Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Masterarbeit

Implementierung von Constraint-Lösern für Curry mittels SMT

Sven Hüser

März 2016

betreut von
Prof. Dr. Michael Hanus
Dipl.-Inf. Jan Tikovsky

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Wattenbek, 28. März 2016

Sven Hüser

Abstract

Constraint-Programming befasst sich mit dem Modellieren und Lösen von Constraint Problemen. Die funktional-logische Programmiersprache Curry eignet sich aufgrund des deklarativen Programmierstils gut als Modellierungssprache für solche Constraint Probleme. Allerdings übersetzen die Curry-Implementierungen PAKCS und KiCS2 in unterschiedliche Sprachen: Prolog und Haskell. Letztere bietet keine direkten Möglichkeiten, Constraint Probleme zu lösen. Diese Arbeit beschreibt die Entwicklung einer Schnittstelle, mit der eigenständige Constraint Solver in Curry verwendet werden können, um Finite-Domain-Constraint-Probleme zu lösen. Dafür wird zunächst eine Bibliothek für die Modellierung von FD-Problemen entwickelt. Anschließend werden Werkzeuge geschaffen, um mit SMT-Solvern zu kommunizieren und diese zu nutzen, Lösungen für FD-Probleme zu finden.

Inhaltsverzeichnis

Listings

Teil I.

Einleitung

1. Motivation

Das Thema beim Constraint-Programming ist das Finden von Lösungen für Probleme, die mithilfe von Constraints angegeben werden. Diese Constraints stellen Beziehungen zwischen Variablen dar und legen Bedingungen fest, die diese Variablen für eine gültige Lösung eines Problems erfüllen müssen. Mit einer deklarativen Beschreibung von Regeln für Variablen über einen endlichen oder unendlichen Wertebereich wird das Problem spezifiziert. Im Fall von endlichen Wertebereichen für die Variablen wird auch von Finite-Domain-Constraint-Programming gesprochen.

Constraint-Programming setzt sich im Grunde aus zwei Teilen zusammen: Auf der einen Seite findet sich eine Komponente zum Formulieren und Modellieren von Problemen, auf der anderen Seite ein Constraint Solver, der Lösungen für das angegebene Problem sucht. Dabei nutzt der Solver verschiedene Techniken, um die Constraints zu vereinfachen und Lösungen für die Variablen zu finden. Sogenannte Labeling-Strategien werden genutzt, um Werte für die Variablen auszuprobieren. Mittels Constraint Propagation werden diese Werte weitergereicht, so dass der Wertebereich – auch Domain genannt – weiterer Variablen eingeschränkt werden kann. Werden die gegebenen Bedingungen verletzt, wechselt der Solver wieder in einen vorherigen, gültigen Zustand und wählt eine neue Belegung für das Labeling.

Die funktional-logische Programmiersprache Curry ist aufgrund des deklarativen Charakters gut geeignet, um Constraint-Programming darin einzubetten. In dieser Arbeit soll eine Schnittstelle für bereits bestehende Constraint Solver entwickelt werden, welche unabhängig von der Curry-Implementierung ist; sie soll sowohl mit PAKCS als auch mit KiCS2 verwendet werden können. Die Implementierung umfasst eine Bibliothek zum Modellieren von Finite-Domain-Problemen sowie eine Schnittstelle, über die SMT-Solver [13] genutzt werden können, um Lösungen für diese Probleme zu suchen.

Zunächst folgen einige Kapitel, die einen Überblick über die Grundlagen liefern, die für das Verständnis der vorliegenden Arbeit nötig sind. Dazu zählen die Programmiersprache Curry, Constraint-Programming mit existierenden Implementierungen und

die Verwendung von Theoremlösern, die schließlich über die entwickelte Schnittstelle angebunden werden sollen.

Die Beschreibung der Implementierung ist weiter unterteilt: Kapitel 5 stellt die Idee vor, wie externe Constraint Solver angebunden werden können. Anschließend wird in Kapitel 6 die Entwicklung einer Bibliothek für das Modellieren von Finite-Domain-Constraint-Problemen in Curry beschrieben, bevor es in Kapitel 7 bis 9 um die Anbindung von SMT- und SAT-Solvern und die allgemeine Erweiterbarkeit der Implementierung geht.

Abschließend wird die Implementierung in Kapitel 10 mithilfe einiger FD-Probleme evaluiert und mit einer bereits bestehenden Implementierung für PAKCS verglichen. Das letzte Kapitel liefert eine Zusammenfassung und gibt einen Ausblick über Punkte, die eventuell verbessert oder weiterentwickelt werden können.

Folgendes Beispiel gibt an dieser Stelle schon einen kleinen Einblick in das, was mit der Implementierung möglich sein wird. Hier werden für Variablen x, y und z mit einem Wertebereich von 1 bis 10 die Constraints $x + y = z$ und $z > 3$ modelliert:

```
main :: [Int]
main =
let vs@[x,y,z] = take 3 (domain 1 10)
      model = x +# y =# z /\ z ># fd 3
  in solveFD [] vs model
```

Auswerten von `main` liefert die Listen $[x, y, z]$, bei denen die Kombinationen der Werte durch die Constraints eingeschränkt sind, zum Beispiel $[2, 3, 5]$.

Teil II.

Grundlagen

2. Curry

In diesem Abschnitt werden die grundlegenden Aspekte der funktional-logischen Programmiersprache Curry vorgestellt. Sie ist Mitglied in der Familie der deklarativen Programmiersprachen, die nützliche Vorteile gegenüber imperativen Sprachen mit sich bringen; dazu gehört oft, dass der Code deutlich kompakter und besser verständlich ist, was ihn auch weniger anfällig für Fehler macht. Die Implementierung der im Laufe dieser Arbeit entwickelten Schnittstelle wurde in Curry vorgenommen.

Wie die Klassifizierung als funktional-logische Sprache deutlich macht, finden sich in Curry sowohl Konzepte aus der funktionalen Programmierung wie Lazy Evaluation, verschachtelte Ausdrücke oder Funktionen höherer Ordnung als auch aus der Logikprogrammierung bekannte Charakteristika wie freie Variablen, partielle Datenstrukturen und Nichtdeterminismus inklusive der Constraintprogrammierung.

Curry ist stark von Haskell beeinflusst und weist von der Syntax her eine große Ähnlichkeit zu der funktionalen Programmiersprache auf. Wie Haskell verwendet auch Curry ein Typinferenzsystem nach Hindley-Milner [9], wobei lediglich das Überladen mittels Typklassen nicht enthalten ist, aber durchaus für zukünftige Versionen in Betracht gezogen werden kann. Im Rahmen dieser Arbeit werden zwei Implementierungen von Curry benutzt: das Kiel Curry System Version 2 (KiCS2)[1] sowie das Portland Aachen Kiel Curry System (PAKCS)[2].

