints: Dazu erzeugt man einmalig mit `genVars` eine Liste mit der gewünschten Anzahl von Constraint-Variablen und bindet diese dann mit Hilfe von `(=:=)` an eine freie (Listen-)Variable. Diese freie Variable repräsentiert dann bei jeder Verwendung die gleiche Liste von Constraint-Variablen und kann somit als Parameter in den FD-Constraint-Funktionen des Modells benutzt werden:

```
queens n l =
       genVars n =:= l &
       domain l 1 n &
       all_safe l &
       labeling l

all_safe [] = success
all_safe (q:qs) = safe q qs 1 & all_safe qs

safe :: Int -> [Int] -> Int -> Success
safe _ [] _ = success
safe q (q1:qs) p = no_attack q q1 p & safe q qs (p+#1)

no_attack q1 q2 p = q1 /=# q2 & q1 /=# q2+#p & q1 /=# q2-#p
```

Listing 59: Beispiel: N-Damen-Problem

Die freie Variable `l`, an die die Constraint-Variablen gebunden werden, übergibt man zusammen mit der Problemgröße `n` bei Aufruf einer Instanz des N-Damen-Problems. Auf diese Weise werden einem bei einem Aufruf wie `queens 4 l where l free` alle Lösungen in Form der verschiedenen Bindungen für `l` angezeigt.

Was genau passiert nun aber, wenn man einer Constraint-Funktion eine freie (Listen-)Variable als Parameter übergibt? Bei der Implementierungsbeschreibung der Constraint-Funktionen der

CLPFD-Bibliothek wurde erklärt, dass Listen-Argumente wie Integer-Argumente zur Normalform ausgewertet werden, bevor sie an die jeweilige Constraint-Funktion übergeben werden. Zuvor werden sie allerdings noch in die sogenannte *spine*-Form überführt. Dabei wird sichergestellt, dass die übergebene Liste endlich ist. Zudem wird durch Aufruf von `ensureNotFree` garantiert, dass die Liste und auch keine ihrer Teillisten durch eine freie Variable repräsentiert wird. Falls doch eine freie Variable vorkommt, so wird die sie repräsentierende `Choice` in KiCS2 mittels Narrowing in eine n-äre Standard-`Choice` umgewandelt.

```
domain :: [Int] -> Int -> Int -> Success
domain vs l u = ((prim_domain $!! (ensureSpine vs)) $!! l) $!! u

ensureSpine :: [a] -> [a]
ensureSpine l = ensureList (ensureNotFree l)
where ensureList []     = []
      ensureList (x:xs) = x : ensureSpine xs
```

Listing 60: Wiederholung: Auswertung von Listen-Argumenten in Constraint-Funktionen

Die nachfolgende Berechnung der Normalform durch den `($!!)`-Operator sorgt dafür, dass die durch das Narrowing neu eingeführten Standard-`Choices` auf die Ergebnisebene propagiert werden, so dass die extern implementierten Constraint-Funktionen wie `prim_domain` letztendlich wieder auf normalisierte Listen-Argumente angewendet werden. Somit können auch freie Variablen als Listen-Argumente für die Constraint-Funktionen der CLPFD-Bibliothek verwendet werden, ohne dass die bisherige Implementierung angepasst werden muss.

Ein Problem gibt es bei der Verwendung des Curry-Gleichheits-Constraints in FD-Constraint-Modellen aber dennoch: Wird eine einfache Constraint-Variable in einem Curry-Modell einmal als Teil einer Liste von Constraint-Variablen in einem `(=:=)`-Constraint und einmal in einem gewöhnlichen Finite-Domain-Constraint der CLPFD-Bibliothek verwendet, so kann es passieren, dass diese beiden Vorkommen der Curry-Variablen auf unterschiedliche FD-Variablen im Haskell-Modell abgebildet werden. Um dies zu verdeutlichen, soll ein weiteres Constraint-Problem betrachtet werden: das *send-more-money*-Rätsel.
Beim *send-more-money*-Rätsel ist es das Ziel, jeden Buchstaben mit einer Ziffer zwischen 0 und 9 zu belegen, so dass die folgende Gleichung erfüllt ist:

```
    S E N D
+   M O R E
= M O N E Y
```

Zusätzlich sollen keine zwei Buchstaben mit der gleichen Ziffer und die Buchstaben `S` und `M` mit einer Ziffer größer als Null belegt werden.

Mit Hilfe der CLPFD-Bibliothek lässt sich dieses Problem wie folgt modellieren:

```
smm l = [S,E,N,D,M,O,R,Y] =:= l &
        domain l 0 9 &
        S ># 0 &
        M ># 0 &
        allDifferent l &
                          1000 *# S +# 100 *# E +# 10 *# N +# D
        +#                1000 *# M +# 100 *# O +# 10 *# R +# E
        =# 10000 *# M +# 1000 *# O +# 100 *# N +# 10 *# E +# Y &
        labeling l
        where S,E,N,D,M,O,R,Y free
```

Listing 61: Beispiel: SEND-MORE-MONEY (Curry)

Das Curry-Modell bildet das oben spezifizierte Problem exakt ab: Die acht verschiedenen Buchstaben aus dem Rätsel werden durch entsprechend benannte freie Variablen repräsentiert. Als Wertebereich werden die Ziffern von 0 bis 9 festgelegt, wobei die 0 für die Buchstaben `S` und `M` zusätzlich ausgeschlossen wird. Des Weiteren wird verlangt, dass die Belegungen der Variablen alle paarweise verschieden sind. Die Summengleichung kann direkt übertragen werden. Es muss nur die Stelligkeit der einzelnen Buchstaben/Ziffern berücksichtigt werden.
Wie auch schon bei der Modellierung des N-Damen-Problems bindet man die Liste der Constraint-Variablen im Curry-Modell an eine freie Variable, die man in den entsprechenden Constraint-Funktionen als Parameter übergibt.

Zur Erkennung des Problems, das durch die Kombination des Curry-Gleichheits-Constraints mit den Finite-Domain-Constraints in dem obigen Beispielmodell entsteht, muss man sich dessen zugehörige Haskell-Repräsentation ansehen. Zunächst betrachtet man, die Haskell-Darstellung des Ausdrucks `[S,E,N,D,M,O,R,Y] =:= l`.

KiCS2 bildet ein solches Gleichheits-Constraint auf eine `Guard` mit einem `ValConstr`-Constraint ab. Mit diesem `ValConstr`-Constraint wird ein Wert - hier die Liste `[S,E,N,D,M,O,R,Y]` - an eine bestimmte `ID` - in diesem Fall die `ID` der freien Variable `l` - gebunden. Zusätzlich enthält es eine Liste von Bindungs-Constraints der Form `ID :=: Decision` (vergleiche Kapitel 2.2.2), die gelöst werden müssen, um diese Bindung durchzuführen.
Zur Wiederholung: Mögliche Bindungsentscheidungen sind unter anderem die Auswahl eines bestimmten Konstruktors (`ChooseN`) oder die Bindung an eine andere Variable (`BindTo`).
Die Liste mit den Bindungs-Constraints enthält einerseits Entscheidungen für die Teillisten von `l`, wobei entweder der Cons-Konstruktor (`ChooseN 1 2`) oder der Konstruktor für die leeren Liste (`ChooseN 0 0`) ausgewählt werden kann. Andererseits werden in der gleichen Liste aber auch Bindungsentscheidungen für die Listenelemente von `l` getroffen.
In der unten angeführten Repräsentation von `[S,E,N,D,M,O,R,Y] =:= l` wird der durch KiCS2 erzeugte `Guard`-Ausdruck etwas vereinfacht dargestellt, da nur die Liste der Bindungs-Constraints

aufgeführt wird. Auf die anderen Argumente des `ValConstr`-Constraints wird der besseren Lesbarkeit wegen verzichtet. Für die Darstellung der `ID`s werden fortlaufende Zahlen beginnend mit `1` verwendet:

```
Guard (ValConstr
  [1 :=: ChooseN 1 2,  2 :=: BindTo 3,     4 :=: ChooseN 1 2,
   5 :=: BindTo 6,     7 :=: ChooseN 1 2,  8 :=: BindTo 9,
  10 :=: ChooseN 1 2, 11 :=: BindTo 12,   13 :=: ChooseN 1 2,
  14 :=: BindTo 15,   16 :=: ChooseN 1 2, 17 :=: BindTo 18,
  19 :=: ChooseN 1 2, 20 :=: BindTo 21,   22 :=: ChooseN 1 2,
  23 :=: BindTo 24,   25 :=: ChooseN 0 0]) C_Success
```

Listing 62: Repräsentation von $[S, E, N, D, M, O, R, Y] =:= l$ in Haskell

Die Teillisten von `l` werden durch die `ID`s `1`, `4`, `7`, `10`, `13`, `16`, `19`, `22` und `25` referenziert. Da `l` an eine Liste mit acht Elementen gebunden werden soll, wird für die ersten acht `ID`s der Cons-Konstruktor gewählt und für die letzte der Konstruktor der leeren Liste. Die Listenelemente - referenziert durch die `ID`s `2`, `5`, `8`, `11`, `14`, `17`, `20` und `23` - werden durch `BindTo`-Entscheidungen an die `ID`s der freien Variablen von `S`, `E`, `N`, `D`, `M`, `O`, `R` und `Y` gebunden. Die zugehörigen `ID`s sind `3`, `6`, `9`, `12`, `15`, `18`, `21` und `24`.

Betrachtet man nun zusätzlich die Haskell-Repräsentation der ersten paar Finite-Domain-Constraints aus dem obigen Modell, so stellt man fest, dass diese zum Teil über unterschiedlichen FD-Variablen definiert werden:

```
Guard (FDConstr
  [FDDomain [FDVar 2,FDVar 5,FDVar 8,FDVar 11,FDVar 14,
             FDVar 17, FDVar 20, FDVar 23] (Const 0) (Const 9),
   FDRel Less (Const 0) (FDVar 3),
   FDRel Less (Const 0) (FDVar 6), ... ]) C_Success
```

Listing 63: SEND-MORE-MONEY (Haskell) - Ausschnitt

Constraint-Funktionen wie `domain`, die eine durch das Curry-Gleichheits-Constraint gebundene Variable `l` als Argument erhalten, erzeugen einen `FDConstraint`-Konstruktorterm über den noch ungebundenen Listenelementen von `l`. Anstatt dass die FD-Darstellung von `[S,E,N,D,M,O,R,Y]`, nämlich `[FDVar 3,FDVar 6,FDVar 9,FDVar 12,FDVar 15,FDVar 18,FDVar 21,FDVar 24]`, verwendet wird, werden die Listenelemente von `l`, bei denen es sich um durch das Narrowing eingeführte freie Variablen handelt, in FD-Variablen übersetzt und über diesen das entsprechende `FDConstraint` konstruiert.
Dies liegt daran, dass zum Zeitpunkt der Konstruktion des Domain-Constraints in KiCS2 die Bindung(sentscheidung)en für die Listenvariable `l` und ihre Elemente noch nicht aufgelöst sind. Somit wird das Constraint über den "falschen" FD-Variablen erzeugt. Solange sich die Gleichheit, die durch das Gleichheits-Constraint in Curry zwischen den Listenelementen von `l` und der Liste

`[S,E,N,D,M,O,R,Y]` ausgedrückt wird, nicht auch im Haskell-Modell wiederfindet, sind die beiden Modelle nicht mehr semantisch äquivalent. Diese Äquivalenz muss wiederhergestellt werden, bevor ein Constraint-Solver auf das Modell angesetzt wird.

Die Wiederherstellung der semantischen Äquivalenz könnte man beispielsweise erreichen, indem man bei Aufruf von `(=:=)` neben dem oben vorgestellten `Guard`-Ausdruck mit dem `ValConstr` weitere `Guard`-Ausdrücke mit geeigneten `FDConstraint`s anlegt. Das Binden eines Integer-Wertes an eine freie Variable könnte man beispielsweise mit Hilfe des bereits bekannten `FDRel Equal`-Constraints ausdrücken.
Das Problem hierbei ist nur, dass der Operator `(=:=)` und auch die externe Funktion, durch die er implementiert wird, überladen sind. Das heißt, der Operator kann nicht nur zur Bindung von Integer-Werten, sondern für eine Vielzahl von Typen verwendet werden. Diese Gleichheit auf vielen verschiedenen Typen ist auch durch Einführung eines neuen Konstruktors für die Datenstruktur `FDConstraint` nicht ohne weiteres ausdrückbar und im Grunde genommen auch nicht gewollt. Schließlich bildet die Datenstruktur `FDConstraint` FD-Constraints über einer Integer-Domain in Haskell ab. Benutzt man die gleiche Datenstruktur nun auch noch zur Abbildung der Term-Gleichheit für beliebige Typen, so vermischt man verschiedene Domains miteinander.
Außerdem würde diese Lösung zu Redundanzen im Code führen. Schließlich wird das Curry-Gleichheits-Constraint in KiCS2 bereits auf die Bindungs-Constraints abgebildet, welche durch einen extra dafür implementierten Solver gelöst werden. Bildet man diese Gleichheit nun durch ein zusätzliches Constraint ab, so wird die gleiche Information zweimal durch die Implementierung gereicht und muss auch zweimal gelöst werden.

Daher wäre es sinnvoller, die bisherige Implementierung von KiCS2 auszunutzen. Dazu müssen die Bindungs-Constraints, die durch Verwendung des `(=:=)`-Operators KiCS2-intern eingeführt werden, vor den Finite-Domain-Constraints gelöst werden. Anschließend muss man eine Möglichkeit finden, die Variablen im FD-Modell nachträglich gemäß den (durch den Solver bestätigten) gültigen Bindungsentscheidungen zu aktualisieren. Dazu soll zunächst die Erzeugung der Bindungs-Constraints sowie ihre Lösung durch den Solver näher betrachtet werden.
Bindungs-Constraints der Form `ID :=: Decision` werden durch die überladene Funktion `bind` erzeugt. Alle unifizierbaren Typen implementieren diese Funktion in ihrer `Unifiable`-Instanz. Beispielsweise ist `bind` auf dem Typ `C_Bool` folgendermaßen definiert:

```
bind :: Unifiable a => ID -> a -> [Constraint]
bind i C_False                          = [i :=: (ChooseN 0 0)]
bind i C_True                           = [i :=: (ChooseN 1 0)]
bind i (Choices_C_Bool (FreeID j) xs) = [i :=: BindTo j]
...
```

Listing 64: `bind`-Implementierung für `C_Bool`

Bindet man eine `ID` an einen booleschen Konstruktor, so wird ein Bindungs-Constraint mit einer passenden `ChooseN`-Bindungsentscheidung erzeugt. Bei der Bindung an eine freie Variable

wird hingegen ein Bindungs-Constraint mit einer `BindTo`-Entscheidung generiert. Ein Bindungs-Constraint legt immer nur die Entscheidung für einen Konstruktor fest. Bei strukturierten Datentypen wie Listen können jedoch durch eine Liste von Bindungs-Constraints Bindungsentscheidungen für Teillisten sowie für die einzelnen Listenelemente getroffen werden. Dies verdeutlicht die Implementierung von `bind` für den Listentyp `OP_List a`:

```
bind i OP_List = [i :=: (ChooseN 0 0)]
bind i (OP_Cons x2 x3) =
  ((i :=: (ChooseN 1 2)):(concat [(bind (leftID i) x2)
                                  ,(bind (rightID i) x3)]))
bind i (Choices_OP_List (FreeID j) xs) = [i :=: BindTo j]
...
```

Listing 65: `bind`-Implementierung für `OP_List a`

Die Bindung einer leeren Liste (`OP_List`) oder einer freien Variable wird nach dem bereits bekannten Muster implementiert. Falls jedoch eine nicht-leere Liste gebunden werden soll, so wird zunächst ein Bindungs-Constraint mit einer passenden `ChooseN`-Bindungsentscheidung für den `OP_Cons`-Konstruktor generiert. Danach werden die Bindungs-Constraints für das Listenelement `x2` sowie für die Restliste `x3` durch Aufruf der entsprechenden `bind`-Instanzen erzeugt. Neue `ID`s für die Bindungs-Constraints des Listenelements bzw. der Restliste werden mit Hilfe der Funktionen `leftID` bzw. `rightID` bereitgestellt. Diese Hilfsfunktionen wurden mit den im KiCS2-Grundlagenkapitel vorgestellten Funktionen `leftSupply` bzw. `rightSupply` implementiert.
Alle erzeugten Bindungs-Constraints werden schließlich in einer Liste zusammengefasst.

Beim Lösen solcher Bindungs-Constraints speichert der Solver die für eine `Choice` (identifiziert durch die `ID` im Bindungs-Constraint) getroffene Bindungsentscheidung in einem globalen Decision-Store ab. Realisiert wird dieser Store durch eine Map, wobei die `ID` der jeweiligen `Choice` als Schlüssel fungiert. Zum Nachschlagen oder Eintragen einer Bindungsentscheidung für eine bestimmte `ID` stellt KiCS2 die Funktionen `lookupDecision` bzw. `setDecision` zur Verfügung.
Das heißt, man kann getroffene Bindungsentscheidung für eine gegebene `ID` und damit auch für eine FD-Variable nachschlagen. Das Problem ist nur, dass man eine Bindungsentscheidung vom Typ `Decision` zurückerhält und nicht den Wert, an den die FD-Variable gebunden wurde.
Die Idee ist nun, dass man eine Funktion `lookupValue` definiert, die für eine gegebene `ID` durch Aufruf von `lookupDecision` die getroffene Bindungsentscheidung bestimmt und diese dann durch eine weitere Funktion namens `fromDecision` in ihren ursprünglichen Wert umwandelt.
Die Funktion `fromDecision` soll dabei in etwa das Verhalten der oben vorgestellten `bind`-Funktion umkehren: So soll für eine gegebene Bindungsentscheidung entweder der passende Konstruktor oder die freie Variable des jeweiligen Typs rekonstruiert werden. Da ausgehend von einer Bindungsentscheidung die Werte verschiedener Typen rekonstruierbar sein sollen, muss auch `fromDecision` überladen werden. Man definiert daher die folgende Typklasse:

```
class Generable a => FromDecisionTo a where
  fromDecision :: Store m => ID -> (Decision,ID) -> m a
```

Listing 66: Typklasse `FromDecisionTo`

Für die Rekonstruktion freier Variablen ist es nötig, dass die Typen der Typklasse `FromDecisionTo` auch die Typklasse `Generable` implementieren, die eine Generator-Funktion für freie Variablen bereitstellt. Des Weiteren wird bei der Implementierung von `fromDecision` auf den globalen Decision-Store zugegriffen, daher findet die Rekonstruktion von Werten in der Store-Monade statt. Bevor nun die Funktion `fromDecision` anhand von Beispielinstanzen erklärt wird, soll zunächst die Implementierung von `lookupValue` betrachtet werden:

```
lookupValue :: (Store m,FromDecisionTo a) => ID -> m a
lookupValue i = do decId <- lookupDecisionID i
                   fromDecision i decId
```

Listing 67: `lookupValue`-Funktion

Diese Funktion bestimmt für eine gegebene `ID` die im Decision-Store eingetragene Bindungsentscheidung und übersetzt diese mit Hilfe von `fromDecision` in den ursprünglichen Wert. Mit der Funktion `lookupDecisionID :: Store m => ID -> m (Decision,ID)` wird dazu für die gegebene `ID i` die Bindungsentscheidung sowie die letzte `ID`, an die `i` gebunden wurde, im Decision-Store nachgeschlagen.
Falls die übergebene `ID` an keine andere Variable gebunden wurde, so kann die hierbei ermittelte `ID` mit der ursprünglichen übereinstimmen. In diesem Fall liefert `lookupValue` die zur `ID i` zugehörige freie Variable vom Typ `a` zurück.
Zusammen mit der ursprünglichen `ID` wird das Tupel aus Bindungsentscheidung und letzter Bindungs-`ID` dann an die Funktion `fromDecision` übergeben. Diese übersetzt dann die ermittelten Bindungsentscheidungen in Konstruktoren zurück und rekonstruiert auf diese Weise den originalen Wert.

Man betrachtet nun zunächst die `FromDecisionTo`-Instanz für `C_Bool`:

```
instance FromDecisionTo C_Bool where
  fromDecision _ ((ChooseN 0 0),_) = return C_False
  fromDecision _ ((ChooseN 1 0),_) = return C_True
  fromDecision _ (NoDecision,j) = return (generate (supply j))
...
```

Listing 68: `FromDecisionTo`-Instanz für `C_Bool`

Wie man sieht, wird das erste Argument, die ursprüngliche `ID`, in allen Fällen ignoriert. Dieses Argument spielt nur bei der Rekonstruktion von Werten für strukturierte Datentypen eine Rolle. Weiter unten wird dies genauer erklärt.

In diesem Fall genügt das Tupel-Argument zur Rekonstruktion des ursprünglichen booleschen Wertes: `ChooseN`-Bindungsentscheidungen werden auf den Originalwertkonstruktor, also hier `C_False` bzw. `C_True`, zurückgeführt (vergleiche `bind`-Implementierung). Handelt es sich bei der Bindungsentscheidung aus dem Decision-Store hingegen um eine `NoDecision`-Entscheidung, so wurde die `ID` an eine (andere) `ID` und damit an eine andere freie Variable gebunden. Zur Wiederherstellung dieser freien Variablen bestimmt man mit `supply` den zur `ID` zugehörigen `IDSupply` und übergibt diesen dann an die `generate`-Funktion, die eine freie Variable vom Typ `C_Bool` zurückliefert.
Für alle anderen Bindungsentscheidungen wird bei Aufruf von `fromDecision` eine Fehlermeldung zurückgegeben.