Der aktuelle Curry Report [10] gibt eine ausführliche Beschreibung von Syntax und Semantik der Programmiersprache. Viele für Einsteiger geeignete Beispiele, wie mit Curry programmiert werden kann, lassen sich im Curry Tutorial [1] nachlesen.

2.1. Datentypen

In Curry können neue Datentypen mit dem Schlüsselwort `data` angegeben werden:

```
data T = C1 τ11 ... τ1n1 | ... | Cm τm1 ... τ1nm
```

[1] Version 0.5.0, verfügbar auf http://www-ps.informatik.uni-kiel.de/kics2/
[2] Version 1.14.0, verfügbar auf https://www.informatik.uni-kiel.de/~pakcs/

Eine solche Definition führt den neuen Datentyp `T` und m neue Datenkonstruktoren `C`$_1, \ldots,$`C`$_m$ ein. Die `C`$_i$ haben jeweils den Typ

$$\tau_{i1} \texttt{ -> } \cdots \texttt{ -> } \tau_{in_i} \texttt{ -> T}$$

für $1 \leq i \leq m$. Auf folgende Weise kann also ein Binärbaum definiert werden, dessen Knoten mit ganzen Zahlen beschriftet sind:

```
data IntTree = Leaf | Node IntTree Int IntTree
```

Damit nicht für jeden neuen Typen, mit dem die inneren Knoten beschriftet sein können, ein neuer Typ angegeben werden muss, ist es hier sinnvoller, einen polymorphen Typen zu definieren:

$$\texttt{data T } \alpha_1 \ldots \alpha_n \texttt{ = C}_1 \ \tau_{11} \ldots \tau_{1n_1} \texttt{ | } \ldots \texttt{ | C}_m \ \tau_{m1} \ldots \tau_{1n_m}$$

Dies führt einen neuen n-stelligen Typkonstruktor `T` ein. Für die Datenkonstruktoren `C`$_i$ ergibt sich folglich der Typ

$$\tau_{i1} \texttt{ -> } \cdots \texttt{ -> } \tau_{i1} \texttt{ -> T } \alpha_1 \ldots \alpha_n$$

wobei die Typausdrücke τ_{ij} aus den Typvariablen $\alpha_1, \ldots, \alpha_n$ sowie selbst definierten oder in Curry eingebauten Typkonstruktoren aufgebaut sind. Ein Binärbaum, parametrisiert über `a`, sieht dann folgendermaßen aus:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Um komplexere Typdefinitionen übersichtlicher zu gestalten, lassen sich mit dem Schlüsselwort `type` Typsynonyme angeben. Der eingangs eingeführte Typ `IntTree` kann auf diese Weise mit dem gerade definierten allgemeinen Typen `Tree a` umgesetzt werden:

```
type IntTree = Tree Int
```

Ob ein Typsynonym oder direkt dessen Definition benutzt wird, ändert nichts an der Typisierung des Programms.

2.2. Funktionen

Funktionen werden in Curry durch eine optionale Signatur (auslassen führt zu Typinferenz) und eine Reihe von Regeln angegeben. Eine Funktion `f` mit Signatur und einer Regel kann damit wie folgt definiert werden:

```
f :: τ1 -> ... -> τn -> τ
f p1 ... pn = e
```

In der Signatur sind $\tau_1, \ldots, \tau_n, \tau$ Typausdrücke, welche bereits im vorigen Abschnitt erwähnt wurden. Für die n-stellige Funktion `f` mit $n \geq 0$ werden Muster `p1 ... pn` auf der linken Regelseite und ein Ausdruck `e` auf der rechten Regelseite angegeben. Über die Muster kann man wie in Haskell bei mehreren Regeln für eine Funktion festlegen, welche Regel angewandt werden soll.

Als konkretes Beispiel einer Funktion betrachten wir wieder den oben definierten `Tree`. Die Funktion `mirrorTree` soll einen Baum als Parameter nehmen und an einer Achse durch den Wurzelknoten spiegeln. Um sich über die Typen einer Funktion im klaren zu sein, ist es ratsam zu Beginn die Signatur aufzuschreiben:

```
mirrorTree :: Tree a -> Tree a
```

Um einen Baum vom Typ `Tree a` zu spiegeln, müssen lediglich die beiden Unterbäume vertauscht und `mirrorTree` rekursiv auf diese angewandt werden. Ist der Baum ein Blatt, geben wir ein Blatt zurück. Es ergibt sich somit Folgendes:

```
mirrorTree Leaf         = Leaf
mirrorTree (Node l a r) = Node (mirrorTree r) a (mirrorTree l)
```

Wie hier zu sehen ist, können in einem Muster sowohl Konstruktorterme als auch Variablen stehen. Erstere matchen auf den jeweils angegebenen Fall, letzte passen bezüglich des Pattern Matchings immer. Variablen werden dann mit dem konkreten Wert belegt und können auf der rechten Seite benutzt werden. Zusätzlich gibt es die spezielle Variable `_` (Unterstrich), die ebenfalls immer passt, für die aber keine Bindung vorgenommen wird.

2.3. Nichtdeterminismus

Anders als in Haskell, wo bei überlappenden Mustern die textuell erste Regel gewählt wird, werden diese Fälle in Curry nichtdeterministisch behandelt und alle passenden Regeln angewandt. Ein einfaches Beispiel dafür ist die 0-stellige Funktion `coin`:

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

Die Auswertung von `coin` liefert sowohl 0 als auch 1 als Ergebnis. Curry benutzt „call time choice“, das heißt, dass Vorkommen der selben Variablen geteilt werden. So wird mehrfaches Auswerten identischer Ausdrücke verhindert. Mit einer Funktion wie

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

ist das Ergebnis von `double coin` entweder `0` oder `2`, weil der Wert von `coin` auf `0` oder `1` festgelegt wird, bevor die Regel angewandt wird. Mit „run time choice“, ohne das Teilen der Werte, wären die möglichen Ergebnisse des Aufrufs von `double coin` entweder `0`, `1` oder `2`.

Bei Regeln können sogar die gleichen linken Regelseiten angegeben werden, wie das nächste Beispiel zeigen soll. In einen Baum vom Typ `Tree a` wird mit der Funktion `replaceLeaf` ein Blatt an einer beliebigen Stelle durch einen anderen Baum ersetzt:

```
replaceLeaf :: Tree a -> Tree a -> Tree a
replaceLeaf x Leaf         = x
replaceLeaf x (Node l a r) = Node (replaceLeaf x l) a r
replaceLeaf x (Node l a r) = Node l a (replaceLeaf x r)

simpleIntTree :: IntTree
simpleIntTree = Node (Node Leaf 2 Leaf) 1 (Node Leaf 3 Leaf)

oneNode :: IntTree
oneNode = Node Leaf 4 Leaf
```

Der Aufruf `replaceLeaf oneNode simpleIntTree` liefert demnach die folgenden vier Ergebnisse, da in `simpleIntTree` jedes Vorkommen von `Leaf` einmal durch `oneNode` ersetzt wird:

```
Node (Node (Node Leaf 4 Leaf) 2 Leaf) 1 (Node Leaf 3 Leaf)
Node (Node Leaf 2 (Node Leaf 4 Leaf)) 1 (Node Leaf 3 Leaf)
Node (Node Leaf 2 Leaf) 1 (Node (Node Leaf 4 Leaf) 3 Leaf)
Node (Node Leaf 2 Leaf) 1 (Node Leaf 3 (Node Leaf 4 Leaf))
```

Manchmal kann es allerdings sinnvoll sein, überlappende Regeln zu nutzen, ohne dass dadurch Nichtdeterminismus eingeführt wird. In diesen Fällen hilft der ebenfalls

aus Haskell bekannte Case-Ausdruck, da dieser deterministisch ist und hier die Muster von oben nach unten probiert werden. Dadurch ist ein solches Beispiel möglich:

```
doSomething :: Tree a -> ...
doSomething tree = case tree of
                     Leaf             -> ...
                     Node Leaf a Leaf -> ...
                     Node l a Leaf    -> ...
                     Node Leaf a r    -> ...
```

Dies kann notwendig sein, wenn I/O Operationen ausgeführt werden, da in diesen kein Nichtdeterminismus erlaubt ist.

Neben überlappenden Mustern existiert auch der `(?)`-Operator, der die nichtdeterministische Wahl ausdrückt. Es wird nichtdeterministisch `x` oder `y` gewählt:

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

Der Operator selbst ist ebenfalls durch überlappende linke Regelseiten implementiert. Auf diese Weise könnte die Definition von oben auch `coin = 0 ? 1` lauten.

2.4. Freie Variablen

Wie zu Beginn erwähnt, deckt Curry auch Paradigmen aus der logischen Programmierung ab. Daher dürfen Ausdrücke freie Variablen enthalten, für die versucht wird, Werte so zu berechnen, dass der jeweilige Ausdruck zu einem Datenterm reduziert werden kann. Freie Variablen müssen explizit in der Form `x free` angegeben werden. Als Beispiel betrachten wir folgende Verwandtschaftsbeziehungen:

```
data Person = Hans | Herbert | Anna | Lisa

mother Hans    = Anna
mother Lisa    = Anna
mother Herbert = Lisa
```

Alle Kinder von `Anna` können wir dann berechnen, indem wir die Gleichung

```
mother x == Anna
  where x free
```

lösen. Führen wir dies in PAKCS oder KiCS2 aus, wird `x` entsprechend der obigen Definition von `mother` an die Werte `Hans`, `Lisa` und `Herbert` gebunden. Die Ausgabe führt die Bindungen von Variablen in geschweiften Klammern vor dem Ergebnisausdruck auf. In den ersten beiden Fällen ist dies `True`, im dritten `False`:

```
{x=Hans} True
{x=Lisa} True
{x=Herbert} False
```

In der Prelude von Curry ist die Funktion `solve True = True` definiert, die hier genutzt werden kann, um bei solchen Gleichungen nur die Fälle zu erhalten, die zu `True` auswerten:

```
solve $ mother x == Anna
  where x free
```

Dadurch kann `x` nicht mit `Herbert` belegt werden. Wenn kein Wert gefunden werden kann – ebenso wenn keine Regel angewandt werden kann – teilen PAKCS und KiCS2 dies mit. Versuchen wir hier, die Mutter von Anna mit `mother Anna == x where x free` zu berechnen, meldet PAKCS

```
*** No value found!
```

KiCS2 gibt stattdessen `!` aus. Freie Variablen dürfen auch nach einem `let` angegeben werden. Folgende Funktion prüft, ob eine Person eine Großmutter ist:

```
isGrandmother g | let x free in mother (mother x) == g = True
                | otherwise = False
```

Der Aufruf `isGrandmother Anna` wertet dann zu `True` aus.

3. Constraint-Programming

Das Ziel bei der Programmierung mit Constraints ist das Lösen von Problemen. Beschrieben werden diese Probleme durch Bedingungen (Constraints), die für eine valide Lösung gelten müssen. Da nicht die Schritte für das Finden einer Lösung beschrieben werden, wie dies bei imperativen Sprachen der Fall wäre, gehört das Constraint-Programming in den Bereich der deklarativen Programmierung. Hat das zu modellierende Problem einen endlichen Wertebereich, spricht man von Finite-Domain-Constraint-Programming.

Ein anschauliches Finite-Domain-Problem ist das sogenannte N-Damen-Problem. Auf einem $N \times N$ Felder großen Schachbrett müssen N Damen so platziert werden, dass keine die jeweils anderen schlagen kann. Eine Dame kann dabei ziehen wie im Schach, sie darf also mit keiner anderen Dame in einer Reihe, Spalte oder Diagonalen des Schachbrettes stehen. Die Lösung für eine Dame ist demnach trivial, sie kann das eine mögliche Feld einnehmen und von keiner anderen Dame geschlagen werden. Für zwei und drei Damen gibt es offensichtlich keine Lösungen. Eine der beiden Lösungen für das 4-Damen-Problem ist in Abbildung 3.1 gegeben.

Die folgenden beiden Abschnitte stellen jeweils eine Modellierung des N-Damen-Problems vor: zunächst mit den bereits in PAKCS vorhandenen Möglichkeiten für Constraint-Programming, anschließend mit der Programmiersprache Picat.