Als Beispiel für einen strukturierten Datentyp wird nun noch die `FromDecisionTo`-Instanz für den Listentyp `OP_List a` betrachtet:

```
instance FromDecisionTo t0 => FromDecisionTo (OP_List t0) where
  fromDecision _ ((ChooseN 0 0),_) = return OP_List
  fromDecision i ((ChooseN 1 2),_) =
    do x3 <- lookupValue (leftID i)
       x4 <- lookupValue (rightID i)
       return (OP_Cons x3 x4)
  fromDecision _ (NoDecision,j) = return (generate (supply j))
...
```

Listing 69: `FromDecisionTo`-Instanz für `OP_List a`

Die Fälle zur Wiederherstellung einer leeren Liste (`ChooseN 0 0`) oder einer freien Variablen werden nach dem gleichen Schema wie oben implementiert. Interessant ist hingegen die Rekonstruktion einer nicht-leeren Liste (`ChooseN 1 2`): Hierbei kommt erstmals das erste Argument von `fromDecision` ins Spiel. Durch Anwendung der Funktionen `leftID` und `rightID` auf dieses `ID`-Argument können die `ID`s für den (äußersten) Konstruktor des Listenelements bzw. der Restliste bestimmt werden. Dann kann durch rekursiven Aufruf der `lookupValue`-Funktion über diesen `ID`s der ursprüngliche Wert für das Listenelement bzw. die Restliste rekonstruiert werden. Schließlich kann mit diesen Werten die originale Liste wieder aufgebaut und zurückgegeben werden.
Nach diesem Prinzip werden für alle strukturierten Datentypen die inneren Konstruktoren und damit auch die inneren Werte durch eine entsprechende Anzahl rekursiver Aufrufe von `lookupValue` bestimmt. Die nachfolgende Zeichnung verdeutlich nochmals den Ablauf bei der Rekonstruktion eines Wertes beginnend mit der Bindung dieses Wertes an eine freie Variable:

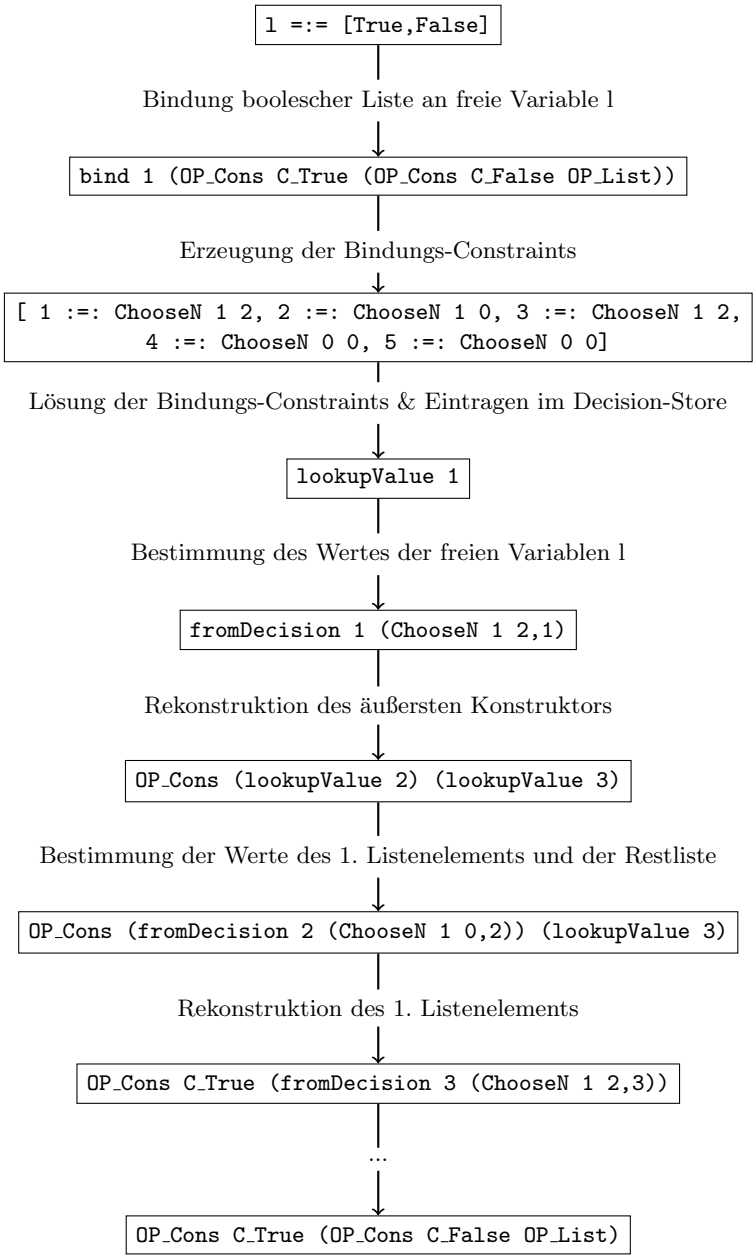


Abbildung 5: Ablauf der Bindung und Rekonstruktion eines Wertes

Zur Wiederherstellung der semantischen Äquivalenz zwischen dem Curry- und dem Haskell-Modell für das *send-more-money*-Rätsel muss man sich nun noch um zwei Dinge kümmern:

1. Man muss eine Funktion definieren, die die Bindungen aller FD-Variablen in allen Constraints eines Modells mit Hilfe von `lookupValue` aktualisiert.

2. Man muss dafür sorgen, dass Curry-Gleichheits-Constraints **vor** den Finite-Domain-Constraints gelöst werden, damit bei der Aktualisierung der FD-Variablen in einem Modell bereits alle Bindungsentscheidungen berechnet worden sind.

Für die Aktualisierung der FD-Variablen definiert man daher die folgende Funktion:

```
updateFDVar :: Store m => FDTerm Int -> m (FDTerm Int)
updateFDVar c@(Const _) = return c
updateFDVar (FDVar i)   = do x <- lookupValue i
                             return (toFDTerm x)
```

Listing 70: Funktion zur Aktualisierung von FD-Variablen

Die Funktion aktualisiert die Bindung einer FD-Variablen, indem sie mit `lookupValue` den aktuellen `C_Int`-Wert berechnet, an den die Variablen-`ID i` gebunden wurde. Dieser Wert wird dann wieder in einen FD-Term transformiert. Für den Fall, dass die FD-Variable nicht durch ein Curry-Gleichheits-Constraint gebunden wurde, wird die ursprüngliche FD-Variable wiederhergestellt. Konstante FD-Terme werden durch die Funktion nicht verändert.

Um nun die Bindungen aller FD-Variablen eines Haskell-Modells zu aktualisieren, definiert man die Funktion

```
updateFDConstr :: Store m => FDConstraint -> m FDConstraint
updateFDConstr (FDRel relOp t1 t2) =
  do t1' <- updateFDVar t1
     t2' <- updateFDVar t2
     return $ FDRel relOp t1' t2'
updateFDConstr (FDAllDifferent vs) =
  do vs' <- mapM updateFDVar vs
     return $ FDAllDifferent vs'
...
```

Listing 71: Aktualisierung von FD-Constraints

Die übrigen Fälle sind nach dem gleichen Schema definiert: Alle in einem `FDConstraint` vorkommenden FD-Variablen werden durch Aufruf von `updateFDVar` durch die FD-Term-Repräsentation ihrer aktuellen Bindung ersetzt.
Diese Funktion muss aufgerufen werden, **bevor** ein Finite-Domain-Solver auf das Constraint-Modell angesetzt wird.

Damit diese Aktualisierungen aber auch wirklich durchgeführt werden können, müssen die entsprechenden Bindungsentscheidungen zuvor in den Decision-Store eingetragen worden sein. Das bedeutet, die Bindungs-Constraints, die durch Aufruf von `(=:=)` KiCS2-intern erzeugt werden, müssen **vor** den Finite-Domain-Constraints gelöst werden.
Dazu ist es erforderlich, dass `Guard`-Ausdrücke mit Bindungs-Constraints noch vor Beginn der Auswertung im Suchbaum nach oben verschoben werden. Am einfachsten lässt sich dies erreichen, indem man neue Regeln für den in KiCS2 vordefinierten Konjunktionsoperator für Constraints

`(&) :: Success -> Success -> Success` definiert. Dieser Operator ist bislang wie folgt implementiert (aus Platzgründen werden im folgenden verkürzte Konstruktorbezeichnungen verwendet also z.B. `Fail` statt `Fail_C_Success`):

```
(&) :: C_Success -> C_Success -> ConstStore -> C_Success
(&) C_Success                    s _  = s
(&) x@Fail       _ _  = x
(&) (Guard c e)    s cs = Guard c ((e & s) cs)
(&) (Choice i a b) s cs = Choice i ((a & s) cs) ((b & s) cs)
(&) (Choices i xs) s cs =
      Choices (narrowID i) (map (\x -> (x & s) cs) xs)
```

Listing 72: Bisherige Realisierung von `(&)`

Wie man sieht, wird bei der bisherigen Implementierung ausschließlich auf das erste Argument des `(&)`-Operators gematcht. Das heißt, dass `Guard`-Ausdrücke mit Bindungs-Constraints, die im zweiten Argument des Konjunktionsoperators vorkommen, nicht nach oben propagiert werden. Dies ist jedoch notwendig, damit die Bindungs-Constraints `vor` den Finite-Domain-Constraints gelöst werden und die Bindungen der FD-Variablen vor Aufruf des Solvers aktualisiert werden können.
Daher definiert man eine Hilfsfunktion `maySwitch`, die die Argumente des `(&)`-Operators vertauscht, falls das erste Argument kein `C_Success`, `Fail` bzw. entsprechender `Guard`-Ausdruck ist. Kommen im zweiten Argument `Guard`s mit Bindungs-Constraints vor, so werden diese nach oben propagiert. Erst danach wird auf die übrigen Konstruktoren im ersten Argument gematcht (darunter auch die `Guard`-Ausdrücke mit den FD-Constraints). Diese werden nach den gleichen Regeln verknüpft wie in der ursprünglichen Implementierung:

```
(&) :: C_Success -> C_Success -> ConstStore -> C_Success
(&) C_Success                       s _  = s
(&) x@Fail                          _ _  = x
(&) (Guard c@(ValConstr _ _ _) e) s cs = Guard c ((e & s) cs)
(&) (Guard c@(StructConstr _) e)  s cs = Guard c ((e & s) cs)
(&) x                               y cs = maySwitch y x cs

maySwitch :: C_Success -> C_Success -> ConstStore -> C_Success
maySwitch C_Success              x                  _  = x
maySwitch y@Fail                 _  = y
maySwitch (Guard c@(StructConstr _) e) x  cs =
  Guard c ((x & e) cs)
maySwitch (Guard c@(ValConstr _ _ _) e) x cs =
  Guard c ((x & e) cs)
maySwitch y (Choice i a b) cs =
  Choice i ((a & y) cs) ((b & y) cs)
```

```
maySwitch y (Choices i xs) cs =
  Choices (narrowID i) (map (\x -> (x & y) cs) xs)
maySwitch y (Guard c e)    cs = Guard c ((e & y) cs)
maySwitch (Guard c e) x    cs = Guard c ((x & e) cs)
maySwitch y x              _  =
  error $ "maySwitch: " ++ show y ++ " " ++ show x
```

Listing 73: Angepasste Realisierung von `(&)`

Durch die Verwendung dieser neuen Implementierung werden sämtliche `Guard`-Ausdrücke mit Bindungs-Constraints im resultierenden Suchbaum nach oben propagiert. Auf diese Weise ist sichergestellt, dass die Bindungs-Constraints bei der Auswertung des Suchbaums vor etwaigen FD-Constraints gelöst werden. Somit können die Bindungen der FD-Variablen eines Constraint-Modells aktualisiert werden, bevor der Solver darauf angesetzt wird.
Das heißt, mit der in diesem Kapitel vorgestellten Erweiterung können die Constraints der CLPFD-Bibliothek nun mit dem Gleichheits-Constraint `(=:=)` kombiniert werden, ohne dass das weiter oben erläuterte Problem (siehe *send-more-money*-Rätsel) auftritt.

4.2. Anschluss der FD-Constraint-Solver des MCP-Frameworks

Mit der bislang beschriebenen Erweiterung der KiCS2-Curry-Implementierung werden in Curry modellierte Finite-Domain-Probleme auf semantisch äquivalente Modelle in Haskell abgebildet. Diese Haskell-Modelle sollen nun durch Integration geeigneter Constraint-Solver in KiCS2 gelöst werden.
Constraint-Solver sind üblicherweise über einer festen Constraint-Sprache definiert. Diese Modellierungssprache gibt vor, welche Constraints der Solver interpretieren und lösen kann. Um nun einen Finite-Domain-Solver an KiCS2 anzuschließen, müssen die Haskell-Modelle vom Typ `[FDConstraint]` daher in die Modellierungssprache des Solvers übersetzt werden. Dann kann der Solver auf das übersetzte Modell angesetzt werden und nach Lösungen suchen. Abschließend müssen die vom FD-Solver gefundenen Lösungen in irgendeiner Form wieder in den Auswertungssuchbaum von KiCS2 eingebunden werden, damit sie auf der Kommandozeile ausgegeben werden können.

Dieses Kapitel beschreibt die Integration eines FD-Solvers in KiCS2 beispielhaft anhand des Anschlusses der FD-Solver des Monadic-Constraint-Programming-Frameworks. Da das Framework bereits im Grundlagenkapitel vorgestellt wurde, konzentriert sich dieses Kapitel auf die Integration der Solver über die oben beschriebene Finite-Domain-Schnittstelle. Für eine ausführlichere Beschreibung der Implementierung des MCP-Frameworks sei an dieser Stelle erneut auf die Artikel [9] und [10] verwiesen.
Das Kapitel ist in drei Abschnitte unterteilt: Der erste Abschnitt beschreibt die Übersetzung von der KiCS2-internen Darstellung von Finite-Domain-Constraints in semantisch äquivalente MCP-Constraints. Im zweiten Abschnitt wird dann darauf eingegangen, wie man aus den übersetzten Constraints ein MCP-(Baum-)Modell konstruiert, wie das Labeling der Constraint-Variablen funktioniert und wie man die FD-Solver des Frameworks aufruft. Der letzte Abschnitt befasst sich schließlich mit der Ausgabe der gefundenen Lösungen durch KiCS2.

4.2.1. Übersetzung der Constraints in die MCP-Modellierungssprache

Constraint-Modelle des Monadic-Constraint-Programming-Frameworks entsprechen Bäumen mit verschiedenen Knotenarten. Die Solver des Frameworks lösen ein solches Modell, indem sie Pfade durch den Baum Knoten für Knoten gemäß einer zuvor festgelegten Suchstrategie verfolgen und je nach Knotenart unterschiedliche Solver-Aktionen durchführen. Zu diesen Solver-Aktionen gehören beispielsweise das Hinzufügen eines Constraints zu ihrem Constraint-Speicher oder die Generierung einer neuen Constraint-Variablen.

Übersetzung von FD-Termen und Listen von FD-Termen:
Im Grundlagenkapitel über das MCP-Framework wurde dessen Finite-Domain-Schnittstelle bereits beschrieben. Diese bietet eine Reihe von Constraint-Funktionen mit zwei unterschiedlichen Implementierungen (in zwei separaten Modulen): Die erste Implementierung erzeugt nur ein zur

Funktion passendes MCP-FD-Constraint. Die zweite fügt dieses Constraint zusätzlich mit Hilfe eines `Add`-Knotens zu einem trivialen Baummodell hinzu.

Anstatt die KiCS2-interne Repräsentation eines FD-Modells direkt in ein MCP-Baummodell zu übersetzen, konstruiert man erst passende MCP-Constraints für jedes `FDConstraint` und baut dann mit diesen erst unmittelbar vor Aufruf des Solvers ein Baummodell auf. Die direkte Überführung in ein monadisches Baummodell würde insbesondere die Übersetzung von Constraint-Variablen erschweren, da im Baummodell eine Collection neuer Variablen einfach durch Aufruf von `exists` erzeugt wird. Man müsste in diesem Fall das `FDConstraint`-Modell vor der Übersetzung analysieren, um mit einem einmaligen Aufruf von `exists` die benötigte Variablenanzahl zu erzeugen. Dann müsste man jeder Variablen aus dem `FDConstraint`-Modell eine der erzeugten Variablen in der Collection zuordnen. Das Problem dabei ist, dass die Variablen erst bei der Auswertung durch den Solver generiert werden. Das heißt, um eine bestimmte Variable zu adressieren, müsste man bei der Modellierung auf das jeweilige Feld in der Collection zugreifen.
Einfacher ist es, die `FDConstraint`-Konstruktorterme zunächst in semantisch äquivalente `Model`-Konstruktorterme zu übersetzen und für die vorkommenden Constraint-Variablen "per Hand" eine entsprechende MCP-Repräsentation zu konstruieren. Auf diese Weise ist eine vorherige Analyse des `FDConstraint`-Modells nicht nötig und man spart sich die umständliche Adressierung der Variablen über die Felder der Collection.

Stattdessen ist es jedoch erforderlich, sich zu merken, welche Constraint-Variablen bereits übersetzt worden sind, um mehrfach vorkommende Variablen immer auf die gleiche MCP-Variable abzubilden. Um dies zu gewährleisten, führt man den folgenden Datentyp zur Repräsentation des Übersetzungszustands ein:

```
type IntVarMap = Map.Map Integer ModelInt

data TLState = TLState {
  intVarMap     :: IntVarMap,
  nextIntVarRef :: Int
}
```

Listing 74: Translation State

Ein `TLState` speichert eine Tabelle `intVarMap`, die einem `Integer`-Schlüssel einen Wert vom Typ `ModelInt` zuordnet, sowie die Referenz für die nächste zu erzeugende MCP-Integer-Variable `nextIntVarRef`. Zur Erinnerung: Im MCP-Framework werden Integer-Variablen durch den Datentyp `data ModelIntTerm = ModelIntVar Int` repräsentiert und der Typ für Integer-Ausdrücke ist `type ModelInt = Expr ModelIntTerm ModelColTerm ModelBoolTerm`. Eine einfache Integer-Variable vom Typ `ModelInt` mit der Variablenreferenz `0` hat also die Form `Term (ModelIntVar 0)`. Mit Hilfe dieses Zustands lässt sich nun eine Funktion zur Übersetzung von FD-Termen vom Typ `FDTerm Int` in einen MCP-Integer-Ausdruck vom Typ `ModelInt` angeben. Um den `TLState` nicht

immer explizit durchreichen zu müssen, wird Haskells `State`-Monade für die Implementierung der Übersetzungsfunktionen verwendet:

```
cte :: Integral a => a -> ModelInt

translateTerm :: FDTerm Int -> State TLState ModelInt
translateTerm (Const x)   = return (cte x)
translateTerm v@(FDVar i) = do $
  state <- get
  let varMap = intVarMap state
  maybe (newVar v) return (Map.lookup (getKey i) varMap)
```

Listing 75: Übersetzung von FD-Termen

Ist der zu übersetzende FD-Term eine Konstante, so wird der konstante Integer-Wert mit Hilfe der Schnittstellenfunktion `cte` in einen MCP-Integer-Ausdruck überführt.
Bei der Übersetzung einer FD-Variablen betrachtet man hingegen zunächst die Tabelle der bislang übersetzten Variablen im aktuellen Zustand. Diese Tabelle durchsucht man unter Verwendung der `Integer`-Repräsentation der FD-Variablen-`ID`, die man durch die vordefinierte KiCS2-Funktion `getKey` erhält. Gibt es für die FD-Variable einen Eintrag - das heißt, die Variable wurde bereits übersetzt - so gibt man diese Übersetzung zurück. Anderfalls wird die Funktion `newVar` aufgerufen, die eine neue MCP-Integer-Variable erzeugt.

```
newVar :: Term Int -> State TLState ModelInt
newVar (Var i) = do $
  state <- get
  let varMap   = intVarMap state
      varRef   = nextIntVarRef state
      nvar     = asExpr (ModelIntVar varRef)
      newState = state
        { nextIntVarRef = varRef + 1
        , intVarMap = Map.insert (getKey i) nvar varMap
        }
  put newState
  return nvar
```

Listing 76: Erzeugung einer MCP-Integer-Variable

Diese Funktion konstruiert mit Hilfe der im `TLState` gespeicherten nächsten Variablenreferenz eine neue MCP-Integer-Variable `nvar`, wobei `asExpr` den zunächst erzeugten `ModelIntTerm` in einen Integer-Ausdruck vom Typ `ModelInt` umwandelt. Für die übersetzte Variable wird dann ein neuer Eintrag in der Übersetzungstabelle angelegt. Außerdem wird der Zähler für die nächste Variablenreferenz inkrementiert. Mit den aktualisierten Werten wird schließlich ein neuer `TLState` konstruiert, der den alten ersetzt, und die generierte MCP-Variable wird zurückgegeben.

Listen von FD-Termen sollen in MCP-Collections vom Typ `ModelCol` übersetzt werden. Dazu definiert man eine Funktion `translateList`:

```
translateList :: [FDTerm Int] -> State TLState ModelCol
translateList vs = do mcpExprList <- mapM translateTerm vs
                      return (list mcpExprList)
```

Listing 77: Übersetzung von Listen von FD-Termen

Mit der zuvor definierten Funktion `translateTerm` werden bei Aufruf von `translateList` alle FD-Terme der Liste in Integer-Ausdrücke vom Typ `ModelInt` übersetzt. Die MCP-Funktion `list` fasst diese Liste von `ModelInts` in einer Collection vom Typ `ModelCol` zusammen.

Übersetzung der FD-Constraints:
Nun, da man in der Lage ist, Argumente der `FDConstraints` in entsprechende Repräsentationen der MCP-Modellierungssprache zu überführen, kann man die `FDConstraints` selbst übersetzen:

```
translateConstr :: FDConstraint -> State TLState Model
translateConstr (FDRel relop t1 t2) = do $
  mcpTerm1 <- translateTerm t1
  mcpTerm2 <- translateTerm t2
  let mcpRelop = translateRelOp relop
  return $ mcpRelop mcpTerm1 mcpTerm2

translateRelOp Equal     = (@=)
translateRelOp Diff      = (@/=)
translateRelOp Less      = (@<)
translateRelOp LessEqual = (@<=)

translateConstr (FDArith arithOp t1 t2 r) = do $
  mcpTerm1  <- translateTerm t1
  mcpTerm2  <- translateTerm t2
  mcpResult <- translateTerm r
  let mcpArithOp = translateArithOp arithOp
  return $ (mcpArithOp mcpTerm1 mcpTerm2) @= mcpResult

translateArithOp Plus  = (@+)
translateArithOp Minus = (@-)
translateArithOp Mult  = (@*)
```

Listing 78: Übersetzung der `FDConstraints` 1

Bei Relationalen und einfachen arithmetischen Constraints werden mit `translateTerm` die zwei bzw. drei beteiligten FD-Terme zunächst in MCP-Integer-Ausdrücke übersetzt. Die Funktionen

`translateRelOp` bzw. `translateArithOp` wählen dann die dem jeweiligen Operator entsprechende Constraint-Funktion aus der FD-Schnittstelle des Frameworks aus. Die gewählte Funktion wird auf die übersetzten Terme angewandt und erzeugt ein passendes MCP-Constraint vom Typ `Model`.

```
translateConstr (FDSum vs r) = do $
  mcpVs     <- translateList vs
  mcpResult <- translateTerm r
  return $ (xsum mcpVs) @= mcpResult

translateConstr (FDAllDifferent vs) = do $
  mcpVs <- translateList vs
  return $ allDiff mcpVs

translateConstr (FDDomain vs l u) = do $
  mcpVs <- translateList vs
  mcpL  <- translateTerm l
  mcpU  <- translateTerm u
  let domain varList lower upper =
    forall varList (\var -> var @: (lower,upper))
  return $ domain mcpVs mcpL mcpU
```

Listing 79: Übersetzung der `FDConstraints` 2

Die Constraints `FDSum` und `FDAllDifferent` werden nach dem gleichen Prinzip übersetzt: Zunächst überführt man ihre Argumente mit `translateList` bzw. `translateTerm` in MCP-Collections bzw. Integer-Ausdrücke. Dann ruft man die passende Constraint-Funktion des Frameworks auf - in diesem Fall `xsum` bzw. `allDiff` - um ein entsprechendes MCP-Constraint zu erzeugen.
Da das MCP-Framework kein Domain-Constraint bereitstellt, mit dem man den Wertebereich für eine Liste von FD-Variablen festlegen kann, muss man bei der Übersetzung von `FDDomain`-Constraints auf die Schnittstellenfunktion `forall` zurückgreifen. Wie im Grundlagenkapitel schon erwähnt, generiert die `forall`-Funktion ein Constraint, das nur gültig ist, wenn ein übergebenes Prädikat für alle Elemente der gegebenen Collection erfüllbar ist. Mit Hilfe der `forall`-Funktion definiert man nun selbst ein Domain-Constraint über MCP-Collections namens `domain`. Diese `domain`-Funktion erhält eine MCP-Collection sowie zwei Integer-Ausdrücke für die untere bzw. obere Grenze des Wertebereichs als Argumente und überprüft mit Hilfe von `forall` die Gültigkeit des Prädikats `(\var -> var @: (lower,upper))` für alle Elemente der Collection. Dieses Prädikat bindet jeweils ein Element der Collection mit Hilfe der MCP-Constraint-Funktion `(@:)` an den durch `lower` und `upper` gegebenen Wertebereich.
Um ein entsprechendes Constraint in der MCP-Modellierungssprache zu konstruieren, ruft man dann einfach `domain` mit den zuvor übersetzten Argumente des `FDDomain`-Constraints auf.