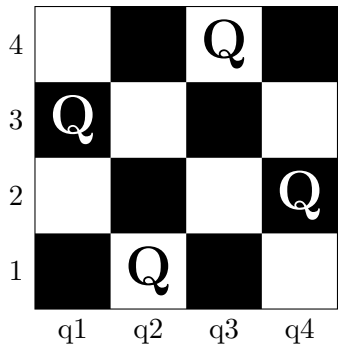

Abbildung 3.1.: Eine Lösung des 4-Damen-Problems

3.1. CLPFD mit PAKCS

Enthalten in der PAKCS-Implementierung von Curry ist unter anderem die Bibliothek `CLP.FP` zum Modellieren von FD-Problemen. Diese nutzt zum Lösen von Constraints das zugrunde liegende Prolog-System. Das N-Damen-Problem lässt sich damit wie in Listing 3.1 gezeigt implementieren.

Die Funktion `queens` erhält als Argumente eine Liste `options` von Optionen für den Solver und die Anzahl der Damen `n`. Gelöst wird das Problem durch einen Aufruf von `solveFD`, welches die Optionen, eine Liste von FD-Variablen und ein Constraint als Parameter nimmt. Frische FD-Variablen über einen bestimmten Wertebereich können über das Constraint `domain` erzeugt werden. Diese Funktion liefert eine unendliche Liste von FD-Variablen, eingeschränkt auf den angegebenen Wertebereich, aus der die benötigte Anzahl herausgenommen werden kann. Die Modellierung des eigentlichen Problems findet über die selbst definierten Constraints `allSafe`, `safe` und `noAttack` statt. Letzteres sorgt dafür, dass zwei Damen sich nicht gegenseitig schlagen können, indem verhindert wird, dass sie in der gleichen Reihe (`q1 /=# q2`) und auf der gleichen Diagonalen (`q1 /=# q2 +# p /\ q1 /=# q2 -# p`) stehen. Mittels `safe` wird für eine Dame sichergestellt, dass keine andere diese bedroht, und `allSafe` wendet `safe` auf alle Damen an. Über die Optionen kann eingestellt werden, welche Strategie für das Labeling der Variablen eingesetzt werden soll. Beispielsweise sorgt `FirstFail` dafür,

```
queens :: [Option] -> Int -> [Int]
queens options n =
  let qs = take n (domain 1 n)
   in solveFD options qs (allSafe qs)

allSafe :: [FDExpr] -> FDConstr
allSafe []     = true
allSafe (q:qs) = safe q qs (fd 1) /\ allSafe qs

safe :: FDExpr -> [FDExpr] -> FDExpr -> FDConstr
safe _ []       _ = true
safe q (q1:qs) p = noAttack q q1 p /\ safe q qs (p +# fd 1)

noAttack :: FDExpr -> FDExpr -> FDExpr -> FDConstr
noAttack q1 q2 p = q1 /=# q2 /\ q1 /=# q2 +# p /\ q1 /=# q2 -# p
```

Listing 3.1: Implementierung des N-Damen-Problems in Curry für PAKCS

dass zuerst die Variablen mit dem eingeschränktesten Wertebereich mit Werten aus diesem belegt werden.

3.2. Picat

Picat[3] ist ein neues Mitglied in der Familie der logischen Programmiersprachen. Eine erste Betaversion wurde von Neng-Fa Zhou im Mai 2013 veröffentlicht. Auch funktionale Aspekte wie Funktionen und imperative Konstrukte wie Schleifen und Zuweisungen sind in die Sprache eingeflossen.

Die hohe Leistungsfähigkeit des in Prolog vorhandenen Nichtdeterminismus wird oft nicht ausgenutzt und ist für deterministische Berechnungen nicht nötig. Aus diesem Grund müssen Regeln in Picat explizit als nichtdeterministisch angegeben werden, wenn dies gewünscht ist. Regeln und Funktionen werden in Picat nicht mittels Unifikation, sondern mit Pattern Matching ausgewählt.

Um Contraint-Satisfaction-Probleme zu lösen, stehen die drei Solver-Module CP, SAT und MIP zur Verfügung, die über eine gemeinsame Schnittstelle angesprochen und somit einfach ausgewechselt werden können. Eine Technik, Berechnungen durch Speichern und Wiederverwenden von Zwischenergebnissen zu beschleunigen, ist Tabling. Das in Picat eingesetzte System ist von B-Prolog abgeleitet, einer weiteren Sprache von Neng-Fa Zhou. Planungsprobleme, wie sie zum Beispiel bei künstlichen Intelligenzen wie Robotern auftreten, können mit dem in Picat enthaltenen Planner-Modul auf einfache Weise modelliert werden.

Im Folgenden werden die für diese Arbeit interessanten Aspekte der Sprache beschrieben und anhand des N-Damen-Problem gezeigt, wie Picat genutzt werden kann, Finite-Domain-Probleme zu modellieren. Eine ausführliche Beschreibung ist im User's Guide to Picat [15] verfügbar.

Listing 3.2 zeigt eine mögliche Implementierung des N-Damen-Problems in Picat. Die Syntax erinnert an Prolog, Regeln haben im Allgemeinen die Form *Head* `=>` *Body*. In diesem Fall ist *Head* das einstellige Prädikat `nqueens`, das als einziges Argument die Anzahl N der Damen nimmt. Im *Body* wird die Variable `Q` gleich einem N-stelligen Array gesetzt und die Werte der Felder auf den Bereich von 1 bis N eingeschränkt. Durch die `foreach`-Schleife – diese wird in ein endrekursives Prädikat übersetzt – werden die Constraints für die Sicherheit der Damen erzeugt: zwei Damen dürfen nicht in der gleichen Zeile beziehungsweise Diagonalen stehen. Abschließend wird `solve` analog zu `labeling` in Prolog benutzt, indem eine optionale Liste von Optionen und

[3] http://picat-lang.org

```
nqueens(N, Q) =>
    Q = new_array(N),
    Q :: 1..N,
    foreach (I in 1..N-1, J in I+1..N)
        Q[I] #!= Q[J],                  % nicht gleiche Zeile
        abs(Q[I] - Q[J]) #!= J-I        % nicht gleiche Diagonale
    end,
    solve([ff],Q).
```

Listing 3.2: N-Damen-Problem in Picat

eine Liste von Variablen übergeben wird. Im gezeigten Code fehlt allerdings noch ein `import`-Statement, das darüber entscheidet, welcher Solver benutzt wird. Zur Auswahl stehen die drei Module `cp`, `mip` und `sat`, die entsprechend einen CP-, MIP- oder SAT-Solver über die Funktion `solve` verfügbar machen.

Mit dem CP-Solver kann das 4-Damen-Problem durch den folgenden Aufruf gelöst werden, der wie zu erwarten auch das in Abbildung 3.1 gezeigte Ergebnis liefert:

```
Picat> nqueens(4,Q).
Q = {2,4,1,3} ?;
Q = {3,1,4,2} ?;

no
```

Nachdem die einzigen beiden Lösungen gefunden wurden, führt die Suche nach einer weiteren zu `no`, wodurch angezeigt wird, dass es keine Lösung mehr gibt.

4. Theoremlöser

Dieses Kapitel befasst sich mit der Erfüllbarkeit von aussagenlogischen Formeln. Außerdem wird auf eine Erweiterung eingegangen, mit der sich komplexere Probleme beschreiben lassen, die beispielsweise auch ganzzahlige Variablen statt reiner Boolescher Variablen zulassen.

Zudem werden Programme – auch Solver genannt – eingeführt, die solche Erfüllbarkeitsprobleme lösen.

4.1. Satisfiability

Satisfiability (SAT) ist ein Entscheidungsproblem, bei dem gefragt wird, ob eine aussagenlogische Formel erfüllbar ist. SAT ist ein NP-vollständiges Problem, ein Algorithmus zum Lösen in Polynomialzeit konnte bisher nicht gefunden werden. Trotzdem lassen sich viele reale Beispiele für Erfüllbarkeitsprobleme effizient mit sogenannten SAT-Solvern lösen.

Eine aussagenlogische Formel φ besteht aus Aussagen. Dies können negierte ($\neg$) und durch Konjunktion ($\wedge$) und Disjunktion ($\vee$) verknüpfte Aussagen oder positive Literale (x) beziehungsweise negative Literale ($\overline{x}$) sein. Erfüllbar ist eine Formel, wenn es für alle vorkommenden Literale eine Belegung mit entweder wahr (1) oder falsch (0) gibt, so dass die gesamte Formel wahr wird. Die Formel

$$\varphi = \big(x_1 \vee \neg(\overline{x}_2 \wedge x_4)\big) \wedge \big(x_3 \wedge (x_1 \vee x_2)\big)$$

ist unter anderem mit der Belegung $\{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 1, x_4 \mapsto 0\}$ erfüllbar.

Häufig wird in der Komplexitätstheorie mit SAT der Spezialfall bezeichnet, bei dem Probleme in konjunktiver Normalform (englisch *conjunctive normal form*, CNF) vorliegen. Eine Formel in CNF ist eine Konjunktion von Klauseln, die selbst wiederum Disjunktionen einer Reihe von Variablen oder deren Negationen sind, und hat demnach die Form

$$\bigwedge_i \bigvee_j \psi_{ij} \,.$$

Die Formel φ von oben ist – ohne hier auf die nötige Umformung einzugehen – in CNF gegeben durch

$$\varphi_{\text{CNF}} = (x_1 \vee x_2 \vee \overline{x}_4) \wedge (x_3) \wedge (x_1 \vee x_2)\,.$$

4.1.1. Lingeling

Lingeling[4] ist einer der angesprochenen SAT-Solver. In Picat kommt er als Solver zum Einsatz, wenn das Modul `sat` importiert wird [14].

Lingeling hat bereits an mehreren Wettbewerben [5, 6] teilgenommen und ist oft unter den besten fünf Solvern zu finden. Auch zwei parallele Varianten des Solvers, die Lingeling jeweils als Backend nutzen, sind erfolgreich: Plingeling und Treengeling [4], auf die im Rahmen dieser Arbeit allerdings nicht weiter eingegangen wird.

4.1.2. DIMACS CNF Format

Um ein standardisiertes Dateiformat für Formeln zu haben, wurde 1993 das DIMACS CNF Format vorgeschlagen [8], das von allen gängigen SAT-Solvern, darunter auch Lingeling, verstanden wird. Die Beschreibung eines Erfüllbarkeitsproblems durch dieses Format ist in zwei Abschnitte geteilt: die Präambel und die Klauseln.

In der Präambel werden Informationen über die Probleminstanz angegeben. Dazu zählen auf der einen Seite Kommentarzeilen, die durch den Buchstaben `c` eingeleitet werden und Erläuterungen in menschenlesbarer Form angeben können. Kommentare werden von den SAT-Solvern ignoriert. Auf der anderen Seite steht die Problemzeile in der Präambel. Jede Datei muss genau eine solche Zeile enthalten, die mit dem Buchstaben `p` anfängt und die das Format der Formel, die Anzahl n der Variablen darin sowie die Anzahl m der Klauseln angibt. Nach den einleitenden Zeichen `c` und `p` muss jeweils ein Leerzeichen folgen. Für das Format sollte `cnf` angegeben werden.

Die Klauseln folgen direkt nach der Problemzeile. Für die Variablen wird angenommen, dass diese von 1 bis n durchnummeriert sind. Die Variable i wird einfach durch i angegeben, die negierte Version durch $-i$. Jede Klausel ist durch eine 0 abgeschlossen, wodurch erlaubt wird, dass mehrere Klauseln in einer Zeile stehen oder eine Klausel über mehrere Zeilen hinweg aufgeschrieben werden kann. Die Formel φ_{CNF} enthält die vier Variablen x_1 bis x_4 und hat drei Klauseln. Eine Datei mit dem in Listing 4.1 aufgeführten Inhalt ist demnach eine mögliche Repräsentation der Formel im DIMACS Format.

[4] http://fmv.jku.at/lingeling/

```
c Beispiel CNF Formel
p cnf 4 3
1 2 -4 0
3 0
1 2 0
```

Listing 4.1: Formel φ_{CNF} im DIMACS CNF Format

Lingeling kann nun mit einer solchen Datei aufgerufen werden, um eine erfüllende Belegung für die Variablen zu suchen. Alternativ kann der Solver auch von der Standardeingabe lesen, das Ende der Eingabe wird dann durch Signalisieren des End Of File (in einer Unix-Shell durch Drücken von Ctrl-D am Zeilenanfang) angegeben. Obiger Dateiinhalt (eingegeben in Lingeling) führt zu dieser Ausgabe auf die Standardausgabe:

```
s SATISFIABLE
v 1 2 3 4 0
```

Wie bei Kommentaren und der Präambel gibt ein Zeichen zu Beginn der Zeile die Bedeutung an: `s` steht für Solution und `v` für Variable. Die erste Zeile sagt aus, dass die Formel erfüllbar ist, die zweite Zeile gibt die erfüllende Belegung für die vier Variablen an. Wie bei der Eingabe steht i für die Belegung mit 1 und $-i$ für die Belegung mit 0.