Das "`FDLabeling`-Constraint" stellt einen besonderen Fall dar: Labeling ist üblicherweise kein

Constraint sondern eine Technik, bei der ein FD-Solver die Constraint-Variablen eines Modells gemäß einer Labeling-Strategie mit Werten aus dem Wertebereich belegt und dann die Gültigkeit der übrigen Constraints unter Berücksichtigung dieser Belegung überprüft. Diese Technik wird zusammen mit der Propagierung von Constraints eingesetzt, um Lösungen zu finden.
Bei der Realisierung der CLPFD-Bibliothek wurde dennoch ein "Labeling-Constraint" eingeführt, das dazu dient, die für das Labeling relevanten Informationen durch die Implementierung bis zum Aufruf der Solver durchzureichen. Diese Informationen müssen nun zwischengespeichert werden, bis ein MCP-Solver aufgerufen wird. Dazu definiert man einen Datentyp namens `MCPLabelInfo`:

```
data MCPLabelInfo = Info {
  labelVars    :: Maybe [FDTerm Int],
  mcpLabelVars :: Maybe ModelCol,
  labelID      :: Maybe ID,
  strategy     :: Maybe LabelingStrategy
}
```

Listing 80: Datenstruktur zur Zwischenspeicherung von Labeling Informationen

Zwischengespeichert werden die FD-Variablen, über denen das Labeling durchgeführt werden soll, deren Repräsentation als MCP-Collection, eine unbenutzte `ID`, die später für die Ausgabe der durch den Solver gefundenen Lösungen benötigt wird, sowie die gewählte Labeling-Strategie.
Dieses Objekt zur Sicherung der Labeling-Informationen wird in den `TLState` aufgenommen, damit man bei der "Übersetzung" eines "`FDLabeling`-Constraints" darauf zugreifen kann:

```
data TLState = TLState {
  ...
  labelInfo :: MCPLabelInfo
}
```

Listing 81: Translation State (angepasst)

Mit diesen Hilfsmitteln kann man die Labeling-Informationen während der Übersetzung sichern:

```
translateConstr (FDLabeling str vs j) = do $
  mcpVs <- translateList vs
  state <- get
  let newInfo  = Info (Just vs) (Just mcpVs) (Just j) (Just str)
      newState = state { labelInfo = newInfo }
  put newState
  return (toBoolExpr True)
```

Listing 82: Übersetzung der `FDConstraints` 3

Zunächst wird die Liste der Labeling-Variablen übersetzt. Dann konstruiert man ein `MCPLabelInfo`-Objekt mit allen nötigen Informationen und aktualisiert damit den `TLState`. Da die Funktion

`translateConstr` ein FD-Constraint vom Typ `Model` eingekapselt in der Monade `State TLState` zurückgeben muss, erzeugt man mit dem Aufruf `toBoolExpr True` ein immer erfüllbares MCP-FD-Constraint als entsprechenden Rückgabewert.

Mit den vorgestellten Funktionen kann man die eigentliche Übersetzungsfunktion angeben, die ein KiCS2-internes Constraint-Modell vom Typ `[FDConstraint]` in eine Liste von Constraints der MCP-Modellierungssprache (vom Typ `[Model]`) übersetzt:

```
baseTLState :: TLState
baseTLState = TLState {
  intVarMap     = Map.empty,
  nextIntVarRef = 0,
  labelInfo     = baseLabelInfo }

translateToMCP :: [FDConstraint] -> ([Model],MCPLabelInfo)
translateToMCP fdCs =
  let (mcpCs,state) =
        runState (mapM translateConstr fdCs) baseTLState
      info = labelInfo state
  in (mcpCs, info)
```

Listing 83: Übersetzungsfunktion

Diese Funktion übersetzt alle Constraints der Liste mit `translateConstr` in MCP-Constraints. Dazu wird die `State`-Monade mit dem Startzustand `baseTLState` ausgeführt und die übersetzten Constraints sowie die gesammelten Labeling-Informationen werden zurückgegeben.

Das folgende Listing zeigt das übersetzte MCP-Modell des 2-Damen-Problems (aus Listing 54):

```
[ BoolAnd
   (BoolAnd
     (Rel (Const 1) ERLess (Plus (Const 1) (Term (ModelIntVar 0))))
     (Rel (Term (ModelIntVar 0)) ERLess (Const 3)))
   (BoolAnd
     (Rel (Const 1) ERLess (Plus (Const 1) (Term (ModelIntVar 1))))
     (Rel (Term (ModelIntVar 1)) ERLess (Const 3))),
  Rel (Term (ModelIntVar 0)) ERDiff (Term (ModelIntVar 1)),
  Rel (Plus (Const 1) (Term (ModelIntVar 1))) EREqual
    (Term (ModelIntVar 2)),
  Rel (Term (ModelIntVar 0)) ERDiff (Term (ModelIntVar 2)),
  Rel (Plus (Const -1) (Term (ModelIntVar 1))) EREqual
    (Term (ModelIntVar 3)),
  Rel (Term (ModelIntVar 0)) ERDiff (Term (ModelIntVar 3)),
  BoolConst True ]
```

Listing 84: Beispiel: 2-Damen-Problem (MCP-Modellierungssprache)

4.2.2. Konstruktion und Lösung von MCP-Baummodellen

Im letzten Abschnitt wurde beschrieben, wie man die KiCS2-interne Darstellung von Finite-Domain-Constraints in die Modellierungssprache des Monadic-Constraint-Programming-Frameworks übersetzt. MCP-Solver lösen die Constraints dieser Modellierungssprache jedoch nicht direkt, sondern sie arbeiten auf einem baumförmigen Modell, das sie gemäß einer zuvor festgelegten Suchstrategie auswerten. Das bedeutet, ein Solver verfolgt einen Pfad durch das Baummodell und interpretiert die Knoten entlang dieses Pfades. Abhängig vom gerade betrachteten Knoten erzeugt er dann z.B. eine neue Constraint-Variable oder trägt ein Constraint in seinen Constraint-Speicher ein. Letztlich führt die Auswertung eines Pfades von der Wurzel bis zu einem Blattknoten entweder zu einer Lösung des modellierten Problems oder zu einem Fehlschlag.

Dieser Abschnitt erklärt die Konstruktion eines solchen Baummodells aus einer Liste von MCP-FD-Constraints. Außerdem wird auf das Labeling von Constraint-Variablen im MCP-Framework eingegangen. Abschließend wird der Aufruf der beiden vom Framework bereitgestellten FD-Solver sowie die Vorbereitung der gefundenden Lösungen für die Ausgabe durch KiCS2 gezeigt.

Konstruktion eines MCP-Baummodells:
Im Grundlagenkapitel über das MCP-Framework wurden zwei unterschiedliche Implementierung für die Constraint-Funktionen der MCP-Modellierungssprache vorgestellt, eine, die nur einen passenden FD-Constraint-Term erzeugt, und eine, die für dieses Constraint einen Baummodellknoten erzeugt. Für die zweite Implementierung wurde die Framework-Funktion `addM` verwendet, die ein gegebenes MCP-FD-Constraint durch Konstruktion eines `Add`-Knotens einem trivialen Baummodell hinzufügt. Diese Funktion kann man auch verwenden, um für eine Liste von MCP-FD-Constraints ein zugehöriges Baummodell zu generieren.
Als zweites Hilfsmittel benötigt man dann noch eine MCP-Collection derjenigen Variablen, über denen das Labeling durchgeführt werden soll. Zur Erinnerung: Modelliert man ein Constraint-Problem direkt mit der MCP-Modellierungssprache, so erfolgt dies durch Angabe eines Baummodells vom Typ `FDSolver solver => Tree solver ModelCol` (vergleiche Kapitel 2.3.2, Listing 34). Über der im Baummodell eingekapselten Collection von Constraint-Variablen wird beim Aufruf eines FD-Solvers das Labeling durchgeführt. Somit kann man zur Übersetzung einer Liste von MCP-FD-Constraints in ein monadisches Baummodell folgende Funktion definieren:

```
addM :: (Constraint solver ~ Either Model q) =>
        Model -> Tree solver ()
addM m = addC $ Left m

toModelTree :: FDSolver s => [Model] -> ModelCol
            -> Tree (FDInstance s) ModelCol
toModelTree model mcpLabelVars =
  mapM_ addM model >> return mcpLabelVars
```

Listing 85: Umwandlung einer Liste von FD-Constraints in ein MCP-Baummodell

Das erzeugte Baummodell ist dabei vom Typ `Tree (FDInstance s) ModelCol`. Das heißt, man parametrisiert das Modell bereits mit dem Solver `FDInstance s`, der als Wrapper für Finite-Domain-Solver dient, die die Typklasse `FDSolver` implementieren. Die Transformation der gegebenen Liste von MCP-FD-Constraints mit der Funktion `addM` erzeugt für jedes Constraint in der Liste einen `Add`-Baumknoten. Durch die Verwendung von `mapM_` werden alle diese `Add`-Knoten sequentialisiert. Das heißt, sie werden zu einem einzigen Baummodell verknüpft. Durch den Aufruf von `return mcpLabelVars` wird schließlich ein `Return`-Knoten für die übergebenen Labeling-Variablen generiert und dem Baummodell als Blattknoten hinzugefügt.
Wie ein direkt mit der MCP-Modellierungssprache formuliertes Modell, ist das resultierende Baummodell dann bereit für das Labeling und den Aufruf eines konkreten FD-Solvers.

Labeling in einem MCP-Modell:
Für das Labeling stellt das MCP-Framework erneut ein Interface zur Implementierung durch konkrete Solver bereit:

```
class (Solver s, Term s t, Show (TermBaseType s t)) =>
  EnumTerm s t where
    type TermBaseType s t :: *

    getDomainSize :: t -> s (Int)
    getDomain :: t -> s [TermBaseType s t]
    setValue :: t -> TermBaseType s t -> s [Constraint s]
    getValue :: t -> s (Maybe (TermBaseType s t))
    ...
```

Listing 86: Labeling Interface für FD-Solver (Ausschnitt)

Eine Instanz der `EnumTerm`-Typklasse wird über zwei Typvariablen mit einer konkreten MCP-Solver- und MCP-Term-Implementierung verknüpft. Zur Wiederholung: Durch eine Instanz der Typklasse `Term` wird ein Term-Typ und damit insbesondere die Darstellung von Constraint-Variablen für einen MCP-Solver festgelegt.
Bei der Instanziierung dieser Typklasse muss ein sogenannter `TermBaseType s t` angegeben werden. Dabei handelt es sich um den für den Wertebereich der Constraint-Variablen verwendeten Typ, also z.B. `Int` für FD-Constraints über Integer-Domains. Darüberhinaus stellt dieses Interface eine Reihe von Funktionen zur Verfügung:

- `getDomainSize`: Bestimmt die Größe des Wertebereichs der gegebenen Constraint-Variable.
- `domain`: Bestimmt den derzeitigen Wertebereich der gegebenen Constraint-Variable. Dieser ist abhängig vom momentanen Zustand des Solvers (= Inhalt des Constraint-Speichers).
- `setValue`: Bindet eine Constraint-Variable durch ein Gleichheits-Constraint an einen Wert aus dem Wertebereich.

- `getValue`: Bestimmt den Wert einer Constraint-Variablen, falls ihr Wertebereich aus nur einem Wert besteht. Ansonsten wird `Nothing` zurückgegeben.

Die Funktion `labelling :: (EnumTerm s t) => ([t] -> s [t]) -> [t] -> Tree s ()` führt dann das eigentliche Labeling durch. Als Parameter erhält sie eine Labeling-Strategie in Form einer Funktion und die Labeling-Variablen. Durch Anwendung der Strategie-Funktion werden die Variablen in die Reihenfolge gebracht, in der das Labeling durchgeführt werden soll. Dann wird jede Labeling-Variable in der umsortierten Liste mit den Werten aus dem Wertebereich belegt. Dazu erzeugt `labelling` ein (Teil-)Baummodell bestehend aus `Add`-Knoten, die ein Constraint hinzufügen, das eine Labeling-Variable an einen Wert aus dem derzeit gültigen Wertebereich bindet. Man spricht von dem "derzeitig gültigen Wertebereich", da das Labeling während der Auswertung eines Baummodells durch den Solver durchgeführt wird. Das heißt, dieser Teil des Baummodells wird dynamisch während der Auswertung durch den Solver konstruiert. Auf diese Weise müssen nicht die Teilmodelle für alle möglichen Variablenbelegungen generiert werden, sondern nur diejenigen, die sich aus den gültigen Constraints während der Solver-Auswertung ergeben.
Um die gleiche Constraint-Variable in einem Baummodell mit unterschiedlichen Werten belegen zu können, werden entsprechende Teilbäume disjunktiv mit `Try`-Knoten verknüpft.
Das MCP-Framework unterstützt vier Labeling-Funktionen: `inOrder`, `firstFail`, `middleOut` und `endsOut`. Das Labeling-Verhalten dieser Funktionen wurde in Kapitel 4.1.1 bereits beschrieben.

Mit der Funktion `assignments` werden schließlich Blattknoten für gefundene Lösungen im Baummodell erzeugt. Das bedeutet, für jede Constraint-Variable aus der Liste der Labeling-Variablen wird ein `Return`-Knoten generiert, falls sich für diese Variable in diesem Auswertungszweig genau eine mögliche Belegung ergeben hat. Andernfalls wird ein `Fail`-Knoten für die Variable erzeugt.
Die Implementierung der `labelling`-Funktion, der `assignments`-Funktion, der Labeling-Strategie-Funktionen sowie die vollständige `EnumTerm`-Typklasse findet man im Anhang B.
Zur Durchführung des Labelings haben die MCP-Entwickler das Baummodell um einen Knoten erweitert:

```
data Tree solver a = ... | Label (solver (Tree solver a))
```

Listing 87: MCP-Baummodell: `Label`-Knoten

Dieser Knoten ermöglicht die dynamische Erzeugung von Variablenbelegungen während des Labelings in Abhängigkeit vom Solver-Zustand. Daher enthält der `Label`-Knoten die generierten Teilmodelle eingekapselt in der Solver-Monade.

Mit den vorgestellten MCP-Funktionen für das Labeling und der Erweiterung des Baummodells kann man die Funktion `labelWith` definieren, die für eine gegebene Labeling-Strategie und eine Liste von Labeling-Variablen bei Aufruf eines FD-Solvers die entsprechenden (Teil-)Baummodelle konstruiert.

```
labelWith :: (FDSolver s, EnumTerm s (FDIntTerm s)) =>
  LabelingStrategy -> ModelCol
  -> Tree (FDInstance s) [TermBaseType s (FDIntTerm s)]
labelWith strategy col = Label $ do
  lst <- getColItems col maxBound
  return $ do
    lsti <- colList col $ length lst
    labelling (matchStrategy strategy) lsti
    assignments lsti

matchStrategy :: EnumTerm s t => LabelingStrategy -> [t] -> s [t]
```

Listing 88: Labeling von MCP-Baummodellen

Mit Hilfe der vordefinierten MCP-Funktionen `getColItems` und `colList` (die Implementierung findet man im Anhang B) wird die übergebene MCP-Collection in eine Liste von Constraint-Variablen vom Typ `ModelInt` transformiert. Für diese Constraint-Variablen werden dann durch Aufruf von `labelling` und `assignments` die (Teil-)Baummodelle für das Labeling bzw. die `Return`-Blattknoten für mögliche Lösungen generiert. Zuvor ordnet die Funktion `matchStrategy` der gegebenen Labeling-Strategie noch die passende Labeling-Strategie-Funktion des Frameworks zu.

Aufruf der MCP-Solver:
Abschließend soll in diesem Abschnitt nun noch gezeigt werden, wie man die beiden FD-Solver des Frameworks aufruft. Dazu definiert man zunächst die folgende Funktion:

```
data MCPSolver = Overton | Gecode

solveWithMCP :: MCPSolver -> [Model] -> MCPLabelInfo -> MCPSolution
solveWithMCP Overton mcpCs info = solveWithOverton mcpCs info
solveWithMCP Gecode  mcpCs info = solveWithGeoce   mcpCs info
```

Listing 89: Aufruf der MCP-Solver

Um einen der beiden Solver des MCP-Frameworks auswählen zu können, definiert man sich die Datenstruktur `MCPSolver` mit den Werten `Overton` für den direkt in Haskell realisierten Solver (basierend auf der Implementierung von David Overton [8]) und `Gecode` für die C++-basierte Gecode-Solver-Bibliothek. Das Framework selbst ruft beide Solver über die gleiche überladene `solve`-Funktion auf, wie man weiter unten sieht.
Die Funktion `solveWithMCP` ruft je nach übergebenem MCP-Solver eine der beiden tatsächlichen Lösungsfunktionen auf. Dabei reicht sie die Liste der übersetzten Constraints sowie die gesammelten Labeling-Informationen weiter.

Ihr Rückgabewert ist vom Typ `MCPSolution`. Es handelt sich hierbei um einen Typ, der die gefundenen Lösungen zusammen mit zusätzlichen Informationen enthält, die die Ausgabe der Lösungen durch KiCS2 erleichtern:

```
type MCPSolution = SolutionInfo C_Int (FDTerm Int)

data SolutionInfo a b = SolInfo {
  solutions :: [[a]],
  labelVars :: [b],
  choiceID  :: ID
}
```

Listing 90: Datenstruktur zur Speicherung von Lösungsinformationen

`MCPSolution` wird mit Hilfe der polymorphen Datenstruktur `SolutionInfo a b` definiert. Diese Datenstruktur speichert alle nötigen Informationen, um die für ein Constraint-Problem gefundenen Lösungen durch KiCS2 ausgeben zu lassen. Dabei legen die beiden Typvariablen zum einen den Typ für die Lösungswerte (`a`) und zum anderen den Typ der für das Labeling verwendeten Constraint-Variablen (`b`) fest. Zu den gespeicherten Informationen gehören die gefundenen Lösungen, die Labeling-Variablen (vor der Übersetzung) sowie eine unbenutzte `ID`. Warum man genau diese Informationen für die Lösungsausgabe benötigt, wird im nächsten Abschnitt erklärt. In diesem Fall werden für die MCP-Solver die gefundenen Integer-Lösungswerte in ihrer `C_Int`-Repräsentation und die Labeling-Variablen in ihrer `(FDTerm Int)`-Darstellung gesichert.

Mit diesem Vorwissen kann nun die Implementierung von `solveWithOverton` beispielhaft betrachtet werden:

```
type OvertonTree = Tree (FDInstance OvertonFD) ModelCol

solveWithOverton :: [Model] -> MCPLabelInfo -> MCPSolution
solveWithOverton mcpCs info = case maybeMCPVars of
  Nothing      -> error "MCPSolver.solveWithOverton: Found no
                         variables for labeling."
  Just mcpVars ->
    let vars      = fromJust (labelVars info)
        choiceID  = fromJust (labelID info)
        strtgy    = fromJust (strategy info)
        modelTree = toModelTree mcpCs mcpVars
        solutions = snd $ solve dfs it $
          (modelTree :: OvertonTree) >>= labelWith strtgy
    in SolInfo (map (map toCurry) solutions) vars choiceID
  where maybeMCPVars = mcpLabelVars info
```

Listing 91: Aufruf des Overton-Solvers

Diese Funktion prüft zunächst, ob das `MCPLabelInfo`-Objekt eine MCP-Collection mit den übersetzten Labeling-Variablen enthält. Ist dies nicht der Fall, so wird die Auswertung der FD-Constraints abgebrochen und eine passende Fehlermeldung zurückgegeben. Andernfalls werden auch die anderen zwischengespeicherten Labeling-Informationen bestimmt und mit `fromJust` aus dem `Maybe`-Wrapper "ausgepackt". Durch Aufruf von `toModelTree` wird dann aus der Liste der MCP-FD-Constraints (`mcpCs`) und mit den übersetzten Labeling-Variablen (`mcpVars`) ein Baummodell konstruiert. Schließlich wird die MCP-Funktion `solve` aufgerufen. Neben einem Baummodell erwartet diese eine primitive Suchstrategie und einen sogenannten Such-Transformer als Argument. Ein solcher Such-Transformer kann verwendet werden, um komplexere Suchalgorithmen umzusetzen. Beispielsweise gibt es Such-Transformer, die die Auswertung eines Baummodells ab einer bestimmten Tiefe im Baum oder ab einer gewissen Anzahl gefundener Lösungen abbrechen.
Hier wird als primitive Suchstrategie die Tiefensuche (`dfs`) und als Such-Transformer der Identitäts-Transformer (`it`) gewählt, der das Baummodell nicht verändert.
Da MCP-Baummodelle mit dem zu verwendenden Constraint-Solver parametrisiert werden, muss das zuvor erzeugte Baummodell vor dem Aufruf von `solve` durch eine Typannotation noch in ein Baummodell für den Overton-Solver (`OvertonTree`) gecastet werden. Auf diese Weise ist sichergestellt, dass die "richtige Implementierung" der überladenen `solve`-Funktion aufgerufen wird.
Außerdem muss noch das Labeling für das Baummodell durchgeführt werden. Dazu wird die weiter oben vorgestellte Funktion `labelWith` mit der gewählten Labeling-Strategie über der im Baummodell eingekapselten Collection der Labeling-Variablen aufgerufen.
Als Rückgabewert liefert die Funktion `solve` ein Tupel bestehend aus der Anzahl der Auswertungsschritte und einer Liste der gefundenen Lösungen zurück. Da man hier nur an den Lösungen interessiert ist, wählt man mit `snd` nur die Lösungsliste aus. Diese Lösungsliste hat abhängig vom Solver entweder den Typ `[[Int]]` (Overton) oder `[[Integer]]`. Ihre einzelnen Lösungswerte werden dann mit Hilfe der Funktion `toCurry` (Gegenstück zur in Kapitel 4.1.2 im Listing 44 vorgestellten `fromCurry`-Funktion) in Werte vom Typ `C_Int` übersetzt. Wie man im nächsten Abschnitt sehen wird, ist dies für die Ausgabe der Lösungswerte durch KiCS2 erforderlich.
Mit den übersetzten Lösungen, den gesicherten Labeling-Variablen und der ebenfalls zwischengespeicherten unbenutzten `ID` wird dann ein neues `MCPSolution`-Objekt konstruiert und zurückgegeben.
Die Funktion `solveWithGecode` wird nach dem gleichen Prinzip implementiert. Dabei ist der einzig wirkliche Unterschied das Casting des Baummodells: In diesem Fall muss nämlich der Typ `Tree (FDInstance (GecodeWrappedSolver RuntimeGecodeSolver)) ModelCol` für die Typannotation verwendet werden, damit die entsprechende `solve`-Implementierung aufgerufen wird. Die vollständige Realisierung des Gecode-Solvers findet man im Anhang C.