Eine unerfüllbare Formel erzeugt eine entsprechende Ausgabe, wobei selbstverständlich keine Variablenbelegung angegeben werden kann:

```
p cnf 2 1
1 0 -1 0

s UNSATISFIABLE
```

4.2. Satisfiability Modulo Theories

Wird SAT mit weiteren Entscheidungstheorien verknüpft, ergibt sich daraus Satisfiability Modulo Theories (SMT) [2]. Mit SMT wird die Erfüllbarkeit von Formeln untersucht, denen eine Theorie wie beispielsweise die der Integer, Arrays oder Bitvektoren zugrunde liegt. Dies erlaubt, Formeln in einer ausdrucksstärkeren Form als durch reine Aussagen- oder Prädikatenlogik darzustellen und auf Erfüllbarkeit zu prüfen. Betrachten wir zum Beispiel die Formel

$$x < 10 \wedge y = 3 * x\,,$$

sind wir daran interessiert, ob diese erfüllbar ist, wenn $<, *, 3$ und 10 die aus der Integer-Arithmetik bekannten Interpretationen haben, und nicht, ob irgendeine Interpretation existiert, so dass die Formel erfüllbar ist. Analog zu den SAT-Solvern werden Programme, die SMT-Probleme lösen, als SMT-Solver bezeichnet. Ein solches Problem kann dann durch zur jeweiligen Theorie passende Entscheidungsverfahren oder durch Kodieren in ein reines SAT-Problem gelöst werden.

4.2.1. Z3

Momentan ist Z3 [13] der State of the Art, der von Microsoft Research entwickelt wird und dessen Quellcode auf GitHub[5] veröffentlicht ist. Z3 wird bei Microsoft in verschiedenen Bereichen der Softwareentwicklung benutzt, um die Korrektheit von Programmen zu verifizieren, Fehler in diesen zu finden oder um erweiterte Testfälle erzeugen zu können, die eine größtmögliche Codeüberdeckung bieten [13].

Die Verwendung als SAT-Solver ist ebenfalls möglich, allerdings hält sich die Ausgabe nicht an den oben beschriebenen DIMACS-Standard. Für die gleiche Eingabe wie in Abschnitt 4.1.2 liefert Z3 diese Ausgabe:

```
sat
1 -2 3 -4
```

Im Falle einer unerfüllbaren Formel ist die Ausgabe hier lediglich `unsat`. Sollen die Ausgaben von Lingeling und Z3 verarbeitet werden, müssen beide Formate verstanden werden.

4.2.2. SMT-LIB

Als Unterstützung für die Forschung bezüglich SMT wurde und wird noch immer die Satisfiability Modulo Theories Library (SMT-LIB) entwickelt. Ziel von SMT-LIB ist es, Beschreibungen der Theorien bereitzustellen und gemeinsame Eingabe- und Ausgabesprachen für SMT-Solver zu entwickeln. Im Folgenden werden nun ausgewählte Teile dieser Sprachen vorgestellt, die für die Modellierung von Finite-Domain-Problemen nötig sind. Nachfolgend werden die Benutzung von Z3 – stellvertretend für SMT-Solver, die den SMT-LIB-Standard umsetzen – sowie Teile der SMT-LIB-Sprachen in Version 2.5 vorgestellt, vollständige Sprachbeschreibungen inklusive abstrakter und konkreter Syntax finden sich in [3].

[5] https://github.com/Z3Prover/z3

Deklarationen und Formeln, die der Nutzer eingibt, werden intern auf einem Stack abgelegt. Variablen, die in den Formeln auftreten, müssen vor Verwendung deklariert werden. Angegeben werden müssen der Name und der Typ (im Folgenden Sorte genannt). Im nachstehenden Beispiel sind zwei Möglichkeiten aufgeführt, eine Integervariable zu deklarieren. Außerdem wird eine Funktion `fun` deklariert, die einen Integer als Parameter bekommt und einen Boolean-Wert zurückgibt:

```
(declare-const a Int)
(declare-fun b () Int)
(declare-fun fun (Int) Bool)
```

Dabei ist `declare-const` lediglich syntaktischer Zucker für die Definition einer 0-stelligen Funktion, also einer Variablen. Dass durch `const` impliziert wird, dass es sich um eine Konstante handelt, ist zwar richtig, da im Sinne der Lösung der Wert konstant ist. Im Folgenden wird allerdings der Begriff Variable benutzt, da es sich bei den Werten im Sinne der Modellierung um Variablen handelt.

Um dem Stack eine Formel hinzuzufügen, wird der `assert` Befehl benutzt. Mit dem Befehl `check-sat` wird dann geprüft, ob die Formeln, die sich auf dem Stack befinden, erfüllbar sind. Erfüllbar heißt, dass es für alle vom Benutzer angegebenen Variablen und Funktionen eine Interpretation gibt, so dass alle Formeln wahr sind:

```
(declare-const a Int)
(declare-fun fun (Int) Bool)
(assert (> a 1336))
(assert (fun a))
(check-sat)
(get-model)
```

Die erste Assertion gibt an, dass `a` größer als 1336 sein muss. Die Zweite sagt aus, dass `fun` angewendet auf `a` wahr sein muss. Für die Rückgabe von `check-sat` gibt es drei Möglichkeiten: (1) `sat`, wenn die Formeln erfüllbar sind, (2) `unsat`, wenn sie nicht erfüllbar sind und (3) `unknown`, wenn der Solver weder das Eine noch das Andere feststellen kann. In dem Fall, dass `sat` zurückgegeben wird, kann mit dem Befehl `get-model` eine Interpretation der Deklarationen abgerufen werden, die alle Formeln wahr macht. Für obiges Beispiel sieht die Antwort von Z3 folgendermaßen aus:

```
(model
  (define-fun a () Int
```

```
    1337)
  (define-fun fun ((x!1 Int)) Bool
    (ite (= x!1 0) true
      true))
)
```

Dargestellt werden diese Interpretationen, indem Definitionen mittels `define-fun` angegeben werden. In dem Modell hat `a` also den Wert 1337. Für die Definition von `fun` wird hier das If-Then-Else Statement `ite` benutzt.