4.2.3. Ausgabe der vom Solver bestimmten Lösungen durch KiCS2

In den beiden vorherigen Abschnitten wurde beschrieben, wie man die KiCS2-interne Darstellung eines FD-Constraint-Modells in ein semantisch äquivalentes MCP-Modell übersetzt und dieses

dann mit einem der beiden FD-Solver des Frameworks löst. Nun muss nur noch eine Möglichkeit gefunden werden, diese Lösungen mit Hilfe von KiCS2 auszugeben. Dazu müssen die Lösungen in irgendeiner Form wieder in den Suchbaum eingebunden werden, den die verschiedenen Suchstrategien von KiCS2 auswerten.

Zunächst sollte man sich noch einmal klarmachen, was man erreichen will: Im Allgemeinen ist es das Ziel beim Lösen eines Finite-Domain-Constraint-Problems, Belegungen für die darin vorkommenden Constraint-Variablen zu finden, so dass alle Constraints erfüllt sind. Modelliert man ein solches Problem mit der in Kapitel 4.1.1 vorgestellten CLPFD-Bibliothek, so kennzeichnet man diejenigen Constraint-Variablen, für die die Solver Belegungen ermitteln sollen, mit Hilfe eines `labeling`- bzw. `labelingWith`-Constraints. Diesen Labeling-Funktionen wird man bei der Modellierung in Curry in der Regel eine Liste von freien Variablen (vom Typ `Int`) übergeben, über denen auch die weiteren FD-Constraints definiert sind. KiCS2-intern werden diese freien Variablen dann zunächst in FD-Constraint-Variablen vom Typ `FDTerm Int` und dann in Integer-Variablen der MCP-FD-Modellierungssprache übersetzt. Die FD-Solver des Frameworks ermitteln dann Belegungen für diese Integer-Variablen.
Somit hat man mit Hilfe der MCP-FD-Solver Integer-Werte für die freien Variablen bestimmt, über denen das Constraint-Problem ursprünglich in Curry formuliert wurde. Das Ziel muss es nun also sein, die gefundenen Lösungen an diese freien Variablen zu binden, so dass deren Bindungen beim Aufruf eines Constraint-Modells in KiCS2 auf der Kommandozeile ausgegeben werden können.

Die Idee ist nun, dass man die vom Solver gefundenen Lösungswerte mit Hilfe von Curry-Bindungs-Constraints an die zu den Labeling-Variablen zugehörigen freien Variablen bindet. Dabei macht man sich die KiCS2-`bind`-Funktion zunutze, die in Kapitel 4.1.4 (vergleiche Listing 64) vorgestellt wurde. Diese überladene Funktion kann einen Wert eines Typen, der die Typklasse `Unifiable` implementiert, durch ein Bindungs-Constraint an eine gegebene `ID` binden.
Mit Hilfe von `bind` erzeugt man für jede Labeling-Variable einen `Guard`-Ausdruck mit einem Bindungs-Constraint, das den gefundenen Lösungswert an die `ID` dieser Labeling-Variable und damit an die zugehörige freie Variable bindet. Da es durchaus vorkommen kann, dass für eine Variable mehrere Belegungen möglich sind, muss man in solchen Fällen je einen `Guard`-Ausdruck für jede Belegung konstruieren und die verschiedenen `Guard`-Ausdrücke dann mit Hilfe von KiCS2-Standard-`Choice`s verknüpfen.
Wenn man nun also einen `Guard`-Ausdruck der Form `Guard (FDConstr fdCS) e` mit einer KiCS2-Suchstrategie wie der Tiefensuche auswertet, dann löst man zunächst das durch `fdCs` gegebene Constraint-Modell mit Hilfe der MCP-FD-Solver. Diese geben einen `MCPSolution`-Wert mit den ermittelten Lösungen und den für das Labeling verwendeten Variablen zurück. Mit Hilfe dieser Informationen generiert man `Guard`-Ausdrücke mit entsprechenden Bindungs-Constraints. Aus diesen `Guard`s, die teilweise auch durch `Choice`s verknüpft sein können, und dem durch die FD-Constraints beschränkten nicht-deterministischen Ausdruck `e` konstruiert man schließlich einen neuen nicht-deterministischen Ausdruck, der mit Hilfe der jeweiligen KiCS2-Suchstrategie weiter ausgewertet werden kann.

Auf diese Weise werden die generierten Bindungs-Constraint gelöst und KiCS2 kann die Bindungen für die zu den Labeling-Variablen zugehörigen freien Variablen, also die Lösungen eines Constraint-Problems, auf der Kommandozeile ausgeben.

```
...
  dfsGuard (FDConstr fdCs) e =
    runMCPSolver Overton fdCs e >>= dfs cont
...

runMCPSolver :: (Store m, NonDet a) => MCPSolver
             -> [FDConstraint] -> a -> m a
runMCPSolver solver fdCs e = do $
  updatedCs <- mapM updateFDConstr fdCs
  let (mcpCs,info) = translateToMCP fdCs
      solutionInfo = solveWithMCP solver mcpCs info
  return $ bindSolutions solutionInfo e
```

Listing 92: Aufruf des Overton-Solvers (Auswertung mit Tiefensuche)

Dabei ist die Funktion `bindSolutions`, die einen neuen nicht-deterministischen Ausdruck mit den passenden Bindungs-Constraints generiert, folgendermaßen definiert:

```
bindSolutions :: NonDet a => MCPSolution -> a -> a
bindSolutions (SolInfo []     _  _) _ = failCons
bindSolutions (SolInfo [s]    vs _) e = bindLabelVars vs s e
bindSolutions (SolInfo (s:ss) vs i) e = choiceCons i solution
                                                   solutions
  where
   solution  = bindLabelVars vs s e
   solutions = bindSolutions (SolInfo ss vs (leftID i)) e
```

Listing 93: Bindung der Constraint-Lösungen

Falls die Lösungsliste des `MCPSolution`-Werts leer ist, so wird der Konstruktor für eine fehlgeschlagene Berechnung zurückgegeben. Falls nur noch eine Lösung gebunden werden muss, so wird die Funktion `bindLabelVars` (siehe unten) aufgerufen. Bei mehreren Lösungen wird hingegen eine binäre `Choice` über den verschachtelten `Guard`-Ausdrücken zur Bindung der ersten Lösung (`solution`) und den `Choice`s zur Bindung der restlichen Lösungen (`solutions`) konstruiert. Für die Konstruktion dieser `Choice` wird die bislang unbenutzte `ID i` verwendet. Zur Bindung der restlichen Lösungen erfolgt ein rekursiver Aufruf von `bindSolutions` mit einem aktualisierten `MCPSolution`-Wert, der die restlichen Lösungen `ss` sowie eine neue `ID` durch den Aufruf von `leftID i` erhält.

Die Funktion `bindLabelVars` bindet die Labeling-Variablen an eine Lösung. Das heißt, jede einzelne Labeling-Variable wird an eine Belegung gebunden. Dazu erzeugt die Funktion `bindLabelVar` für jede Variable einen `Guard`-Ausdruck mit einem Curry-Bindungs-Constraint, das durch den Aufruf von `bind` generiert wird. Durch den rekursiven Aufruf von `bindLabelVars` werden die einzelnen `Guard`-Ausdrücke ineinander verschachtelt, so dass letztendlich nur ein einzelner Ausdruck resultiert.
Stimmt die Länge der Liste der Variablenbelegungen nicht mit der Länge der Liste der Labeling-Variablen überein, so gibt `bindLabelVars` eine Fehlermeldung zurück.
Konstante Werte in der Liste der Labeling-Variablen werden von `bindLabelVar` ignoriert.

```
bindLabelVars :: NonDet a => [FDTerm Int] -> [C_Int] -> a -> a
bindLabelVars []     []     e = e
bindLabelVars []     (_:_)  _ = error "bindLabelVars: List of
  labeling variables and solutions have different length"
bindLabelVars (_:_)  []     _ = error "bindLabelVars: List of
  labeling variables and solutions have different length"
bindLabelVars (v:vs) (s:ss) e =
  bindLabelVar v s (bindLabelVars vs ss e)

bindLabelVar :: NonDet a => FDTerm Int -> C_Int -> a -> a
bindLabelVar (Var i) v e   =
  guardCons (ValConstr i v (bind i v)) e
bindLabelVar (Const _) _ e = e
```

Listing 94: Bindung der Labeling-Variablen

4.3. Entwicklung generischer Schnittstellen zur Unterstützung weiterer Constraints und FD-Solver

In den beiden vorigen Kapiteln wurde beschrieben, wie man die KiCS2-Curry-Implementierung um eine Bibliothek für Finite-Domain-Constraints erweitern kann, indem man die Solver des Haskell-basierten Monadic-Constraint-Programming-Frameworks in KiCS2 integriert. Die bisherige Realisierung hat jedoch einige Nachteile: Beispielsweise gibt es keine Schnittstelle, die einem das Hinzufügen weiterer Constraint-Bibliotheken erleichtert. Stattdessen müsste für jeden neu eingeführten Constraint-Typ der in den `Guard`-Ausdrücken verwendete Typ `Constraints` erweitert werden, wenn man so vorgeht wie bei der FD-Constraint-Erweiterung. Des Weiteren gibt es bislang auch kein allgemeines Interface zur Anbindung weiterer FD-Solver an KiCS2.

Dieses Kapitel beschreibt die Definition generischer Schnittstellen, die die Integration weiterer Constraint-Typen und FD-Solver in KiCS2 erleichtern. Es ist in drei Abschnitte unterteilt: Der erste Abschnitt führt eine Schnittstelle für FD-Terme in KiCS2 ein, durch die unter anderem die Übersetzung der Haskell-Repräsentation eines Curry-Typs in einen FD-Term implementiert wird. Im zweiten Teilabschnitt wird ein generisches Interface zur Anbindung weiterer FD-Solver an KiCS2 vorgestellt. Der letzte Abschnitt dieses Kapitels befasst sich schließlich mit der Erweiterung von KiCS2 um eine allgemeine Constraint-Schnittstelle, die es ermöglicht, beliebige Constraint-Typen mit Hilfe von `Guard`-Ausdrücken durch die Implementierung durchzureichen und lösen zu lassen.

4.3.1. FD-Term-Interface

Bei der Vorstellung der Erweiterung von KiCS2 um eine Bibliothek zur Modellierung von Finite-Domain-Constraints wurde in Bezug auf die Integer-Argumente dieser Constraints stets mit festen Typen programmiert. So wurde beispielsweise eine Funktion `toFDTerm` implementiert, die die Haskell-Repräsentation eines Curry-Integer-Werts in einen Integer-FD-Term übersetzt (vergleiche Abschnitt 4.1.2, Listing 43). Des Weiteren wurde mit `updateFDVar` eine Funktion zur Aktualisierung von Integer-FD-Variablen in Bezug auf Bindungen definiert, die durch das Curry-Gleichheits-Constraint `(=:=)` eingeführt wurden (vergleiche Abschnitt 4.1.4, Listing 70). Schließlich hat man noch die Funktion `bindLabelVar` implementiert, mit deren Hilfe ein durch den Solver gefundener Integer-Wert an die zu einer Labeling-Variable zugehörige freie Variable gebunden wird (vergleiche Abschnitt 4.2.3, Listing 94). Alle diese Funktionen arbeiten mit den Typen `C_Int` bzw. `FDTerm Int`. Durch Einführung einer Typklasse `Constrainable` soll von der Festlegung auf diese Typen abstrahiert werden.
Die Typklasse ist somit ein KiCS2-internes Interface, das in erster Linie zur Transformation von durch Constraints beschränkbare Curry-Typen (bzw. genauer deren Repräsentation als Haskell-Typ) in KiCS2-interne Term-Typen dient.

```
class Constrainable ctype ttype where
  toCsExpr :: ctype -> ttype
```

Listing 95: Interface für durch Constraints beschränkbare Typen

Für diese Transformation stellt die Typklasse die Funktion `toCsExpr` zur Implementierung bereit. Es handelt sich bei `Constrainable` um eine Typklasse mit mehr als einem Typparameter (Spracherweiterung *MultiParamTypeClasses*). Daher ist es auch möglich mehr als eine interne Term-Repräsentation für einen Curry-Typ anzugeben.
Zur Umwandlung von Curry-Integer-Werten vom Typ `C_Int` in FD-Terme vom Typ `FDTerm Int` kann man nun beispielsweise die folgende Instanz definieren:

```
instance Constrainable C_Int (FDTerm Int) where
  toCsExpr (Choices_C_Int i@(FreeID _) _) = FDVar i
  toCsExpr x                              = Const (fromCurry x)
```

Listing 96: Beispielinstanz zur Erzeugung von Integer-FD-Termen

Man implementiert die Funktion `toCsExpr` dazu einfach nach dem Vorbild von `toFDTerm` (vergleiche Kapitel 4.1.2, 43). Damit kann man die speziellere Funktion (`toFDTerm`) durch eine allgemeinere (`toCsExpr`) ersetzen.

Mit Hilfe von `Constrainable` kann man auch bei den beiden anderen oben erwähnten Funktionen von konkreten Typen abstrahieren. Dazu passt man beispielsweise die Implementierung von `updateFDVar` wie folgt an:

```
updateFDVar :: (Constrainable c (FDTerm t), FromDecisionTo c,
                Store m) => FDTerm t -> m (FDTerm t)
updateFDVar c@(Const _) = return c
updateFDVar (FDVar i)   = do x <- lookupValue i
                             return (toCsExpr x)
```

Listing 97: Funktion zur Aktualisierung von FD-Variablen (angepasst)

Für diese Realisierung kommen die Haskell-Spracherweiterungen *flexible contexts* und *functional dependencies* zum Einsatz. Die erstgenannte Erweiterung ermöglicht die Einschränkung des `Constrainable`-Typklassen-Constraints auf Term-Typen vom Typ `FDTerm t`. Die zweite ist erforderlich, um die Typklasse `Constrainable` folgendermaßen anpassen zu können:

```
class Constrainable ctype ttype | ttype -> ctype where
...
```

Auf diese Weise wird eine funktionale Abhängigkeit zwischen dem internen Term-Typ und dem ursprünglichen Curry-Typ eingeführt. Diese Abhängigkeit wird für die obige Implementierung der

Funktion `updateFDVar` benötigt, damit die richtige `Constrainable`-Instanz für den Aufruf von `toCsExpr` ausgewählt werden kann. Ohne diese funktionale Abhängigkeit könnte man die obige allgemeinere Definition für `updateFDVar` nicht in dieser Form angeben.
Auch für die Funktionen zur Bindung von Lösungen, also `bindSolutions`, `bindLabelVars` und `bindLabelVar`, die in Kapitel 4.2.3 vorgestellt wurden, lassen sich mit Hilfe von `Constrainable` allgemeinere Typsignaturen angeben. Ihre Implementierung bleibt dabei unverändert.

```
bindLabelVar :: (Constrainable c (FDTerm t), Unifiable c,
                 NonDet a) => FDTerm t -> c -> a -> a

bindSolutions :: (Constrainable c (FDTerm t), Unifiable c,
                  NonDet a) =>
                   SolutionInfo c (FDTerm t) -> a -> a

bindLabelVars :: (Constrainable c (FDTerm t), Unifiable c,
                  NonDet a) => [FDTerm t] -> [c] -> a -> a
```

Listing 98: Angepasste Typsignaturen für Funktionen zur Bindung von Constraint-Lösungen

Die Typklasse `Constrainable` bietet also ein generisches Interface, mit dessen Hilfe man jeden durch Constraints beschränkbaren Typ auf eine oder mehrere unterschiedliche Constraint-Term-Repräsentationen abbilden kann. Außerdem ermöglicht sie die Abstraktion von konkreten Term-Typen bei weiteren Funktionen, die für die Entwicklung der KiCS2-CLPFD-Bibliothek definiert wurden.

4.3.2. FD-Solver-Interface

Im Rahmen dieser Arbeit wurden die Finite-Domain-Solver des Monadic-Constraint-Programming-Frameworks in KiCS2 integriert. Es wäre jedoch wünschenswert, dass man in Zukunft ohne großen Aufwand weitere FD-Solver in KiCS2 einbinden kann.
Dazu wird in diesem Abschnitt ein generisches FD-Solver-Interface vorgestellt, das Funktionen zur Implementierung bereitstellt, durch die man externe FD-Solver an KiCS2 anschließen kann. Zunächst wird schrittweise das Interface präsentiert, um dann abschließend zu zeigen, wie man die MCP-Solver über diese Schnittstelle an KiCS2 anbindet.
Die Schnittstelle wird durch die folgende Typklasse realisiert:

```
class ExternalFDSolver solver constraint where
...
```

Listing 99: Interface zur Integration von FD-Solvern

Die Typklasse `ExternalFDSolver` wird mit zwei Typvariablen parametrisiert: `solver` für den verwendeten FD-Solver und `constraint` für die FD-Constraints, die durch die (bei der Instanziierung) angegebene Implementierung gelöst werden.

Für die Realisierung verwendet man die *associated types*-Spracherweiterung (oder auch *type families*- bzw. *data families*-Spracherweiterung), mit der man die folgenden Hilfstypen einführt.

```
class ExternalFDSolver solver constraint where
  data SolverModel solver constraint :: *
  data LabelInfo solver constraint :: *
  data Solutions solver constraint :: *
  ...
```

Listing 100: Interface zur Integration von FD-Solvern - associated types

Mit dem Typ `SolverModel solver constraint` wird die FD-Modellierungssprache eines Solvers festgelegt, also eine Repräsentation derjenigen Constraints, die der Solver interpretieren und lösen kann. Der Typ `LabelInfo solver constraint` dient dazu, alle Informationen zu speichern, die für das Labeling durch den konkreten FD-Solver erforderlich sind, also zum Beispiel die Variablen, über denen das Labeling durchgeführt werden soll, oder die Labeling-Strategie. Und durch `Solutions solver constraint` wird festgelegt, in welcher Form die Lösungen eines bestimmten FD-Solvers zurückgegeben werden.

Diese Hilfstypen kommen in den vom `ExternalFDSolver`-Interface zur Verfügung gestellten Funktionen zur Anwendung:

```
class ExternalFDSolver s c where
  ...
  translate :: s -> [c] -> (SolverModel s c, LabelInfo s c)

  solveWith :: s -> SolverModel s c -> LabelInfo s c
            -> Solutions s c

  makeCsSolutions :: NonDet a => s -> Solutions s c -> a -> a
  ...
```

Listing 101: Interface zur Integration von FD-Solvern - Funktionen 1

Vier Funktionen müssen bei der Instanziierung von `ExternalFDSolver` implementiert werden (die vierte Funktion ist zusammen mit einer Default-Implementierung weiter unten angegeben):

- `translate`: Dient dazu, die allgemeine Repräsentation von FD-Constraints in ein semantisch äquivalentes Model mit den Constraints der Modellierungssprache des jeweiligen Solvers (`SolverModel s c`) zu übersetzen. Zusätzlich werden bei der Übersetzung alle für das Labeling relevanten Informationen gesammelt und als Wert vom Typ `LabelInfo s c` zurückgegeben.

- `solveWith`: Mit dieser Funktion wird der konkrete Solver `s` aufgerufen. Dazu werden das übersetzte Modell sowie die gesammelten Labeling-Informationen übergeben. Die Funktion liefert schließlich die gefundenen Lösungen in der zuvor spezifizierten Form zurück (`Solutions s c`).

- `makeCsSolutions`: Durch diese Funktion werden die vom FD-Solver gefundenen Lösungen (`Solutions s c`) in einen nicht-deterministischen Suchbaum (=nicht-deterministischen Ausdruck) integriert, der dann durch die KiCS2-Suchstragien weiter ausgewertet werden kann.

- `runSolver`: Ruft einen konkreten FD-Solver direkt mit der allgemeinen KiCS2-internen FD-Constraint-Repräsentation auf und liefert einen nicht-deterministischen Ausdruck zur weiteren Auswertung durch die KiCS2-Suchstrategien zurück. Für diese Funktion gibt die `ExternalFDSolver`-Typklasse eine Default-Implementierung an, die die anderen von der Typklasse bereitgestellten Funktionen kombiniert, um dieses Modell zu übersetzen, das übersetzte Modell lösen zu lassen und schließlich die gefundenen Lösungen in einen nicht-deterministischen Ausdruck zu überführen.

```
class ExternalFDSolver s c where
  ...
  runSolver :: NonDet a => s -> [c] -> a -> a
  runSolver solver fdCs e =
    do let (solverCs,info) = translate solver fdCs
           solutions       = solveWith solver solverCs info
       return $ makeCsSolutions solver solutions e
```

Listing 102: Interface zur Integration von FD-Solvern - Funktionen 2

`runSolver` ist diejenige Funktion, die bei Auswertung eines `Guard`-Ausdrucks mit FD-Constraints von den KiCS2-Suchstrategien aufgerufen wird, um die Constraints von einem (MCP-)Solver lösen zu lassen und einen nicht-deterministischen Ausdruck mit den Lösungen zur weiteren Auswertung zu konstruieren. Diese Funktion ersetzt also den Aufruf der Funktion `runMCPSolver` während der Auswertung durch Suchstrategien wie der Tiefensuche (vergleiche Kapitel 4.2.3, Listing 92).
Die Default-Implementierung von `runSolver` soll zeigen, wie man die Funktionen der FD-Solver-Schnittstelle zur Lösung von FD-Constraints kombinieren kann. Falls nötig kann sie jedoch bei der Instanziierung der Typklasse auch durch eine eigene Realisierung überschrieben werden.

Abschließend soll nun die `ExternalFDSolver`-Instanz für die FD-Solver des Monadic-Constraint-Programming-Frameworks und die KiCS2-interne Repräsentation von FD-Constraints durch den Typ `FDConstraint` betrachtet werden:

```
instance ExternalFDSolver MCPSolver FDConstraint where
  newtype SolverModel MCPSolver FDConstraint =
    ModelWrapper [Model]
```

```
newtype LabelInfo MCPSolver FDConstraint   =
  LabelWrapper MCPLabelInfo
newtype Solutions MCPSolver FDConstraint   =
  SolWrapper (SolutionInfo C_Int (FDTerm Int))

translate Overton fdCs = translateOverton fdCs
translate Gecode  fdCs = translateGecode fdCs

solveWith = solveWithMCP

makeCsSolutions _ (SolWrapper solutions) e =
  bindSolutions solutions e
```

Listing 103: Integration der MCP-Solver über die FD-Solver-Schnittstelle

Die drei *associated types* implementiert man mit Hilfe von Wrapper-Typen, in die man die passenden Typen, also `[Model]` als Typ für die MCP-FD-Modellierungssprache, `MCPLabelInfo` für die MCP-Labeling-Informationen und `SolutionInfo C_Int (FDTerm Int)` zur Darstellung von Lösungen, "einpackt". Diese Typcontainer sind erforderlich, wenn man die *associated types* durch Datentypen implementiert, die bereits an anderer Stelle definiert wurden.

Anders als in Kapitel 4.2.1 beschrieben gibt es für die beiden MCP-Solver unterschiedliche Übersetzungsfunktionen. Dies hängt damit zusammen, dass die beiden Solver auf unterschiedlichen Darstellungen von MCP-Collections arbeiten. Die in Kapitel 4.2.1 beschriebene Übersetzungsfunktion `translateToMCP` (vergleiche Listing 83) entspricht allerdings im Wesentlichen dem Übersetzungsvorgang für den Overton-Solver. Im Anhang C wird der Code für die beiden Übersetzungsfunktionen angegeben. Dort wird auch noch einmal kurz auf diese Problematik eingegangen.
Die Interface-Funktion `solveWith` wird mit Hilfe der in Kapitel 4.2.2 vorgestellten Funktion `solveWithMCP` (vergleiche Listing 89) implementiert. Deren Typsignatur und Implementierung ist nur in Hinblick auf die oben angegebenen Wrapper-Typen (also z.B. das "Ein- und Auspacken" der ursprünglich verwendeten Werte) anzupassen (siehe Anhang C).
Um die durch den Solver gefundenen Lösungen in den KiCS2-Suchbaum einzubauen, erzeugt man mit Hilfe von `bindSolutions` `Guard`-Ausdrücke mit passenden Bindungs-Constraints, wie in Kapitel 4.2.3 beschrieben. Das heißt, `makeCsSolutions` wird durch Aufruf von `bindSolutions` (vergleiche Listing 93) implementiert.

4.3.3. Generische Constraint-Schnittstelle

Im letzten Unterkapitel wurde ein generisches Interface zur Anbindung von Finite-Domain-Solvern an die KiCS2-Curry-Implementierung vorgestellt und beispielhaft für die Solver des Monadic-Constraint-Programming-Frameworks implementiert.
Dieser Abschnitt befasst sich mit der Entwicklung einer generischen Schnittstelle für Constraints.

Das bedeutet, mit Hilfe dieser Schnittstelle soll beim Durchreichen von Constraints (mittels `Guard`-Ausdrücken) und deren Lösung von konkreten Constraint-Typen wie FD-Constraints abstrahiert werden.

Dazu macht man sich Ideen zunutze, die für die Entwicklung einer "erweiterbaren dynamisch-getypten Hierarchie von Exceptions" in Haskell verwendet wurde (siehe [7]): Diese Arbeit stellt eine Möglichkeit vor, eine Objekt-orientierte API in Haskell zu realisieren. Sie beschreibt die Einführung eines erweiterbaren Exception-Typs in Haskell mit Hilfe von *Haskell existentials* und der *Data.Typeable*-Bibliothek. Des Weiteren erklärt sie, wie man Exception Handler durch Verwendung der *scoped-type-variables*-Spracherweitung implementieren kann.

Einige dieser Ideen werden aufgegriffen, um einen erweiterbaren Constraint-Typ in KiCS2 zu implementieren. Dieser Constraint-Typ soll als Wrapper für die tatsächlichen Constraints dienen. Das bedeutet, man definiert einen allgemeinen Constraint-Typ, in den konkrete Constraints wie die in dieser Arbeit vorgestellten FD-Constraints "eingepackt" werden. Mit Hilfe dieses Constraint-Wrappers werden die Constraints dann durch die Implementierung durchgereicht.
Dieser Constraint-Wrapper wird folgendermaßen definiert:

```
data WrappedConstraint =
  forall c . WrappableConstraint c => CWrapper c
```

Listing 104: Wrapper für Constraints

Wie man sieht, wird für die Definition das Haskell-Schlüsselwort `forall` verwendet. Es handelt sich hierbei um einen sogenannten *existential quantified type*. Der einzige Konstruktor von `WrappedConstraint` hat den Typ `CWrapper :: forall c . WrappableConstraint c => c -> WrappedConstraint`. Das bedeutet, man kann beliebige Typen an diesen Konstruktor übergeben. Solange diese die Typklasse `WrappableConstraint` implementieren, resultiert am Ende unabhängig vom ursprünglichen Typ stets ein Wert vom Typ `WrappedConstraint`. Wenn man einen Wert vom Typ `WrappedConstraint` vorliegen hat, weiß man also nicht, welcher Typ darin "eingepackt" wurde, aber man weiß, dass es sich um einen Typen handelt, der die Eigenschaften der Typklasse `WrappableConstraint` besitzt.
Diese Typklasse dient als Interface für den Constraint-Wrapper. So stellt sie unter anderem Funktionen zum "Ein- bzw. Auspacken" eines konkreten Constraints in bzw. aus dem Wrapper zur Verfügung:

```
class (Typeable c, Show c) => WrappableConstraint c where
  wrapCs :: c -> WrappedConstraint

  unwrapCs :: WrappedConstraint -> Maybe c

  updateVars :: Store m => c -> m c
```

Listing 105: Interface für erweiterbaren Constraint-Typ

Insgesamt werden drei Funktionen zur Implementierung bereitgestellt:

- `wrapCs`: "Packt" ein Constraint in den Constraint-Wrapper ein.
- `unwrapCs`: Mit dieser Funktion wird ein Constraint - falls möglich - wieder aus dem Wrapper "ausgepackt". Enthält der Wrapper nicht ein Constraint des erwarteten Typs, so soll diese Funktion `Nothing` zurückgeben.
- `updateVars`: Aktualisiert - falls vorhanden - die Constraint-Variablen in dem gegebenen Constraint hinsichtlich Bindungen, die durch das KiCS2-Gleichheits-Constraint `(=:=)` eingeführt wurden. Hierfür ist es erforderlich, die Bindungen im monadischen Decision-Store von KiCS2 nachzuschlagen (vergleiche Kapitel 4.1.4).

Typen, die diese Typklasse implementieren, müssen außerdem eine `Show`- und `Typeable`-Instanz besitzen. Ersteres hängt damit zusammen, dass es möglich sein soll, die "eingepackten" Constraints auf der Kommandozeile auszugeben (z.B. zum Debuggen). Die `Typeable`-Instanz ist hingegen für die Realisierung der Default-Implementierung von `unwrapCs` erforderlich. Für alle drei Funktionen werden nämlich Default-Implementierungen bereitgestellt:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b

class (Typeable c, Show c) => WrappableConstraint c where
  wrapCs :: c -> WrappedConstraint
  wrapCs = CWrapper

  unwrapCs :: WrappedConstraint -> Maybe c
  unwrapcs (CWrapper c) = cast c

  updateVars :: Store m => c -> m c
  updateVars = return
```

Listing 106: `cast`-Funktion und Default-Implementierung des Interface für erweiterbaren Constraint-Typ

Zum "Einpacken" eines Constraints wird einfach der Konstruktor `CWrapper` aufgerufen. "Ausgepackt" wird ein Constraint mit Hilfe der Funktion `cast` aus *Data.Typeable*. Lässt sich der gegebene Wert in den angegebenen Zieltyp casten, so liefert `cast` den Zielwert "eingepackt" in einem `Just` zurück. Andernfalls wird `Nothing` zurückgegeben. Damit man `cast` anwenden kann, ist es notwendig, dass Typen, die `WrappableConstraint` implementieren, auch eine `Typeable`-Instanz zur Verfügung stellen.
Die `updateVars`-Funktion wird in der Default-Implementierung einfach durch `return` realisiert. Das heißt, per Default werden die Variablenbindungen nicht aktualisiert. Diese Funktionsdefinition sollte man bei der Instanziierung von `WrappableConstraint` also gegebenenfalls überschreiben.

Um den Constraint-Wrapper in `Guard`-Ausdrücken zum Durchreichen von Constraints durch die Implementierung einsetzen zu können, muss der `Constraints`-Typ noch entsprechend angepasst werden. Dazu ersetzt man den speziell für FD-Constraints eingeführten Konstruktor `FDConstr` (vergleiche Listing 41) durch einen Konstruktor für `WrappedConstraints`:

```
data Constraints
  = ...
  | WrappedConstr [WrappedConstraint]
```

Listing 107: Anpassung des Datentyps `Constraints`

Damit ist es möglich, verschiedene Constraint-Typen, die die Typklasse `WrappableConstraint` implementieren, in einer heterogenen Liste vom Typ `[WrappedConstraint]` mit Hilfe von `Guard`-Ausdrücken durch die Implementierung durchzureichen. Der Vorteil dieser Realisierung gegenüber der ursprünglichen ist, dass man bei der Einführung weiterer Constraint-Typen in KiCS2 den `Constraints`-Typ nun nicht mehr erweitern oder anpassen muss. Anstatt für jeden weiteren Constraint-Typ einen neuen `Constraints`-Konstruktor zu definieren, kann man diese über das `WrappableConstraint`-Interface in KiCS2 integrieren.

Beispielhaft soll nun die Implementierung dieses Interfaces für die im Rahmen dieser Arbeit eingeführten Finite-Domain-Constraints vorgestellt werden:

```
data FDConstraint = ...
 deriving (Show,Typeable)

instance WrappableConstraint FDConstraint where
  updateVars = updateFDConstr updateFDVar
```

Listing 108: Implementierung des `WrappableConstraint`-Interfaces für FD-Constraints

Zur automatischen Erzeugung der erforderlichen `Typeable`-Instanz für den Typ `FDConstraint` verwendet man die Haskell-Spracherweiterung *derive data typeable*.
Bei der Instanziierung von `WrappableConstraint` wird `updateVars` neu implementiert und zwar mit den bekannten Funktionen `updateFDVar` und `updateFDConstr` (vergleiche Listings 70 und 71). Für die übrigen Funktionen der Schnittstelle verwendet man die Default-Implementierung.
Auch die Konstruktion der `Guard`-Ausdrücke durch die extern implementierten Funktionen der CLPFD-Bibliothek müssen entsprechend angepasst werden. Als Anschauungsbeispiel werden hier die angepassten Versionen der externen Funktionen für das Gleichheits-Constraint `(=#)` und das Wertebereichs-Constraint `domain` angegeben.

```
external_d_C_prim_FD_equal :: C_Int -> C_Int -> ConstStore
                           -> C_Success
external_d_C_prim_FD_equal x y _ =
  let c = wrapCs $ newRelConstr Equal x y
  in Guard_C_Success (WrappedConstr [c]) C_Success
```

```
newRelConstr :: RelOp -> C_Int -> C_Int -> FDConstraint
newRelConstr relOp x y = FDRel relOp (toCsExpr x) (toCsExpr y)

external_d_C_prim_domain :: OP_List C_Int -> C_Int -> C_Int
                         -> ConstStore -> C_Success
external_d_C_prim_domain vs l u _ =
  let c = wrapCs $
            FDDomain (toFDList vs) (toCsExpr l) (toCsExpr u)
  in Guard_C_Success (WrappedConstr [c]) C_Success
```

Listing 109: Externe FD-Constraint-Funktionen (angepasst)

Die Erzeugung der `FDConstraint`-Konstruktorterme bleibt dabei bis auf die Aufrufe von `toCsExpr` statt `toFDTerm` unverändert (vergleiche Listings 49 und 50). Zudem werde diese Terme durch Aufruf von `wrapCs` in einen Constraint-Wrapper "eingepackt" und mit diesem wird dann ein neuer `Guard`-Ausdruck konstruiert.

Schließlich muss auch die Funktion zum Einsammeln der Constraints angepasst werden: In Kapitel 4.1.3 wurde zum Einsammeln der FD-Constraints die Funktion `searchFDCs` definiert (vergleiche Listing 57). Diese wird durch die folgende Funktion ersetzt, die alle "eingepackten" Constraints in einem gegebenen Ausdruck sucht und in einem einzigen `Guard`-Ausdruck sammelt.

```
searchWrappedCs :: NormalForm a => a -> [WrappedConstraint] -> a
searchWrappedCs x wcs =
  match swChoice swNarrowed choicesCons failCons swGuard swVal x
 where
  swChoice i x1 x2    = choiceCons i (searchWrappedCs x1 wcs)
                                     (searchWrappedCs x2 wcs)
  swNarrowed i xs     = choicesCons i (map
                          (\x' -> searchWrappedCs x' wcs) xs)
  swGuard (WrappedConstr wc) e = searchWrappedCs e (wcs ++ wc)
  swGuard c e         = guardCons c (searchWrappedCs e wcs)
  swVal v | null wcs  = v
          | otherwise = guardCons (WrappedConstr wcs) v
```

Listing 110: Einsammeln von `WrappedConstraint`s

Mit dieser Funktion kann man - wie auch schon in Kapitel 4.1.3 für FD-Constraints beschrieben - alle in einem Pfad des KiCS2-Suchbaums vorkommenden `WrappedConstraint`s einzusammeln. Dies ist erforderlich, damit man beim Aufruf eines spezifischen Constraint-Solvers ein vollständiges Constraint-Modell an diesen weitergeben kann.

Die Frage ist nun, wie man alle Constraints eines speziellen Typs aus der heterogenen Constraint-Liste herausfiltert und den richtigen Solver für diese Constraints auswählt. Anders formuliert: Wie sorgt man dafür, dass alle "eingepackten" Constraints (durch Aufruf des passenden Solvers) gelöst werden? Die Idee zur Lösung dieses Problems liefert erneut [7]: In der Arbeit von Simon Marlow wird eine Möglichkeit vorgestellt, mit der man verschiedene Exception-Klassen in einem Ausdruck durch Angabe einer Liste von Exception-Handlern abfangen kann. Übertragen auf diese Arbeit bedeutet das, dass man versucht, die Liste der `WrappedConstraint`s zu lösen, indem man eine Liste von Constraint-Solvern angibt, die man nacheinander ausprobiert. Man durchsucht also die heterogene Liste aller Constraints nach einem Constraint-Typ für einen bestimmten Constraint-Solver. Findet man diesen Constraint-Typ, so lässt man die Constraints von dem entsprechenden Solver lösen und macht dann nach dem gleichen Prinzip mit der heterogenen Restliste und der Restliste der definierten Solver weiter. Anderfalls versucht man direkt einen anderen Solver anzuwenden. Nach diesem Muster fährt man fort, bis entweder alle Constraints in der heterogenen Liste gelöst wurden oder bis keine Constraint-Solver mehr zur Verfügung stehen, für die man die Liste nach Constraints durchsuchen könnte.

Bevor diese Lösung vorgestellt wird, passt man zunächst das Interface für KiCS2-externe FD-Solver, das im vorigen Abschnitt vorgestellt wurde, noch ein wenig an:

```
class WrappableConstraint c => ExternalFDSolver s c where
  ...
  runSolver :: (NonDet a, Store m) => s -> [c] -> a -> m a
  runSolver solver wcs e =
    do let updatedCs       = mapM updateVars wcs
           (solverCs,info) = translate solver updatedCs
           solutions       = solveWith solver solverCs info
       return $ makeCsSolutions solver solutions e
```

Listing 111: Anpassung des FD-Solver-Interfaces

Und zwar ergänzt man die Default-Implementierung von `runSolver` um den Aufruf der Funktion `updateVars`. Auf diese Weise werden die Variablenbindungen in den einzelnen FD-Constraints aktualisiert, bevor diese in Constraints der MCP-FD-Modellierungssprache übersetzt werden. Damit man diese Funktion aufrufen kann, ist es außerdem erforderlich, das Typklassen-Constraint `WrappableConstraint c` zur Typklassendefinition hinzuzufügen.

Nach diesem kurzen Einschub wird im Folgenden die Realisierung der oben vorgestellten Idee zur Lösung aller Constraints in einer heterogenen Liste vom Typ `[WrappedConstraint]` beschrieben. Für die Definition der Liste von Constraint-Solvern, die man versucht nacheinander anzuwenden, um alle Constraints in der heterogenen Liste zu lösen, führt man den folgenden Datentyp ein:

```
data (Store m, NonDet a) => Solver m a =
  forall c . (WrappableConstraint c) => Solver ([c] -> a -> m a)
```

Listing 112: Solver für `WrappedConstraints`

Wie schon bei `WrappedConstraint` macht der Einsatz der Haskell-Spracherweiterung *existential quantified type* in der Definition von `Solver m a` es möglich, Solver für verschiedene Constraint-Typen in diesen einzelnen Typ zu "packen". `Solver`, der einzige Konstruktor dieses Typs, erwartet eine Funktion als Argument, die ihrerseits eine Liste von "einpackbaren" Constraints sowie einen nicht-deterministischen Ausdruck entgegennimmt und einen neuen nicht-deterministischen Ausdruck eingekapselt in der `Store`-Monade zurückgibt.
Wie man sieht, entspricht der Typ dieses Funktions-Arguments dem Typ von `runSolver` aus der `ExternalFDSolver`-Typklasse, wenn man diese Funktion mit einem konkreten FD-Solver partiell appliziert. Somit kann man eine Liste der bislang von KiCS2 unterstützten Constraint-Solver wie folgt angeben:

```
runGecode :: (Store m, NonDet a) => [FDConstraint] -> a -> m a
runGecode = \(cs :: [FDConstraint]) e -> runSolver Gecode cs e

runOverton :: (Store m, NonDet a) => [FDConstraint] -> a -> m a
runOverton = \(cs :: [FDConstraint]) e -> runSolver Overton cs e

solvers :: (Store m, NonDet a) => [Solver m a]
solvers = [Solver runGecode, Solver runOverton]
```

Listing 113: Bislang unterstützte Constraint-Solver in KiCS2

Bei der Implementierung der `Solver`-Funktionen wird die Liste der Constraints, die der Gecode- bzw. Overton-Solver als Argument erwartet, durch eine Typannotation in den konkreten Typ `FDConstraint` gecastet. Wie gleich gezeigt wird, ist diese Typannotation (mittels Spracherweiterung *scoped type variables*) notwendig, um die passenden Constraints für den jeweiligen Solver aus der heterogenen Liste herauszufiltern.

Das Herausfiltern aller Constraints eines bestimmten Typs erfolgt mit der Funktion `filterCs`:

```
filterCs :: WrappableConstraint c => [WrappedConstraint]
         -> ([c],[WrappedConstraint])
filterCs []       = ([],[])
filterCs (wc:wcs) = let (cs,wcs') = filterCs wcs
                    in case unwrapCs wc of
                         Just c  -> (c:cs,wcs')
                         Nothing -> (cs,wc:wcs')
```

Listing 114: Filtern der heterogenen Constraint-Liste

Diese gibt ein Tupel bestehend aus der Liste der Constraints des gesuchten Typs sowie der Liste aller übrigen `WrappedConstraints` zurück. Um die Constraints eines bestimmten Typs herauszufiltern, ruft sie die Funktion `unwrapCs` aus dem `WrappableConstraint`-Interface auf: Je nachdem, ob sich ein Constraint des gesuchten Typs aus dem `WrappedConstraint wc` "auspacken" lässt, wird dieses Constraint der einen oder anderen Ergebnisliste hinzugefügt. Nach diesem Muster werden alle `WrappedConstraints` der heterogenen Liste gefiltert.
Da `filterCs` die Funktion `unwrapCs` aufruft, die ihrerseits mit Hilfe von `cast` aus *Data.Typeable* implementiert wurde, ist es notwendig, den Typ für die herauszufilternden Constraints anzugeben. Denn das Casting funktioniert nur, wenn der Zieltyp bekannt ist. Aus diesem Grund sind die Typannotationen oben bei der Definition der `Solver`-Funktionen erforderlich.

Mit der Filterfunktion und der Liste der unterstützten Constraint-Solver kann man nun eine Funktion angeben, die versucht sämtliche "eingepackten" Constraints in der heterogenen Liste zu lösen, indem sie diese nacheinander nach Constraints für die einzelnen Solver durchsucht.

```
solveAll :: (Store m, NonDet a) => [WrappedConstraint]
         -> [Solver m a] -> a -> m a
solveAll wcs []                          _ = error $
  "SolverControl.solveAll: Not solvable with supported solvers."
solveAll wcs ((Solver solve):solvers) e =
  case filterCs wcs of
    ([],[])   -> return failCons
    ([],wcs') -> solveAll wcs' solvers e
    (cs,[])   -> solve cs e
    (cs,wcs') -> do e' <- solve cs e
                    solveAll wcs' solvers e'
```

Listing 115: Lösung aller "eingepackten" Constraints

Diese Funktion erhält drei Argumente: die heterogene Liste der zu lösenden Constraints, die Liste der von KiCS2 unterstützten Constraint-Solver und den durch die Constraints beschränkten nicht-deterministischen Ausdruck, der im Erfolgsfall - also der Lösung aller Constraints - weiter ausgewertet werden soll.
Falls nach dem "Ausprobieren" aller unterstützten Constraint-Solver weiterhin noch ungelöste Constraints übrig sind, so wird eine entsprechende Fehlermeldung zurückgegeben.
Solange die Liste der Solver nicht leer ist, versucht `solveAll` durch Aufruf von `filterCs` die passenden Constraints für den jeweiligen Solver aus der heterogenen Liste herauszusuchen. Diese Suche kann zu vier verschiedenen Ergebnissen führen:

1. Beide Ergebnislisten von `filterCs` sind leer: Dieser Fall sollte eigentlich niemals auftreten, da dies bedeutet, dass die heterogene Liste schon vor dem Aufruf von `solveAll` leer war, was wiederum bedeutet, dass ein `Guard`-Ausdruck mit einer leeren heterogenen Liste ausgewertet

worden ist. Ein solcher `Guard`-Ausdruck sollte normalerweise niemals erzeugt werden. Tritt dieser Fall dennoch auf, so wird ein `Fail`-Knoten zurückgegeben.

2. Die Liste des gesuchten Constraint-Typs ist leer, die heterogene Liste nicht: In diesem Fall wird durch einen rekursiven Aufruf von `solveAll` versucht, die übrigen Constraints mit dem nächsten Constraint-Solver aus der Liste der unterstützten Solver zu lösen.

3. Es wurden Constraints des gesuchten Typs gefunden und die Liste der noch verbliebenen `WrappedConstraint`s ist leer: In diesem Fall wird der entsprechende Constraint-Solver auf die gefundenen Constraints angewendet. Ein rekursiver Aufruf von `solveAll` ist nicht mehr erforderlich, da keine weiteren Constraints zum Lösen in der heterogenen Liste vorhanden sind.

4. Es wurden Constraints des gesuchten Typs gefunden und die Liste der noch verbliebenen `WrappedConstraint`s ist nicht leer: Wie im vorigen Fall wird der entsprechende Solver zur Lösung der herausgefilterten Constraints aufgerufen. Allerdings erfolgt dieses Mal ein rekursiver Aufruf von `solveAll` mit den übrigen `WrappedConstraint`s, den restlichen Solvern sowie dem nicht-deterministischen Ausdruck, der durch das Lösen der zuvor gefundenen Constraints erzeugt wurde.

Die Funktion `solveAll` wird während der KiCS2-Auswertung aufgerufen und zwar in dem Fall, dass ein `Guard`-Ausdruck mit `WrappedConstraint`s ausgewertet werden soll. Beispielhaft wird hier dieser Aufruf für die Implementierung der Tiefensuche in KiCS2 gezeigt. Zusätzlich wird die Auswertung der übrigen `Guard`-Ausdrücke mit angegeben:

```
...
dfsGuard _ (WrappedConstr wcs) e =
  solveAll wcs solvers e >>= dfs cont
dfsGuard _ cs e =
  solve cs e >>= \mbSltn -> case mbSltn of
    Nothing          -> mnil
    Just (reset, e') -> dfs cont e' |< reset
...
```

Listing 116: Lösung aller “eingepackten“ Constraints während der KiCS2-Tiefensuche

Wie man sieht, wird `solveAll` mit den `WrappedConstraints` `wcs` und dem durch sie beschränkten (nicht-)deterministischen Ausdruck `e` sowie der zuvor definierten Liste der unterstützten Constraint-Solver `solvers` aufgerufen. Der resultierende nicht-deterministische Ausdruck wird mit Hilfe des Tiefensuche-Algorithmus weiter ausgewertet.

Sonderfall: Curry-Bindungs-Constraints:
Zum Abschluss dieses Kapitels soll nun noch kurz darauf eingegangen werden, warum man die Curry-Bindungs-Constraints (definiert durch die Datentypen `Constraint` und `Constraints`) nicht auch in den Constraint-Wrapper “gepackt“ hat.

Auf den ersten Blick erscheint es durchaus sinnvoll, sie auch in den Constraint-Wrapper "einzupacken", da dann wirklich alle Constraint-Typen gleich behandelt werden könnten.
Möglich wäre dies, wenn man den Datentyp für die Curry-Bindungs-Constraints in seiner ursprünglichen Form belässt und `Guard`-Ausdrücke den folgenden Typ hätten (im Folgenden beispielhaft angegeben für `C_Bool`):

```
data C_Bool = ...
            | Guard_C_Bool [WrappedConstraint] C_Bool
```

Listing 117: Alternativer Typ für `Guard`-Ausdrücke

Das bedeutet, anstelle von `Constraints` würde jeder `Guard`-Ausdruck eine Liste von in den Constraint-Wrapper "eingepackten" Constraints enthalten. Auf diese Weise könnte man das Interface für `WrappableConstraint`s auch für die Curry-Bindungs-Constraints implementieren.