Solche Definitionen können auch von dem Benutzer angegeben werden. Die Funktion `fun` könnte zum Beispiel so definiert sein:

```
(define-fun fun ((x!1 Int)) Bool
    (= x!1 42))
```

Betrachten wir mit dieser Definition noch einmal das Beispiel mit den beiden Assertions von oben, liefert das `check-sat` Kommando diesmal `unsat`:

```
(declare-const a Int)
(define-fun fun ((x!1 Int)) Bool
    (= x!1 42))
(assert (> a 1336))
(assert (fun a))
(check-sat)
(get-model)
```

Da die Formeln hier nun unerfüllbar sind, liefert `get-model` sogar einen Fehler, weil keine Interpretation existiert:

```
(error "line 7 column 14: model is not available")
```

Konfigurieren kann man den Solver mit dem Befehl `set-option`. Durch die Option `:produce-models` aktiviert das Erstellen von Modellen und muss gesetzt werden, bevor Deklarationen oder Assertions eingegeben werden. Z3 hat diese standardmäßig eingeschaltet, für andere SMT-Solver muss sie eventuell gesetzt werden, damit der `get-model` Befehl benutzt werden kann. Die Logik, die die zugrunde liegenden Theorien bestimmt, wird über `set-logic` gesetzt. Quantorenfreie lineare Integer-Arithmetik wird mit `QF_LIA` bezeichnet:

```
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-fun ...)
(assert ...)
(check-sat)
(get-model)
```

Auch die Allgemeingültigkeit von Formeln kann untersucht werden. Eine Formel ist allgemeingültig, wenn für sie jede beliebige (sinnvolle) Interpretation erfüllbar ist. In Listing 4.2 werden zwei Variablen `a` und `b` deklariert und eine Funktion `demorgan` definiert, die das De Morgansche Gesetz darstellt. Dieses ist also gültig, wenn die Negation – angegeben über die Assertion `(not demorgan)` – unerfüllbar ist. Wie erwartet gibt der Solver `unsat` aus.

Die bis hier vorgestellten Befehle reichen bereits aus, um damit in SMT-LIB Finite-Domain-Probleme zu modellieren. Dazu betrachten wir an dieser Stelle mit dem Puzzle `SEND + MORE = MONEY` ein weiteres wohlbekanntes Beispiel aus der Familie der Finite-Domain-Probleme. Dabei müssen den Buchstaben S, E, N, D, M, O, R und Y jeweils unterschiedliche Ziffern aus der Menge $\{0, \ldots, 9\}$ zugewiesen werden, so dass die Gleichung stimmt. Für S und M darf die 0 nicht gewählt werden, weil Zahlen normalerweise nicht damit beginnen. Ein mögliches Modell mit SMT-LIB des Puzzles ist in Listing 4.3 zu sehen.

Anstelle des `get-model` Befehl wird hier allerdings `get-value` benutzt. Wie oben gesehen, wird das Modell in Form von Definitionen angegeben. Betrachten wir allerdings lediglich Finite-Domain-Probleme, enthält das Modell für uns überflüssige Informationen, da wir wissen, dass unsere Variablen alle von der Sorte Integer sind. Bei dem `get-value` Kommando können die Variablen angegeben werden, deren Werte uns

```
(declare-const a Bool)
(declare-const b Bool)
(define-fun demorgan () Bool
    (= (and a b)
       (not (or (not a) (not b)))))
(assert (not demorgan))
(check-sat)
```

Listing 4.2: Beweis der Allgemeingültigkeit des De Morganschen Gesetzes

interessieren, und es werden nur die relevanten Informationen ausgegeben – nämlich der Name und der Wert:

```
((s 9) (e 5) (n 6) (d 7) (m 1) (o 0) (r 8) (y 2))
```

Wie sich mit der oben erwähnten Logik `QF_LIA` schon erahnen lässt, können mit SMT-LIB auch nicht lineare Probleme und quantifizierte Formeln auf Erfüllbarkeit untersucht werden. Da nicht lineare Integer-Arithmetik allerdings unentscheidbar ist, kann es vorkommen, dass der `check-sat` Befehl `unknown` zurückgibt oder nicht terminiert. Als einfaches Beispiel für eine prädikatenlogische Formel, in der sowohl All- als auch Existenzquantor vorkommen, soll die folgende Formel dienen:

$$\forall x \, \exists y (x < y) \vee \exists y \, \forall x (y < x \vee y = x)$$

Die Symbole $<$ und $=$ sollen dabei die standardmäßigen Interpretationen aus der Integer-Arithmetik haben. Eine direkte Umsetzung der Formel ist in Listing 4.4 zu sehen, die Allgemeingültigkeit wird analog zu dem De Morgan Beispiel gezeigt.

```
(declare-const s Int)
⋮
(declare-const y Int)
(assert (and (<= 0 s) (<= s 9)))
⋮
(assert (and (<= 0 y) (<= y 9)))
(assert (distinct s e n d m o r y))
(assert (not (= s 0)))
(assert (not (= m 0)))
(assert (=
    (+ (+ (* 1000 s) (* 100 e) (* 10 n) d)
       (+ (* 1000 m) (* 100 o) (* 10 r) e))
    (+ (* 10000 m) (* 1000 o) (* 100 n) (* 10 e) y)))
(check-sat)
(get-value (s e n d m o r y))
```

Listing 4.3: SMT-LIB-Version des `SEND + MORE = MONEY` Puzzles

```
(define-fun quants () Bool
    (or (forall ((x Int))
                (exists ((y Int))
                        (< x y)))
        (exists ((x Int))
                (forall ((y Int))
                        (or (< x y)
                            (= x y))))))
(assert (not quants))
(check-sat)
```

Listing 4.4: Formel mit Quantoren in SMT-LIB

Teil III.

Implementierung

5. Entwurf

Ziel dieser Arbeit ist die Entwicklung einer Schnittstelle für externe Constraint-Solver für die Programmiersprache Curry, um Finite-Domain-Constraint-Probleme zu lösen. Der verwendete Solver soll leicht austauschbar sein, so dass ein Problem auf einfache Weise mit unterschiedlichen Solvern gelöst werden kann. Getestet wird die Schnittstelle vorwiegend mit dem SMT-Solver Z3 und dem SAT-Solver Lingeling.

Auf Grund des unterliegenden Prolog-Systems bietet PAKCS bereits das Lösen von FD-Problemen über das in Abschnitt 3.1 vorgestellte Modul an. Da hier allerdings „externe“ – in PAKCS eingebaute – Funktionen benutzt werden, kann dieses nicht mit KiCS2 eingesetzt werden. Haskell selbst bietet auch keine nativen Möglichkeiten zum Lösen solcher Probleme, so dass wir in dieser Arbeit untersuchen, wie sich externe, eigenständige Solver über eine Bibliothek in Curry integrieren lassen.

Damit die zu entwickelnde Bibliothek in mehreren Curry-Implementierungen benutzt werden kann, wird diese komplett in Curry geschrieben. Der Einsatz externer Funktionen, geschrieben in Prolog beziehungsweise Haskell, ist in PAKCS und KiCS2 zwar möglich, verlangt aber eine jeweilige Programmierung und ist nicht portabel.