```
data Constraints
  = forall a . ValConstr ID a [Constraint]
  | StructConstr [Constraint]
 deriving (Show,Typeable)

instance WrappableConstraint Constraints
```

Listing 118: `WrappableConstraint`-Instanz für Curry-Bindungs-Constraints

Nun müsste man bei der Erzeugung von `Guard`-Ausdrücken für Curry-Bindungs-Constraints diese nur noch in ein `WrappedConstraint` "einpacken" sowie den KiCS2-internen Solver für diese Constraints über die Funktion `solveAll` ansteuerbar machen, also ihn zur Liste der von KiCS2 unterstützten Solver hinzufügen.

Diese Lösung hätte den Vorteil, dass man bei der Implementierung der KiCS2-Suchfunktionen die Constraints **aller** `Guard`-Ausdrücke durch `solveAll` lösen lassen könnte.
Problematisch wird das Wrapping der Curry-Bindungs-Constraints jedoch in den Fällen, in denen man diese Constraints mit anderen Constraint-Typen wie zum Beispiel den Finite-Domain-Constraints kombiniert.
In der bislang vorgestellten Implementierung wurden die Curry-Bindungs-Constraints in solchen Fällen im KiCS2-Auswertungsbaum "nach oben" verschoben. Auf diese Weise konnten sie vor den FD-Constraints gelöst werden und damit mögliche Bindungen der FD-Variablen noch vor dem Aufruf eines Solvers entsprechend aktualisiert werden. Dieses Problem ließe sich jedoch lösen, indem man versucht, den Solver für die Curry-Bindungs-Constraints beim Aufruf von `solveAll` als erstes anzuwenden. Anders formuliert: Der Solver für die Curry-Bindungs-Constraints müsste in der Liste der unterstützten Constraint-Solver der erste sein. Auf diese Weise stünden etwaige aktuelle Bindungen für die FD-Variablen noch rechtzeitig vor Aufruf der FD-Solver zur Verfügung.

Die Curry-Bindungs-Constraints bestimmen jedoch nicht nur die Bindungsentscheidungen für freie Variablen, sondern auch für `NarrowedChoices`. Und über solchen `NarrowedChoices` werden sämtliche FD-Constraints generiert, die eine freie Variable als Listen-Argument erhalten. Zur Erinnerung: Listen-Argumente von CLPFD-Constraint-Funktionen werden durch Aufruf von `ensureSpine` in eine sogenannte *spine*-Form überführt. Dabei werden die `Choices` für etwaige freie Listen-Variablen in `NarrowedChoices` transformiert.

Behandelt man die Curry-Bindungs-Constraints nun wie alle anderen Constraints und "packt" diese in den Constraint-Wrapper, so werden diese beim Einsammeln der `WrappedConstraint`s durch die Funktion `searchWrappedCs` zusammen mit den übrigen Constraints "nach unten" im KiCS2-Auswertungsbaum sortiert. Dies hat allerdings auch zur Folge, dass die Bindungs-Constraints für `NarrowedChoices` "nach unten" verschoben werden und damit auch unterhalb des zugehörigen `NarrowedChoice`-Knotens im Auswertungsbaum platziert werden. Die Konsequenz ist, dass Berechnungen von Ausdrücken, in denen FD-Constraints mit freien Variablen als Listen-Argument mit Unifikationsausdrücken über diesen freien Variablen kombiniert werden, nicht mehr terminieren. Da der `Guard`-Knoten mit den Bindungs-Constraints im KiCS2-Auswertungsbaum "unterhalb" der `NarrowedChoice`-Knoten liegt, werden die Bindungs-Constraints nicht mehr vor der Auswertung der `NarrowedChoice`-Knoten gelöst. Dies führt dazu, dass alle Pfade der `NarrowedChoices` ausgewertet werden anstatt nur diejenigen, die durch die Bindungsentscheidung vorgegeben werden. Deshalb terminiert die Berechnung in solchen Fällen nicht mehr.
Im Anhang D findet man auch noch einmal ein einfaches Beispiel, das diese Problematik verdeutlicht.

Lösen ließe sich dieses Problem nur, indem man die Curry-Bindungs-Constraints beim Einsammeln von den übrigen `WrappedConstraint`s trennt. Dafür müsste man diese allerdings aufwendig mittels `filterCs` herausfiltern. Da damit das ursprüngliche Ziel, alle Constraint-Typen gleich zu behandeln, ad absurdum geführt würde, erscheint es sinnvoller, die Curry-Bindungs-Constraints von vornherein anders zu behandeln als die übrigen Constraint-Typen und sie daher nicht in den Constraint-Wrapper zu "packen".

5. Evaluation

In diesem Kapitel soll die vorgestellte Erweiterung von KiCS2 um eine Finite-Domain-Constraint-Bibliothek und die Integration extern implementierter FD-Solver mit Hilfe einiger Benchmarks evaluiert werden.
Alle Benchmarks wurden auf einem PC mit Intel Core 2 Duo (3 GHz) Prozessor und 3 GB Arbeitsspeicher unter Ubuntu 12.04 ("Precise Pangolin") durchgeführt. Dabei wurde KiCS2 Version 0.2 mit dem Glasgow Haskell Compiler (GHC Version 7.4.1) ausgeführt. Mit Hilfe der `time`-Option von KiCS2 wurde jeweils die Ausführungszeit in Sekunden gemessen und dann der Mittelwert von drei Messdurchläufen berechnet. Falls ein Benchmark-Programm innerhalb von 10 Minuten kein Ergebnis geliefert hat, wird dies bei den Messergebnissen durch "n.a." kenntlich gemacht.
Zunächst wurde getestet, inwieweit sich das "nach oben" Verschieben von `Guard`-Ausdrücken mit Bindungs-Constraints im KiCS2-Auswertungsbaum sowie das zusätzliche Ablaufen dieses Baumes zur Suche nach `WrappedConstraints` mittels `searchWrappedCs` auf die Auswertung von Ausdrücken auswirkt, die keine Finite-Domain-Constraints enthalten. Dazu wurden die Benchmarks für die Unifikation, die KiCS2 zur Verfügung stellt (*UnificationBench.curry*), einmal mit und einmal ohne die CLPFD-Erweiterung ausgeführt. Zu den Benchmark-Programmen gehören `last` (Berechnung des letzten Elements einer Liste), `grep` (Matching eines Wortes mit einem regulären Ausdruck), `halfPeano` (Halbierung einer Peano-Zahl), `varInExp` (Suche nach einer Variable in einem arithmetischen Ausdruck), `simplify` (Vereinfachung eines arithmetischen Ausdrucks), `palindrome` (Überprüft, ob eine Liste ein Palindrom ist) und `horseMan` (Lösung einer Gleichung bezüglich der Köpfe und Beine von Pferden und Menschen):

Benchmark	KiCS2	KiCS2 + CLPFD
`last`	1.18	1.18
`grep`	1.14	1.24
`halfPeano`	53.17	54.1
`varInExp`	2.01	n.a.
`simplify`	54.56	95.75
`palindrome`	64.3	85.93
`horseMan`	11.58	14.79

Abbildung 6: Benchmarks: Unifikation in KiCS2 ohne und mit CLPFD-Erweiterung

Wie man an den Ergebnissen sieht, benötigt die erweiterte KiCS2-Version in fast allen Fällen länger als die ursprüngliche Version. Dies hängt vermutlich in erster Linie mit dem "Hochziehen" der `Guard`-Ausdrücke mit Bindungs-Constraints im KiCS2-Auswertungsbaum zusammen. Besonders extrem wirkt sich dies auf den `varInExp`-Benchmark aus, bei dem versucht wird, die einzige Variable in einem tief verschachtelten arithmetischen Ausdruck (25.000 Knoten) mit Hilfe der Unifikation zu finden.
Auch die Vereinfachung eines arithmetischen Ausdrucks mit 4003 Knoten benötigt fast doppelt so lange, wenn man die erweiterte KiCS2-Version verwendet. Bei den übrigen Benchmarks bleibt der Overhead für die CLPFD-Erweiterung hingegen im Rahmen.

In einem weiteren Benchmark wurde die Performance der Finite-Domain-Solver anhand des N-Damen-Problems verglichen. Betrachtet wurden dabei eine generate&test-Variante des N-Damen-Problems ohne FD-Constraints (siehe Anhang E), die Lösung des CLPFD-Modells (vergleiche Listing 59) mit den beiden MCP-FD-Solvern, wobei jeweils einmal die `InOrder`- und einmal die `FirstFail`-Labeling-Strategie getestet wurden, und der direkte Aufruf des Gecode-Solvers ohne KiCS2 mit dem im Grundlagenkapitel vorgestellten MCP-Modell (vergleiche Listing 34).

n	gen. & test	Overton		Gecode		Gecode (direkt)
		`InOrder`	`FirstFail`	`InOrder`	`FirstFail`	`FirstFail`
4	0.04	<0.01	<0.01	<0.01	<0.01	<0.01
5	0.66	0.02	0.02	0.02	0.02	0.01
6	11.12	0.03	0.03	0.02	0.02	0.02
7	225.99	0.13	0.10	0.07	0.04	0.03
8	n.a.	0.48	0.38	0.18	0.09	0.07
9	n.a.	2.28	1.82	0.74	0.29	0.20
10	n.a.	10.16	8.45	2.64	0.86	0.65
11	n.a.	51.62	43.14	13.16	3.58	2.80
12	n.a.	301.06	248.87	83.95	18.91	14.26

Abbildung 7: Benchmarks: Performance der Solver im Vergleich

Die Benchmarks zeigen, dass die generate&test-Implementierung des N-Damen-Problems für größere n nicht mit den FD-Solvern konkurrieren kann. Bereits die Berechnung des 8-Damen-Problems benötigt länger als zehn Minuten. Die MCP-FD-Solver lösen das gleiche Problem in unter einer Sekunde. Auch für größere n-Werte berechnen beide Solver die Lösungen in einer akzeptablen Zeit. Wie zu erwarten war, ist der auf C++-basierende Gecode-Solver allerdings erheblich schneller als der in Haskell realisierte Overton-Solver. Durch Einsatz einer vorteilhaften Labeling-Strategie wie `FirstFail`, bei der die Labeling-Variablen mit dem am weitesten eingeschränkten Wertebereich bevorzugt werden, können die Zeiten für beide MCP-Solver noch einmal erheblich verbessert werden.
Vergleicht man den direkten Aufruf des Gecode-Solvers über das MCP-Framework mit dem Aufruf dieses Solvers über KiCS2, so zeigt sich, dass die Performance fast identisch ist. Erst bei der Lösung größerer Modelle wirkt sich der zusätzliche Übersetzungsschritt, der in KiCS2 durchgeführt werden muss, auf die Ausführungszeiten aus.

6. Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde die KiCS2-Curry-Implementierung um die Möglichkeit zur Programmierung mit Finite-Domain-Constraints erweitert.
Dazu wurde eine einfache Bibliothek zur Modellierung von Finite-Domain-Constraint-Problemen in Curry implementiert.
Zur Lösung eines solchen Modells wurden die beiden FD-Solver des Monadic-Constraint-Programming-Frameworks - der direkt in Haskell implementierte Overton-Solver sowie der auf C++-basierende Gecode-Solver - in KiCS2 integriert. Dazu wurden die FD-Constraints in semantisch äquivalente MCP-FD-Constraints übersetzt und die resultierenden MCP-Modelle durch Aufruf eines MCP-Solvers gelöst.
In einem zweiten Entwicklungsschritt wurde dann von den speziellen FD-Constraints und -Solvern abstrahiert. So wurde eine generische Schnittstelle für Constraints entwickelt, deren Implementierung es ermöglicht, KiCS2 um weitere Constraint-Typen zu erweitern. Des Weiteren wurde ein allgemeines Interface zur Integration von FD-Solvern in KiCS2 eingeführt. Diese Schnittstellen wurden für die FD-Constraints und die FD-Solver des MCP-Frameworks beispielhaft implementiert.
Mit Hilfe von Benchmarks wurde gezeigt, dass die Verwendung der CLPFD-Bibliothek zur Modellierung von FD-Constraint-Problemen erheblich effizienter ist als ein entsprechendes Problem mit Hilfe des generate&test-Verfahrens zu modellieren. Außerdem erreicht man mit dieser Bibliothek trotz des zusätzlichen Übersetzungsschritts eine annähernd mit der direkten Lösung durch das MCP-Framework vergleichbare Performance.

Für die Zukunft wäre es sicherlich interessant, weitere Constraint-Typen und (FD-)Solver in KiCS2 zu integrieren und auf diese Weise die hier vorgestellten Schnittstellen weiter zu testen und zu verbessern.
Außerdem könnte man beim Aufruf der MCP-Solver abhängig von der jeweils verwendeten KiCS2-Auswertungsstrategie eine dazu passende MCP-Suchstrategie und einen geeigneten MCP-Such-Transformer einsetzen, anstatt in jedem Fall die Tiefensuche und den Identitäts-Such-Transformer zu benutzen.
Des Weiteren könnte man versuchen, die FD-Constraints mit der eingekapselten Suche von KiCS2 zu kombinieren, so dass man eigene Suchstrategien über FD-Constraint-Modellen definieren könnte. Eine Lösung speziell für die MCP-Solver könnte in diesem Fall sein, die Suchstrategien mit Hilfe des MCP-Frameworks zu realisieren, das Interfaces zur Entwicklung eigener Suchstrategien und -Transformer zur Verfügung stellt.

A. Installation und Benutzung der KiCS2-CLPFD-Bibliothek

A.1. Installation

Um die KiCS2-Finite-Domain-Constraint-Bibliothek nutzen zu können, muss man zunächst eine KiCS2-Distribution, die diese Bibliothek beinhaltet, von der Homepage des Lehrstuhls für Programmiersprachen und Übersetzerkonstruktion herunterladen und installieren (Link: http://www-ps.informatik.uni-kiel.de/kics2/download/). Alternativ kann man auch eine aktuelle Version aus dem KiCS2-Repository herunterladen. Weitere Hinweise und eine Installationsanleitung findet man unter http://www-ps.informatik.uni-kiel.de/kics2/repos/ oder auch in dem entpackten KiCS2-tarfile.

Nun ist es noch erforderlich, das Monadic-Constraint-Programming-Framework, das als Solver-Backend für die CLPFD-Bibliothek verwendet wird, zu installieren. Um auch den Gecode-Solver verwenden zu können, sollte man allerdings zunächst die entsprechende C++-Constraint-Solver-Bibliothek installieren. Dazu lädt man von http://www.gecode.org/download.html die gepackten Quelldateien der **Gecode Version 3.1.0** herunter. Es ist wichtig, dass man genau diese Version installiert, da das Monadic-Constraint-Programming-Framework keine andere Version unterstützt. Zunächst entpackt man die Quelldateien mit

```
tar xzvf gecode-3.1.0.tar.gz
```

Dann ruft man das Konfigurationsskript auf:

```
cd gecode-3.1.0/
./configure
```

Schließlich kann man die Quelldateien kompilieren und danach Gecode installieren:

```
make
make install
```

Nach der Installation sollte man die Gecode-Bibliotheken noch zum *library path* hinzufügen, damit die (Beispiel-)Programme richtig gelinkt werden können.
Weitere Hinweise zu Gecode kann man unter http://www.gecode.org/documentation.html in der Anleitung *Modeling and Programming with Gecode* finden.

Nun kann man das Monadic-Constraint-Programming-Framework installieren. Dazu lädt man von Hackage DB (http://hackage.haskell.org/package/monadiccp-0.7.4) die neueste Version 0.7.4 (Stand: Juli 2012) des Frameworks herunter. Nachdem man die Source-Dateien entpackt hat (`tar xzvf monadiccp-0.7.4.tar.gz`), kann man die Konfiguration durchführen. Dabei muss man beachten, dass das Konfigurationsflag für die Installation der MCP-Gecode-Quelldateien gesetzt und der Pfad zu den Bibliotheken und Header-Dateien der Gecode-Installation angegeben werden muss:

```
cd monadiccp-0.7.4
runhaskell Setup.hs configure --extra-include-dirs=
  /usr/local/include/ --extra-lib-dirs=/usr/local/lib/
  --flags="RuntimeGecode"
```

Nach erfolgreicher Konfiguration kann man die MCP-Pakete schließlich bauen und installieren:

```
runhaskell Setup.hs build
runhaskell Setup.hs install
```

Danach sollte man das Framework benutzen können. Zum Testen kann man nun eines der Beispielmodelle kompilieren und ausführen. Mit den folgenden Befehlen kann man beispielsweise das 4-Damen-Problem vom Gecode-Solver lösen lassen:

```
cd examples
ghc --make Queens.hs
./Queens gecode_run 4
```

Zuerst wird also der Solver-Typ angegeben (`gecode_run`) und dann etwaige Parameter für das Constraint-Problem.

A.2. Benutzung

Bei der Modellierung von Finite-Domain-Constraints mit der CLPFD-Bibliothek sollte man einige Dinge berücksichtigen:

1. Ein CLPFD-Modell sollte im Allgemeinen folgende Form haben:

   ```
   [<unification_constraints>] & <domain_constraints> &
     <fd_constraints> & <labeling_constraint>
   ```

   Als erstes gibt man etwaige Curry-Gleichheits-Constraints für die Constraint-Variablen an, dann legt man einen Wertebereich für die Constraint-Variablen fest, danach folgen beliebige FD-Constraints über den eingeführten Constraint-Variablen und schließlich initiiert man deren Labeling durch Angabe eines "Labeling-Constraints".

2. Die Curry-Gleichheits-Constraints sind optional und können gegebenenfalls auch an anderer Stelle spezifiziert werden. Sie zuerst anzugeben, kann jedoch den Lösungsvorgang beschleunigen, da diese Constraints während der Auswertung dann KiCS2-intern nicht verschoben werden müssen.

3. Für das Labeling kann - falls gewünscht - mittels des `labelingWith`-Constraints eine spezielle Labeling-Strategie angegeben werden. Verwendet man stattdessen das `labeling`-Constraint, so wird das Labeling der Constraint-Variablen in der gegebenen Reihefolge durchgeführt. Verzichtet man völlig auf ein "Labeling"-Constraint, so erhält man eine Fehlermeldung.

4. Weiterhin ist zu beachten, dass für sämtliche Labeling-Variablen auch ein Wertebereich festgelegt wird. Bei Nichtberücksichtigung wird ebenfalls eine Fehlermeldung ausgegeben.

B. Auszüge aus dem Monadic-Constraint-Programming-Framework

```
data Expr t c b =
    Term t
  | ExprHole Int
  | Const Integer
  | Plus (Expr t c b) (Expr t c b)
  | Minus (Expr t c b) (Expr t c b)
  | Mult (Expr t c b) (Expr t c b)
  | Div (Expr t c b) (Expr t c b)
  | Mod (Expr t c b) (Expr t c b)
  | Abs (Expr t c b)
  | At (ColExpr t c b) (Expr t c b)
  | Fold (Expr t c b -> Expr t c b -> Expr t c b) (Expr t c b) (ColExpr t c b)
  | Cond (BoolExpr t c b) (Expr t c b) (Expr t c b)
  | ColSize (ColExpr t c b)
  | Channel (BoolExpr t c b)
```

Listing 119: Repräsentation von Integer-Ausdrücken

```
data ColExpr t c b =
    ColTerm c
  | ColList [Expr t c b]
  | ColRange (Expr t c b) (Expr t c b)
  | ColMap (Expr t c b -> Expr t c b) (ColExpr t c b)
-- ColSlice f n c -> c[f(0)..f(n-1)]
  | ColSlice (Expr t c b -> Expr t c b) (Expr t c b) (ColExpr t c b)
  | ColCat (ColExpr t c b) (ColExpr t c b)
```

Listing 120: MCP-Collections

```
data BoolExpr t c b =
    BoolTerm b
  | BoolConst Bool
  | BoolAnd (BoolExpr t c b) (BoolExpr t c b)
  | BoolOr (BoolExpr t c b) (BoolExpr t c b)
  | BoolNot (BoolExpr t c b)
  | BoolCond (BoolExpr t c b) (BoolExpr t c b) (BoolExpr t c b)
  | Rel (Expr t c b) ExprRel (Expr t c b)
  | BoolAll (Expr t c b -> BoolExpr t c b) (ColExpr t c b)
  | BoolAny (Expr t c b -> BoolExpr t c b) (ColExpr t c b)
  | ColEqual (ColExpr t c b) (ColExpr t c b)
  | BoolEqual (BoolExpr t c b) (BoolExpr t c b)
  | AllDiff Bool (ColExpr t c b)
  | Sorted Bool (ColExpr t c b)
  | Dom (Expr t c b) (ColExpr t c b)
```

Listing 121: MCP-FD-Constraints

```
-- convertible to expressions:
class ToExpr tt cc bb t where
  toExpr :: t -> Expr tt cc bb

-- convertible to collection-expressions:
class ToColExpr tt cc bb c where
  toColExpr :: c -> ColExpr tt cc bb

-- convertible to boolean expressions:
class ToBoolExpr tt cc bb b where
  toBoolExpr :: b -> BoolExpr tt cc bb

(@+), (@-), (@*), (@/), (@%) :: (Eq t, Eq c, Eq b, ToExpr t c b p, ToExpr t c b q)
                              => p -> q -> Expr t c b
a @+ b = simplify $ (toExpr a) `Plus` (toExpr b)
a @- b = simplify $ (toExpr a) `Minus` (toExpr b)
a @* b = simplify $ (toExpr a) `Mult` (toExpr b)
a @/ b = simplify $ (toExpr a) `Div` (toExpr b)
a @% b = simplify $ (toExpr a) `Mod` (toExpr b)

class (Eq tt, Eq cc, Eq bb) => ExprClass tt cc bb a where
  (@=)  :: a -> a -> BoolExpr tt cc bb
  (@/=) :: a -> a -> BoolExpr tt cc bb
  a @/= b = boolSimplify $ BoolNot $ a @= b

class (Eq tt, Eq cc, Eq bb) => ExprRange tt cc bb r where
  (@:)  :: Expr tt cc bb -> r -> BoolExpr tt cc bb

(!)   :: (Eq t, Eq c, Eq b) => ColExpr t c b -> Expr t c b -> Expr t c b
(@!!) :: (Eq t, Eq c, Eq b) => ColExpr t c b -> Integer -> Expr t c b
(@..) :: (Eq t, Eq c, Eq b) => Expr t c b -> Expr t c b -> ColExpr t c b
(@++) :: (Eq t, Eq c, Eq b) => ColExpr t c b -> ColExpr t c b -> ColExpr t c b

(@?) :: (Eq t, Eq c, Eq b) => BoolExpr t c b -> (Expr t c b, Expr t c b)
     -> Expr t c b
c @? (t,f) = simplify $ Cond c t f

(@??) :: (Eq t, Eq c, Eq b) => BoolExpr t c b -> (BoolExpr t c b, BoolExpr t c b)
      -> BoolExpr t c b
c @?? (t,f) = boolSimplify $ BoolCond c t f

c!p     = simplify $ At c p
c @!! p = simplify $ At c (Const p)
a @.. b = colSimplify $ ColRange (toExpr a) (toExpr b)
a @++ b = colSimplify $ ColCat (toColExpr a) (toColExpr b)

size :: (Eq t, Eq c, Eq b) => ColExpr t c b -> Expr t c b
size a = simplify $ ColSize a

xfold :: (Eq t, Eq c, Eq b) => (Expr t c b -> Expr t c b -> Expr t c b)
```

```
      -> Expr t c b -> ColExpr t c b -> Expr t c b
xfold f i c = simplify $ Fold (\a b -> f a b) i c

xsum :: (Num (Expr t c b), Eq t, Eq c, Eq b) => ColExpr t c b -> Expr t c b
xsum c = xfold (+) (Const 0) c

list :: (Eq t, Eq c, Eq b) => [Expr t c b] -> ColExpr t c b
list x = colSimplify $ ColList x

xhead :: (Eq t, Eq c, Eq b, ToColExpr t c b p) => p -> Expr t c b
xhead c = simplify $ At (toColExpr c) (Const 0)

xtail :: (Eq t, Eq c, Eq b, ToColExpr t c b p) => p -> ColExpr t c b
xtail c = let cc = toColExpr c in colSimplify $
  ColSlice (\x -> simplify (x `Plus` (Const 1)))
           (simplify $ (size cc) `Minus` (Const 1)) cc

slice :: (Eq t, Eq c, Eq b) => ColExpr t c b -> ColExpr t c b -> ColExpr t c b
slice c p = case (c,p) of
  (_,ColRange l h) -> colSimplify $
    ColSlice (\x -> simplify (l `Plus` x))
             (simplify $ Const 1 `Plus` (simplify $ h `Minus` l)) c
  (_,ColMap f (ColRange l h)) -> colSimplify $
    ColSlice (\i -> simplify $ f $ simplify (l `Plus` i))
             (simplify $ Const 1 `Plus` (simplify $ h `Minus` l)) c
  (_,ColSlice f n c2) -> colSimplify $
    ColSlice (\i -> simplify $ c2 `At` (f i)) n c
  _ -> xmap (\i -> simplify $ c `At` i) p

xmap :: (Eq t, Eq c, Eq b) => (Expr t c b -> Expr t c b) -> ColExpr t c b
     -> ColExpr t c b
xmap f c = colSimplify $ ColMap f c

loopall :: (Eq t, Eq c, Eq b) => (Expr t c b,Expr t c b)
        -> (Expr t c b -> BoolExpr t c b) -> BoolExpr t c b
loopall (l,h) f = boolSimplify $ BoolAll f $ colSimplify $ ColRange l h

loopany :: (Eq t, Eq c, Eq b) => (Expr t c b,Expr t c b)
        -> (Expr t c b -> BoolExpr t c b) -> BoolExpr t c b
loopany (l,h) f = boolSimplify $ BoolAny f $ colSimplify $ ColRange l h

forall :: (Eq t, Eq c, Eq b) => (ColExpr t c b)
       -> (Expr t c b -> BoolExpr t c b) -> BoolExpr t c b
forall c f = boolSimplify $ BoolAll f c

forany :: (Eq t, Eq c, Eq b) => (ColExpr t c b)
       -> (Expr t c b -> BoolExpr t c b) -> BoolExpr t c b
forany c f = boolSimplify $ BoolAny f c

channel :: (Eq t, Eq c, Eq b) => BoolExpr t c b -> Expr t c b
```

```
channel = simplify . Channel

inv :: (Eq t, Eq c, Eq b, ToBoolExpr t c b p) => p -> BoolExpr t c b

a @|| b = boolSimplify $ BoolOr (toBoolExpr a) (toBoolExpr b)
a @&& b = boolSimplify $ BoolAnd (toBoolExpr a) (toBoolExpr b)
inv a = boolSimplify $ BoolNot (toBoolExpr a)

(@<), (@>), (@<=), (@>=) ::  (Eq t, Eq c, Eq b) => Expr t c b -> Expr t c b
                          -> BoolExpr t c b

a @< b = boolSimplify $ Rel a ERLess b
a @> b = boolSimplify $ Rel b ERLess a
a @<= b = boolSimplify $ Rel a ERLess (simplify $ b `Plus` (Const 1))
a @>= b = boolSimplify $ Rel b ERLess (simplify $ a `Plus` (Const 1))

sorted c = boolSimplify $ Sorted False c
sSorted c = boolSimplify $ Sorted True c
allDiff c = boolSimplify $ AllDiff False c
allDiffD c = boolSimplify $ AllDiff True c
```

Listing 122: MCP-FD-Constraint-Konstruktorfunktionen

```
class (Solver s, Term s (FDIntTerm s), Term s (FDBoolTerm s),
 Eq (FDBoolSpecType s), Ord (FDBoolSpecType s), Enum (FDBoolSpecType s),
 Bounded (FDBoolSpecType s), Show (FDBoolSpecType s), Eq (FDIntSpecType s),
 Ord (FDIntSpecType s), Enum (FDIntSpecType s), Bounded (FDIntSpecType s),
 Show (FDIntSpecType s), Eq (FDColSpecType s), Ord (FDColSpecType s),
 Enum (FDColSpecType s), Bounded (FDColSpecType s), Show (FDColSpecType s),
 Show (FDIntSpec s), Show (FDColSpec s), Show (FDBoolSpec s)) => FDSolver s where

  -- term types
  type FDIntTerm s    :: * -- a Term of s, representing Integer variables
  type FDBoolTerm s   :: * -- a Term of s, representing Bool variables

  -- spec types
  type FDIntSpec s    :: * -- a type specifying an Integer expression; should at
                           -- least support constant Integer's and FDIntTerm's
  type FDBoolSpec s   :: * -- a type specifying a Bool expression; should at least
                           -- support constant Bool's and FDBoolTerm's
  type FDColSpec s    :: * -- a type specifying a Integer array expression; should
                           -- at least support lists of Int's and lists of
                           -- IntTerm's

  -- spec type type
  type FDIntSpecType s :: *  -- a type specifying the type of an FDIntSpec s,
                             -- in case there is more than one
  type FDBoolSpecType s :: * -- a type specifying the type of an FDBoolSpec s,
                             -- in case there is more than one
  type FDColSpecType s :: *  -- a type specifying the type of an FDColSpec s,
                             -- in case there is more than one
```

```
-- constructors for specifiers
fdIntSpec_const     :: EGPar         -> (FDIntSpecType s, s (FDIntSpec s))
fdBoolSpec_const    :: EGBoolPar     -> (FDBoolSpecType s, s (FDBoolSpec s))
fdColSpec_const     :: EGColPar      -> (FDColSpecType s, s (FDColSpec s))
fdColSpec_list      :: [FDIntSpec s] -> (FDColSpecType s, s (FDColSpec s))
fdIntSpec_term      :: FDIntTerm s   -> (FDIntSpecType s, s (FDIntSpec s))
fdBoolSpec_term     :: FDBoolTerm s  -> (FDBoolSpecType s, s (FDBoolSpec s))
fdColSpec_size      :: EGPar         -> (FDColSpecType s, s (FDColSpec s))
fdIntVarSpec        :: FDIntSpec s   -> s (Maybe (FDIntTerm s))
fdBoolVarSpec       :: FDBoolSpec s  -> s (Maybe (FDBoolTerm s))

-- function to inform about allowed types for nodes
fdTypeReqBool :: s (EGEdge -> [(EGVarId,FDBoolSpecTypeSet s)])
fdTypeReqInt ::  s (EGEdge -> [(EGVarId,FDIntSpecTypeSet s)])
fdTypeReqCol ::  s (EGEdge -> [(EGVarId,FDColSpecTypeSet s)])
fdTypeReqBool = return (\(EGEdge { egeLinks = EGTypeData { boolData = l } })
  -> map (\x -> (x,Set.fromList [minBound..maxBound])) l)
fdTypeReqInt = return (\(EGEdge { egeLinks = EGTypeData { intData = l } })
  -> map (\x -> (x,Set.fromList [minBound..maxBound])) l)
fdTypeReqCol = return (\(EGEdge { egeLinks = EGTypeData { colData = l } })
  -> map (\x -> (x,Set.fromList [minBound..maxBound])) l)

fdTypeVarInt  :: s (Set (FDIntSpecType s))
fdTypeVarBool :: s (Set (FDBoolSpecType s))
fdTypeVarInt  = return $ Set.singleton maxBound
fdTypeVarBool = return $ Set.singleton maxBound

-- rating functions for specification of terms
fdSpecify :: Mixin (SpecFn s)
fdSpecify = mixinId

-- inspect collections
fdColInspect :: FDColSpec s -> s [FDIntTerm s]

-- function to request processing an edge in a graph
fdProcess :: Mixin (EGConstraintSpec -> FDSpecInfo s -> FDInstance s ())

-- add equality constraints
fdEqualBool :: FDBoolSpec s -> FDBoolSpec s -> FDInstance s ()
fdEqualInt :: FDIntSpec s -> FDIntSpec s -> FDInstance s ()
fdEqualCol :: FDColSpec s -> FDColSpec s -> FDInstance s ()

fdConstrainIntTerm :: FDIntTerm s -> Integer -> s (Constraint s)
fdSplitIntDomain :: FDIntTerm s -> s ([Constraint s],Bool)
fdSplitBoolDomain :: FDBoolTerm s -> s ([Constraint s],Bool)
```

Listing 123: `FDSolver`-Typklasse

```
class (Solver s, Term s t, Show (TermBaseType s t)) => EnumTerm s t where
  type TermBaseType s t :: *

  getDomainSize :: t -> s (Int)
  getDomain :: t -> s [TermBaseType s t]
  setValue :: t -> TermBaseType s t -> s [Constraint s]
  splitDomain :: t -> s ([[Constraint s]],Bool)
  splitDomains :: [t] -> s ([[Constraint s]],[t])
  getValue :: t -> s (Maybe (TermBaseType s t))
  defaultOrder :: [t] -> s [t]
  enumerator :: (MonadTree m, TreeSolver m ~ s) => Maybe ([t] -> m ())

  getDomainSize x = do
    r <- getDomain x
    return $ length r

  getValue x = do
    d <- getDomain x
    return $ case d of
      [v] -> Just v
      _ -> Nothing
  splitDomain x = do
    d <- getDomain x
    case d of
      [] ->  return ([],True)
      [_] -> return ([[]],True)
      _ ->   do
        rr <- mapM (setValue x) d
        return (rr,True)

  splitDomains [] = return ([[]],[])
  splitDomains (a@(x:b)) = do
    s <- getDomainSize x
    if s==0
      then return ([],[])
      else if s==1
        then splitDomains b
        else do
          (r,v) <- splitDomain x
          if v
            then return (r,b)
            else return (r,a)

  defaultOrder = firstFail
  enumerator = Nothing
```

Listing 124: `EnumTerm`-Typklasse

```
labelling :: (MonadTree m, TreeSolver m ~ s, EnumTerm s t)
          => ([t] -> s [t]) -> [t] -> m ()
labelling _ [] = true
labelling o l = label $ do
  ll <- o l
  (cl,c) <- splitDomains ll
  let ml = map (\l -> foldr (/\) true $ map addC l) cl
  return $ do
    levelList ml
    labelling return c

levelList :: (Solver s, MonadTree m, TreeSolver m ~ s) => [m ()] -> m ()
levelList [] = false
levelList [a] = a
levelList l = let len     = length l
                  (p1,p2) = splitAt (len `div` 2) l
              in (levelList p1) \/ (levelList p2)
```

Listing 125: MCP-`labelling`-Funktion

```
assignment :: (EnumTerm s t, MonadTree m, TreeSolver m ~ s)
           => t -> m (TermBaseType s t)
assignment q = label $ getValue q >>=
  \y -> (case y of Just x -> return $ return x; _ -> return false)

assignments :: (EnumTerm s t, MonadTree m, TreeSolver m ~ s)
            => [t] -> m [TermBaseType s t]
assignments = mapM assignment
```

Listing 126: MCP-`assignments`-Funktion

```
firstFail :: EnumTerm s t => [t] -> s [t]
firstFail qs = do ds <- mapM getDomainSize qs
                  return [ q | (d,q) <- zip ds qs
                             , then sortWith by d ]

inOrder :: EnumTerm s t => [t] -> s [t]
inOrder = return

middleOut :: EnumTerm s t => [t] -> s [t]
middleOut l = let n = (length l) `div` 2 in
              return $ interleave (drop n l) (reverse $ take n l)

endsOut :: EnumTerm s t => [t] -> s [t]
endsOut  l = let n = (length l) `div` 2 in
             return $ interleave (reverse $ drop n l) (take n l)

interleave []     ys = ys
interleave (x:xs) ys = x:interleave ys xs
```

Listing 127: MCP-Labeling-Strategien

```
colList :: (Constraint s ~ Either Model q, MonadTree m, TreeSolver m ~ s)
        => ModelCol -> Int -> m [ModelInt]
colList col len = do addM $ (Sugar.@=) (size col) (asExpr len)
                     return $ map (\i -> col!cte i) [0..len-1]

getColItems :: FDSolver s => ModelCol -> FDColSpecType s
            -> FDInstance s [FDIntTerm s]
getColItems c tp = do [cc] <- getColTerm [c] tp
                      lst <- liftFD $ fdColInspect cc
                      return lst

getColTerm :: FDSolver s => [ModelCol] -> FDColSpecType s
           -> FDInstance s [FDColSpec s]
getColTerm m tp = do
  s <- get
  put $ s { fdsForceCol = m++(fdsForceCol s) }
  commit
  s2 <- get
  let ids = map (\x -> decompColLookup (fdsDecomp s2) x) m
  specs <- mapM (\(Just id) -> getColSpec_ id (Set.singleton tp)) ids
  return $ map (snd . myFromJust ("getColTerm(tp="++(show tp)++")")) specs
```

Listing 128: MCP-Hilfsfunktionen zur Implementierung von `labelWith`

C. MCP-Solver-Implementierung

Wie schon im Kapitel 4.3.2 angedeutet, verwendet man zur Übersetzung der Constraints in die MCP-FD-Modellierungssprache unterschiedliche Implementierungen für die beiden Constraint-Solver. Genauer gesagt, die Implementierung variiert für die beiden Solver hinsichtlich der Übersetzung von FD-Listen in MCP-Collections. Der Gecode-Solver erwartet, dass man für jede FD-Liste eine MCP-Collection-Variable einführt und etwaigen konstanten Werten in dieser Liste mit Hilfe der `(@!!)`-Funktion des `(@=)`-Constraints eine Position in der durch die Variable repräsentierten Collection zuweist (vergleiche hierzu die Funktion `newColCs`). Der Overton-Solver kann hingegen auch mit konstanten MCP-Collections, also mit Listen vom Typ `ModelInt` arbeiten.
Um neue MCP-Collection-Variablen für den Gecode-Solver einführen zu können, ist es notwendig, FD-Listen eindeutig identifizieren zu können. Nur auf diese Weise kann gewährleistet werden, dass bei der Übersetzung die gleiche FD-Liste auf die gleiche MCP-Collection-Variable abgebildet wird. Daher wird der unten angegebene Datentyp zur Repräsentation von FD-Listen eingeführt. Für diese FD-Listen werden dann bei der Übersetzung nach dem gleichen Prinzip, das auch schon zur Übersetzung von FD-Variablen angewandt wurde, neue MCP-Collection-Variablen eingeführt (siehe dazu die Funktionen `translateGecodeList` und `newColVar`).

```
-- representation of lists of fd terms:
-- ID to identify a specific fd list is necessary for translating
-- fd lists into mcp collections for the Gecode solver
data FDList a = FDList ID [a]

-- helper function to convert curry integer lists to lists of fd terms
toFDList :: Constrainable a b => ID -> CP.OP_List a -> FDList b
toFDList i vs = FDList i (toFDList' vs)
  where
   toFDList' CP.OP_List        = []
   toFDList' (CP.OP_Cons v vs) = toCsExpr v : toFDList' vs
```

Listing 129: Einführung eines Identifiers für FD-Listen

```
data FDConstraint
  = FDRel RelOp (FDTerm Int) (FDTerm Int)
  | FDArith ArithOp (FDTerm Int) (FDTerm Int) (FDTerm Int)
  | FDSum (FDList (FDTerm Int)) (FDTerm Int)
  | FDAllDifferent (FDList (FDTerm Int))
  | FDDomain (FDList (FDTerm Int)) (FDTerm Int) (FDTerm Int)
  | FDLabeling LabelingStrategy (FDList (FDTerm Int)) ID
 deriving (Eq,Show,Typeable)
```

Listing 130: Angepasste `FDConstraints`

MCPSolver-Modul:

```
{-# LANGUAGE TypeFamilies, FlexibleContexts #-}

module MCPSolver where

import Types
import FDData
import ExternalSolver
import qualified Curry_Prelude as CP

import Data.Expr.Sugar
import Control.CP.ComposableTransformers (solve)
import Control.CP.SearchTree (addC, Tree (..),MonadTree(..))
import Control.CP.EnumTerm
import Control.CP.FD.Model
import Control.CP.FD.FD (FDInstance, FDSolver, getColItems)
import Control.CP.FD.Interface (colList)
import Control.CP.FD.Solvers
import Control.CP.FD.Gecode.Common (GecodeWrappedSolver)
import Control.CP.FD.Gecode.Runtime (RuntimeGecodeSolver)
import Control.CP.FD.OvertonFD.OvertonFD
import Control.CP.FD.OvertonFD.Sugar

import qualified Data.Map as Map
import Control.Monad.State
import Data.Maybe (fromJust)
import Data.List ((\\))

-- ---------------------------------------------------------------------------
-- WrappableConstraint instance for FDConstraint
-- ---------------------------------------------------------------------------
instance WrappableConstraint FDConstraint where
  updateVars = updateFDConstr updateFDVar

-- ---------------------------------------------------------------------------
-- ExternalFDSolver instance for MCP Solvers
-- ---------------------------------------------------------------------------
instance ExternalFDSolver MCPSolver FDConstraint where
  newtype SolverModel MCPSolver FDConstraint = ModelWrapper [Model]

  -- |Type for storing labeling information for the MCP solvers:
  -- @labelVars    - labeling variables in original representation
  -- @domainVars   - list of fd variables, for which a domain was defined
  --                 necessary to check, whether a domain was defined for the
  --                 labeling variables
  -- @mcpLabelVars - labeling variables translated into corresponding
  --                 MCP representation
  -- @labelID      - fresh ID, necessary for constructing choices over solutions,
  --                 when transforming solver solutions into binding constraints
  -- @strategy     - labeling strategy
```

```
  data LabelInfo MCPSolver FDConstraint =
    Info { labelVars    :: Maybe (FDList (FDTerm Int))
         , domainVars   :: [FDTerm Int]
         , mcpLabelVars :: Maybe ModelCol
         , labelID      :: Maybe ID
         , strategy     :: Maybe LabelingStrategy }

  newtype Solutions MCPSolver FDConstraint =
    SolWrapper (SolutionInfo CP.C_Int (FDTerm Int))

  translate Overton fdCs = translateOverton fdCs
  translate Gecode  fdCs = translateGecode fdCs

  solveWith = solveWithMCP

  makeCsSolutions _ (SolWrapper solutions) e = bindSolutions solutions e

-- type synonyms for easier access to associated types
type MCPModel     = SolverModel MCPSolver FDConstraint
type MCPLabelInfo = LabelInfo MCPSolver FDConstraint
type MCPSolution  = Solutions MCPSolver FDConstraint

-- ----------------------------------------------------------------------------
-- Translation to MCP model
-- ----------------------------------------------------------------------------
-- Stores MCP representation of constraint variables
-- @key   - Integer value provided by (getKey i) (i :: ID)
-- @value - MCP representation of constraint variable with ID i
type IntVarMap = Map.Map Integer ModelInt

-- Stores MCP representation of lists of constraint variables
-- @key   - Integer value provided by (getKey i) (i :: ID)
-- @value - MCP representation of list of constraint variables with ID i
type ColVarMap = Map.Map Integer ModelCol

-- Translation state for Haskell's state monad
-- @intVarMap     - Table of already translated constraint variables
-- @colVarMap     - Table of already translated lists of constraint variables
--                  (only used for translateGecode)
-- @nextIntVarRef - Next variable reference
-- @nextColVarRef - Next list variable reference (only used for translateGecode)
-- @additionalCs  - additional constraints for MCP collections
--                  (only used for translateGecode)
-- @labelInfo     - collected labeling information
data TLState = TLState {
  intVarMap     :: IntVarMap,
  colVarMap     :: ColVarMap,
  nextColVarRef :: Int,
  nextIntVarRef :: Int,
  additionalCs  :: [Model],
```

```
  labelInfo     :: MCPLabelInfo }

-- Initial state
baseTLState :: TLState
baseTLState = TLState {
  intVarMap     = Map.empty,
  colVarMap     = Map.empty,
  nextIntVarRef = 0,
  nextColVarRef = 0,
  additionalCs  = [],
  labelInfo     = baseLabelInfo }

-- Initial (empty) labeling information
baseLabelInfo :: MCPLabelInfo
baseLabelInfo = Info {
  labelVars    = Nothing,
  domainVars   = [],
  mcpLabelVars = Nothing,
  labelID      = Nothing,
  strategy     = Nothing }

-- The Overton and Gecode solvers work on different representations of lists of
-- constraint variables. Therefore each solver has its own translation function:

-- Translates list of finite domain constraints into a MCP model for the
-- Overton Solver and collects labeling information if available
-- using Haskell's state monad
translateOverton :: [FDConstraint] -> (MCPModel,MCPLabelInfo)
translateOverton fdCs =
  let (mcpCs,state) = runState (mapM (translateConstr translateOvertonList) fdCs)
                       baseTLState
      info          = labelInfo state
  in (ModelWrapper mcpCs, info)

-- Translates list of finite domain constraints into a MCP model for the
-- Gecode Solver and collects labeling information if available
-- using Haskell's state monad
translateGecode :: [FDConstraint] -> (MCPModel,MCPLabelInfo)
translateGecode fdCs =
  let (mcpCs,state) = runState (mapM (translateConstr translateGecodeList) fdCs)
                       baseTLState
      info          = labelInfo state
      mcpColCs      = additionalCs state
  in (ModelWrapper (mcpCs ++ mcpColCs), info)

-- Translates a single finite domain constraint into a specific MCP constraint
-- using Haskell's state monad.
-- This function works for both solvers by calling different functions to
-- translate lists
-- @tlList - function to translate lists of constraint variables
```

```
--           to MCP collections
translateConstr :: (FDList (FDTerm Int) -> State TLState ModelCol) -> FDConstraint
                -> State TLState Model
translateConstr _ (FDRel relop t1 t2)                  = do
  mcpTerm1 <- translateTerm t1
  mcpTerm2 <- translateTerm t2
  let mcpRelop = translateRelOp relop
  return $ mcpRelop mcpTerm1 mcpTerm2
translateConstr _ (FDArith arithOp t1 t2 r)            = do
  mcpTerm1  <- translateTerm t1
  mcpTerm2  <- translateTerm t2
  mcpResult <- translateTerm r
  let mcpArithOp = translateArithOp arithOp
  return $ (mcpArithOp mcpTerm1 mcpTerm2) @= mcpResult
translateConstr tlList (FDSum vs r)                    = do
  mcpVs     <- tlList vs
  mcpResult <- translateTerm r
  return $ (xsum mcpVs) @= mcpResult
translateConstr tlList (FDAllDifferent vs)             = do
  mcpVs <- tlList vs
  return $ allDiff mcpVs
translateConstr tlList (FDDomain vs@(FDList _ ts) l u) = do
  mcpVs <- tlList vs
  mcpL  <- translateTerm l
  mcpU  <- translateTerm u
  state <- get
  let info     = labelInfo state
      dVars    = domainVars info
      newInfo  = info { domainVars = dVars ++ ts }
      newState = state { labelInfo = newInfo }
  put newState
  return $ domain mcpVs mcpL mcpU
translateConstr tlList (FDLabeling str vs j)           = do
  mcpVs <- tlList vs
  state <- get
  let info     = labelInfo state
      newInfo  = info { labelVars    = Just vs
                      , mcpLabelVars = Just mcpVs
                      , labelID      = Just j
                      , strategy     = Just str }
      newState = state { labelInfo = newInfo }
  put newState
  return (toBoolExpr True)

-- Constraining a MCP collection of variables to a domain
-- defined by a lower and upper boundary
domain :: ModelCol -> ModelInt -> ModelInt -> Model
domain varList lower upper = forall varList (\var -> var @: (lower,upper))

-- Translates integer terms to appropriate MCP terms
```

```
-- using Haskell's state monad
translateTerm :: FDTerm Int -> State TLState ModelInt
translateTerm (Const x) = return (cte x)
translateTerm v@(FDVar i) = do
  state <- get
  let varMap = intVarMap state
  maybe (newVar v) return (Map.lookup (getKey i) varMap)

-- Creates a new MCP variable for the given constraint variable
-- Updates the translation state by inserting the MCP representation
-- of the variable into the map and incrementing the varref counter
newVar :: FDTerm Int -> State TLState ModelInt
newVar (FDVar i) = do
  state <- get
  let varMap   = intVarMap state
      varRef   = nextIntVarRef state
      nvar     = asExpr (ModelIntVar varRef :: ModelIntTerm ModelFunctions)
      newState = state { nextIntVarRef = varRef + 1
                       , intVarMap = Map.insert (getKey i) nvar varMap }
  put newState
  return nvar

-- Translates lists of fd terms to MCP collection for the Overton Solver
translateOvertonList :: FDList (FDTerm Int) -> State TLState ModelCol
translateOvertonList (FDList _ vs) = do mcpExprList <- mapM translateTerm vs
                                        return (list mcpExprList)

-- Translates lists of fd terms to MCP collection for the Gecode Solver
translateGecodeList :: FDList (FDTerm Int) -> State TLState ModelCol
translateGecodeList l@(FDList i vs) = do
  state <- get
  let varMap = colVarMap state
  maybe (newColVar l) return (Map.lookup (getKey i) varMap)

-- Creates a new MCP collection variable for the given list,
-- Updates the translation state by inserting its MCP representation
-- into the map and incrementing the corresponding varref counter,
-- Creates additional constraints for the collection variable
-- describing its size and elements (only used for translateGecode)
newColVar :: FDList (FDTerm Int) -> State TLState ModelCol
newColVar (FDList i vs) = do
  mcpVs <- mapM translateTerm vs
  state <- get
  let varMap   = colVarMap state
      varRef   = nextColVarRef state
      nvar     = asCol (ModelColVar varRef :: ModelColTerm ModelFunctions)
      colCs    = additionalCs state
      newState = state { nextColVarRef = varRef + 1
                       , colVarMap = Map.insert (getKey i) nvar varMap
                       , additionalCs = colCs ++ (newColCs nvar mcpVs) }
```

```
  put newState
  return nvar

-- Creates additional constraints for collection variables
-- describing the size of a collection and its elements
-- (only used for translateGecode)
newColCs :: ModelCol -> [ModelInt] -> [Model]
newColCs col vs = (size col @= cte (length vs)) : newColCs' col vs 0
  where
   newColCs' _   []     _ = []
   newColCs' col (v:vs) n = ((col @!! n) @= v) : newColCs' col vs (n+1)

-- Translates relational operators to appropriate MCP operators
translateRelOp Equal = (@=)
translateRelOp Diff  = (@/=)
translateRelOp Less  = (@<)

-- Translates arithmetic operators to appropriate MCP operators
translateArithOp Plus  = (@+)
translateArithOp Minus = (@-)
translateArithOp Mult  = (@*)

-- ---------------------------------------------------------------------------
-- Solving MCP model
-- ---------------------------------------------------------------------------
-- Types for MCP model trees parametrized with specific FD solver:
type OvertonTree = Tree (FDInstance OvertonFD) ModelCol
type GecodeTree  =
  Tree (FDInstance (GecodeWrappedSolver RuntimeGecodeSolver)) ModelCol

-- Calls solve function for specific solver
solveWithMCP :: MCPSolver -> MCPModel -> MCPLabelInfo -> MCPSolution
solveWithMCP Overton (ModelWrapper mcpCs) info = solveWithOverton mcpCs info
solveWithMCP Gecode  (ModelWrapper mcpCs) info = solveWithGecode  mcpCs info

-- Calls Overton Solver with corresponding model tree
-- and prepares solutions for KiCS2
solveWithOverton :: [Model] -> MCPLabelInfo -> MCPSolution
solveWithOverton mcpCs info = case maybeLabelVars of
  Nothing -> error "MCPSolver.solveWithOverton: Found no variables for labeling."
  Just lVars ->
    if (not (inDomain lVars dVars))
      then error "MCPSolver.solveWithOverton: At least for one Labeling variable
                  no domain was specified."
      else let mcpVars   = fromJust (mcpLabelVars info)
               choiceID  = fromJust (labelID info)
               strtgy    = fromJust (strategy info)
               modelTree = toModelTree mcpCs mcpVars
               solutions = snd $ solve dfs it $
                 (modelTree :: OvertonTree) >>= labelWith strtgy
```

```
          in SolWrapper (SolInfo (map (map toCurry) solutions) lVars choiceID)
  where maybeLabelVars = labelVars info
        dVars          = domainVars info

-- Calls Gecode Solver with corresponding model tree
-- and prepares solutions for KiCS2
solveWithGecode :: [Model] -> MCPLabelInfo -> MCPSolution
solveWithGecode mcpCs info = case maybeLabelVars of
  Nothing -> error "MCPSolver.solveWithGecode: Found no variables for labeling."
  Just lVars ->
    if (not (inDomain lVars dVars))
      then error "MCPSolver.solveWithGecode: At least for one Labeling variable
                  no domain was specified."
      else let mcpVars   = fromJust (mcpLabelVars info)
               choiceID  = fromJust (labelID info)
               strtgy    = fromJust (strategy info)
               modelTree = toModelTree mcpCs mcpVars
               solutions = snd $ solve dfs it $
                 (modelTree :: GecodeTree) >>= labelWith strtgy
           in SolWrapper (SolInfo (map (map toCurry) solutions) lVars choiceID)
  where maybeLabelVars = labelVars info
        dVars          = domainVars info

-- checks, if a domain was specified for every labeling variable
inDomain :: FDList (FDTerm Int) -> [FDTerm Int] -> Bool
inDomain (FDList _ lVars) dVars = null $ lVars \\ dVars

-- Label MCP collection with given strategy
labelWith :: (FDSolver s, MonadTree m, TreeSolver m ~ FDInstance s,
  EnumTerm s (FDIntTerm s)) => LabelingStrategy -> ModelCol
  -> m [TermBaseType s (FDIntTerm s)]
labelWith strategy col = label $ do
  lst <- getColItems col maxBound
  return $ do
    lsti <- colList col $ length lst
    labelling (matchStrategy strategy) lsti
    assignments lsti

-- select corresponding MCP labeling function for given labeling strategy
matchStrategy :: EnumTerm s t => LabelingStrategy -> [t] -> s [t]
matchStrategy FirstFail = firstFail
matchStrategy MiddleOut = middleOut
matchStrategy EndsOut   = endsOut
matchStrategy _         = inOrder

-- Transform a list of MCP constraints into a monadic MCP model tree
toModelTree :: FDSolver s => [Model] -> ModelCol -> Tree (FDInstance s) ModelCol
toModelTree model mcpLabelVars =
  mapM_ (\m -> addC (Left m)) model >> return mcpLabelVars
```

D. Beispiel für Sonderbehandlung der Curry-Bindungs-Constraints hinsichtlich des Wrappings

Das folgende Beispiel zeigt die in Kapitel 4.3.3 angesprochene Problematik hinsichtlich des Wrappings der Curry-Bindungs-Constraints noch einmal auf:

```
l =:= genVars 2 & domain l 1 2 & labeling l where l free
```

Listing 131: Beispiel für Problem bei Kombination von Curry-Bindungs-Constraints und FD-Constraints

Wie man sieht, wird in diesem Beispiel ein Curry-Gleichheits-Constraint über der freien Variable `l` kombiniert mit FD-Constraints über der gleichen freien Variable. Die externe Implementierung von `domain` und `labeling` sorgt dafür, dass die `Choice`, durch die `l` intern repräsentiert wird, in eine `NarrowedChoice` umgewandelt wird. Der Suchraum dieser `NarrowedChoice` besteht aus allen möglichen Listen, die `l` annehmen könnte, also die leere Liste, die Liste mit einer freien Integer-Variablen, mit zwei freien Integer-Variablen usw.
Löst man das durch `l =:= genVars 2` erzeugte Bindungs-Constraint für die Variable `l`. So legt die zugehörige Bindungsentscheidung fest, dass in den `NarrowedChoices` nur die Pfade weiter ausgewertet werden sollen, in denen `l` eine zweielementige Liste ist.
Die folgenden drei Abbildungen von KiCS2-Auswertungsbäumen für den obigen Beispielausdruck verdeutlichen die Problematik: Abbildung 8 zeigt den KiCS2-Auswertungsbaum, **bevor** durch Aufruf von `searchWrappedCs` alle "eingepackten" Constraints eingesammelt werden.
Abbildung 9 stellt den Auswertungsbaum nach dem Aufruf von `searchWrappedCs` dar, das heißt, alle `WrappedConstraints` in einem Pfad des Baumes wurden in einem `Guard`-Ausdruck gesammelt. In diesem Fall sind auch die Curry-Bindungs-Constraints in einem Constraint-Wrapper "eingepackt".
In Abbildung 10 wird der Auswertungsbaum ebenfalls nach dem Aufruf von `searchWrappedCs` dargestellt. Die Curry-Bindungs-Constraints werden dieses Mal allerdings gesondert behandelt und nicht in den Constraint-Wrapper "gepackt".
Alle drei Abbildungen sind stark vereinfacht: Das für den Ausdruck `l =:= genVars 2` erzeugte Bindungs-Constraint wird durch `CBC_l` (für **C**urry-**B**indungs-**C**onstraint für **l**) repräsentiert. Auch die FD-Constraints werden verkürzt dargestellt: So steht `DomainN` für ein passendes `FDDomain`-Constraint über einer `N`-stelligen Liste. Das heißt, `Domain0` ist äquivalent zu dem Constraint-Term `FDDomain [] (Const 1) (Const 2)`, `Domain1` zu dem `FDConstraint`-Konstruktorterm `FDDomain [FDVar 1] (Const 1)(Const 2)` usw. Für `LabelN` gilt das gleiche: Hier entsprechen die verkürzten Formen passenden `FDLabeling`-Constraints mit der entsprechenden Anzahl von Listenelementen. Abgesehen von dem Curry-Bindungs-Constraint sind alle anderen Constraints in einem Constraint-Wrapper "eingepackt". Nur in Abbildung 9 ist das Curry-Bindungs-Constraint auch in einem solchen Wrapper "verpackt".

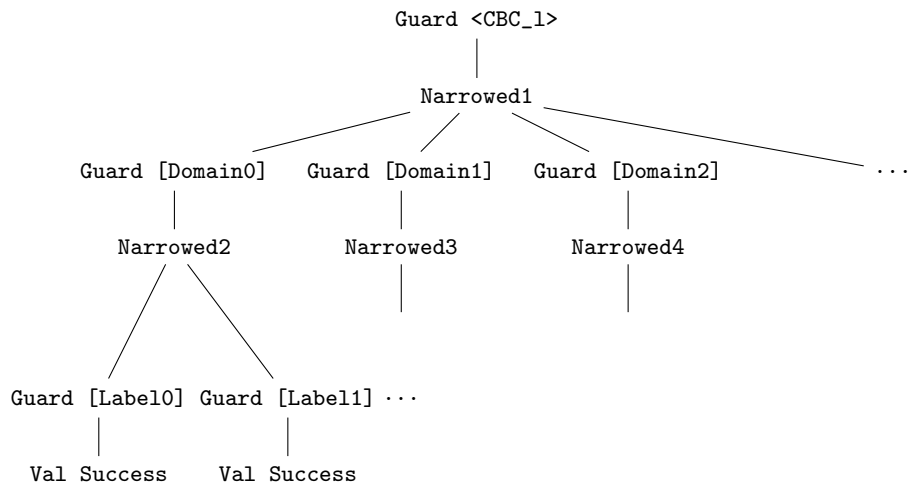


Abbildung 8: Ausschnitt des Suchbaums **vor** Aufruf von `searchWrappedCs`

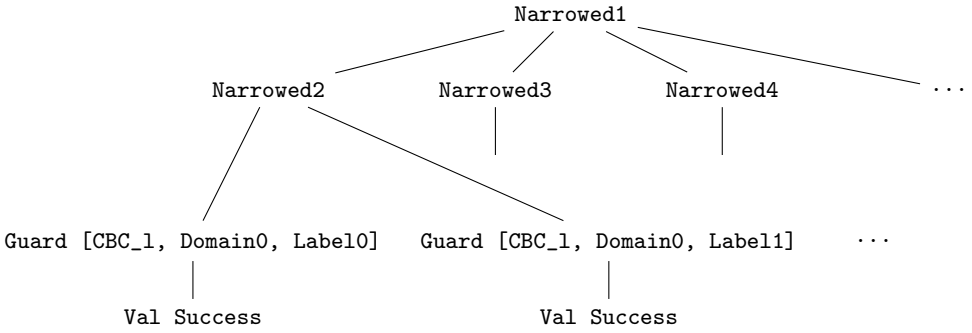


Abbildung 9: Ausschnitt des Suchbaums **nach** Aufruf von `searchWrappedCs` (Wrapping der Curry-Bindungs-Constraints)

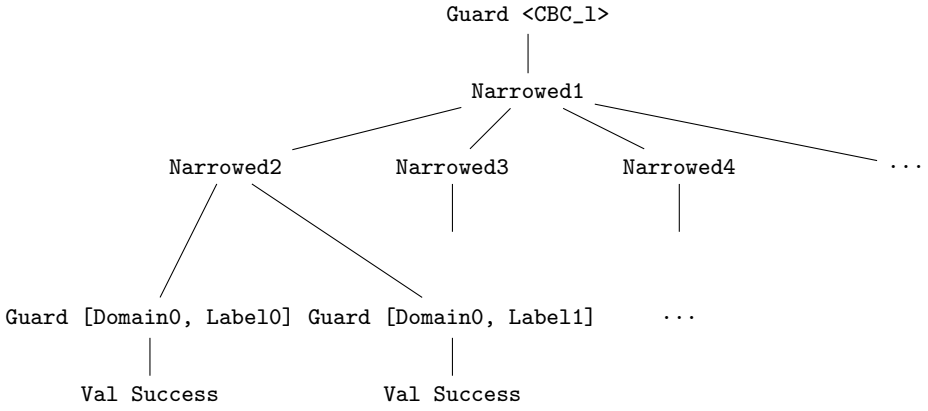


Abbildung 10: Ausschnitt des Suchbaums **nach** Aufruf von `searchWrappedCs` (separate Behandlung der Curry-Bindungs-Constraints)

Der Vergleich der Abbildungen 9 und 10 zeigt das Problem: Behandelt man alle Constraint-Typen gleich, so werden die Curry-Bindungs-Constraints ebenfalls eingesammelt. Dadurch werden sie

in einen `Guard`-Knoten “unterhalb“ der `NarrowedChoice`-Knoten verschoben. Selbst wenn man nun zuerst die Curry-Bindungs-Constraints löst, wird die Auswertung nicht mehr terminieren, da die `NarrowedChoice`-Knoten zu diesem Zeitpunkt bereits ausgewertet wurden. Es können also nicht mehr die “richtigen“ Pfade der `NarrowedChoice`s für die Auswertung ausgewählt werden. Stattdessen werden alle Pfade nacheinander ausgewertet.
Verhindern lässt sich dies nur, wenn man die Curry-Bindungs-Constraints gesondert behandelt und nicht mit den anderen Constraints einsammelt, denn in diesem Fall bleiben die `Guard`-Knoten mit Bindungs-Constraints “oberhalb“ der `NarrowedChoice`-Knoten im Auswertungsbaum (vergleiche Abbildung 10).

E. Generate & Test-Realisierung des N-Damen-Problems

```
-- Definition der Ungleichheit mit Hilfe von (=:=)
diff :: a -> a -> Success
diff x y = (x == y) =:= False

-- Definition eines "kleiner-gleich"-Constraints mit (=:=)
lessEqual :: a -> a -> Success
lessEqual x y = (x <= y) =:= True

-- Befindet sich die Dame auf dem Schachbrett?
posOnBoard :: Int -> Int -> Success
posOnBoard q n = lessEqual 1 q & lessEqual q n

-- Generiert alle möglichen Platzierungen für n Damen
-- auf einem nxn-Schachbrett
generate :: Int -> [Int] -> Success
generate n l =
  l =:= genVars n & foldl (\s q -> posOnBoard q n & s) success l

-- Testet, ob keine Dame eine andere schlagen kann
test :: [Int] -> Success
test [] = success
test (q:qs) = safe q qs 1 & test qs

-- Testet, ob die Dame q von keiner anderen Dame schlagbar ist
safe :: Int -> [Int] -> Int -> Success
safe _ [] _ = success
safe q (q1:qs) p = no_attack q q1 p & safe q qs (p+1)

-- Testet, ob q1 und q2 sich nicht gegenseitig schlagen können
no_attack q1 q2 p = diff q1 q2 & diff q1 (q2+p) & diff q1 (q2-p)

-- Erzeugt n freie Variablen
genVars n = if n == 0 then [] else var : genVars (n-1)
 where var free

-- Beispielaufruf: generate 4 l & test l where l free
```

Listing 132: Generate & Test Realisierung des N-Damen-Problems

Literatur

[1] Brassel, B. ; Hanus, M. ; Peemöller, B. ; Reck, F.: Implementing Equational Constraints in a Functional Language. In: *Proc. of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011)*, INFSYS Research Report 1843-11-06 (TU Wien), 2011, S. 22–33

[2] Brassel, B. ; Hanus, M. ; Peemöller, B. ; Reck, F.: KiCS2: A New Compiler from Curry to Haskell. In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, Springer LNCS 6816, 2011, S. 1–18

[3] Gecode: *Generic Constraint Developement Environment*. `http://www.gecode.org/`

[4] Gent, Ian P. ; Miguel, Ian ; Rendl, Andrea: Tailoring solver-independent constraint models: a case study with ESSENCE' and MINION. In: *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation*. Berlin, Heidelberg : Springer-Verlag, 2007 (SARA'07). – ISBN 978–3–540–73579–3, 184–199

[5] Gockel, Tilo: *Form der wissenschaftlichen Ausarbeitung*. Berlin; Heidelberg : Springer, 2008 `http://www.formbuch.de`. – ISBN 978–3–540–78613–9

[6] Hanus, Michael et a.: *Curry: An integrated functional logic language (version 0.8.2)*. `http://www-ps.informatik.uni-kiel.de/currywiki/`. Version: 2006

[7] Marlow, Simon: An extensible dynamically-typed hierarchy of exceptions. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. New York, NY, USA : ACM, 2006 (Haskell '06). – ISBN 1–59593–489–8, 96–106

[8] Overton, David: *Haskell FD Library*. `http://overtond.blogspot.de/2008/07/pre.html`

[9] Schrijvers, Tom ; Stuckey, Peter ; Wadler, Philip: Monadic constraint programming. In: *Journal of Functional Programming* 19 (2009), November, Nr. 6, 663–697. `http://dx.doi.org/10.1017/S0956796809990086`. – DOI 10.1017/S0956796809990086

[10] Wuille, Pieter ; Schrijvers, Tom: Monadic Constraint Programming with Gecode. In: Frisch, Alan M. (Hrsg.) ; Lee, Jimmy (Hrsg.): *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation,*, 2009, 171–185

[11] Wuille, Pieter ; Schrijvers, Tom: Expressive models for Monadic Constraint Programming. In: Mancini, Toni (Hrsg.) ; Pearson, Justin K. (Hrsg.): *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation,*, 2010, 15

[12] Wuille, Pieter ; Schrijvers, Tom: Parameterized models for on-line and off-line use. In: Marino, Julio (Hrsg.): *WFLP 2010 Post-conference Proceedings,*, Springer-verlag, 2011, 101–118