Grundlage des Ganzen bildet ein Modell für Finite-Domain-Probleme in Curry. Es müssen Constraints dargestellt werden können, die vorrangig aus arithmetischen und Vergleichsoperationen bestehen, und es muss eine Möglichkeit geben, frische Variablen einzuführen. Zur Repräsentation der Constraints sind entsprechende Curry-Datenstrukturen nötig, die über Smart-Konstruktoren verwendet werden.

Die Curry-Datenstruktur kann nun als Schnittstelle dienen, um konkrete Solver wie SMT- und SAT-Solver anzubinden. Unsere Repräsentation in Curry muss in ein weiteres Modell überführt werden, mit dem der jeweilige Solver arbeiten kann, im Fall von SMT-Solvern wird also ein Modell der Sprache SMT-LIB benötigt. Da FD-Probleme betrachtet werden, genügt es, sich hier auf den Sprachausschnitt zu beschränken, der zur Modellierung von FD-Problemen nötig ist. Ist das FD-Modell in die Curry-Datenstrukturen für das SMT-Modell überführt, kann dieses mittels Pretty Printer in eine Datei – oder praktischer, in die Standardeingabe des Solver-Prozesses – geschrieben werden und der Solver beginnt zu arbeiten. Die gefundenen Lösungen müssen geparst

und entsprechend aufbereitet in Curry zurückgegeben werden. Über geeignete Optionen können Einstellungen wie das Abfragen einer bestimmten Anzahl an Lösungen oder das Speichern des erzeugten Programmes in der Solversprache vorgenommen werden.

Da in dieser Arbeit das e**X**terne Lösen von **FD**-Problemen behandelt wird, findet sich die komplette Implementierung in einer hierarchischen Struktur im Namensraum `XFD`.

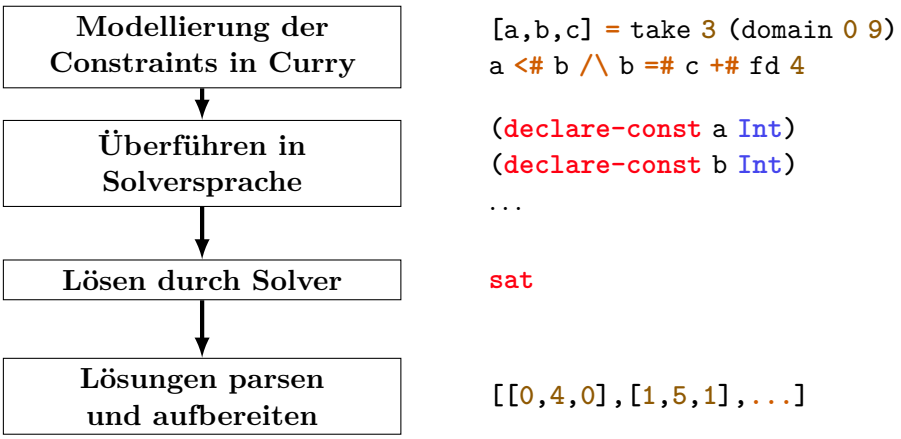

Abbildung 5.1.: Schematischer Ablauf des Lösevorgangs von Constraint-Problemen

6. Finite-Domain-Constraints in Curry

Der folgende Abschnitt beschreibt den Aufbau der Finite-Domain-Constraint-Bibliothek in Curry. Dies umfasst zunächst die von der Bibliothek bereitgestellten Constraints, anschließend wird auf die zugrunde liegenden Curry-Datentypen eingegangen.

6.1. Finite-Domain-Bibliothek

Die Finite-Domain-Constraint-Bibliothek ist sehr stark an der in PAKCS enthaltenen Bibliothek `CLP.FD` orientiert und befindet sich in dem Modul `XFD.FD`. Darin werden die folgenden Funktionen über FD-Ausdrücke, deren Signaturen in Listing 6.1 aufgeführt sind, definiert:

- Besondere Bedeutung hat das `domain`-Constraint, mit dem durch `domain min max` eine unendliche Liste von frischen FD-Variablen mit einem Wertebereich von `min` bis `max` generiert wird. Mithilfe der in Currys Prelude definierten Funktion `take` kann die benötigte Anzahl an Variablen abgegriffen werden. Die Benutzung von `domain` folgt immer dem folgenden Schema:

  ```
  let vs = take n (domain min max)
  in  expression
  ```

  Werden Variablen mit unterschiedlichen Wertebereichen benötigt, erhält man diese mit entsprechenden weiteren Aufrufen von `domain`.

  Da wie in Abschnitt 4.2 gesehen später alle Variablen für den SMT-Solver einen Namen benötigen, soll an dieser Stelle bereits dafür gesorgt werden, dass die

```
(+#),(-#),(*#) :: FDExpr -> FDExpr -> FDExpr
domain         :: Int    -> Int    -> [FDExpr]
fd             :: Int    -> FDExpr
abs            :: FDExpr -> FDExpr
```

Listing 6.1: Übersicht über die FD-Ausdrücke der Bibliothek

FD-Variablen durch das `domain`-Constraint jeweils einen möglichst einzigartigen Namen zugewiesen bekommen. Auf diese Thematik wird in Abschnitt 6.2 genauer eingegangen, wenn wir die Implementierung der Funktionen betrachten.

- Mit der Hilfsfunktion `fd` werden Integer in FD-Ausdrücke transformiert, so dass diese in unserem Modell benutzt werden können. Dies ist nötig, da in den im Rahmen dieser Arbeit betrachteten Curry-Implementierungen Typklassen bisher nicht unterstützt werden.
- Mit `(+#)`, `(-#)` und `(*#)` werden die bekannten arithmetischen Operationen bezeichnet, `abs` liefert den Betrag eines Ausdrucks.

Die Constraints in Listing 6.2 setzen sich nun teilweise aus diesen FD-Ausdrücken zusammen:

- In der ersten Zeile sind die in der Integer-Arithmetik definierten Vergleichsoperatoren zu finden. Diese verknüpfen jeweils zwei Ausdrücke mit dem entsprechenden relationalen Operator.
- `(/\)` und `(\/)` sehen aus wie die aus der Logik bekannten Junktoren $\wedge$ und $\vee$ und verknüpfen dementsprechend zwei Constraints mit Und beziehungsweise Oder. Mittels `andC` und `orC` kann jeweils eine Liste von Constraints mit dem jeweiligen Junktor kombiniert werden.
- Durch `notC` wird die Negation eines Constraints ausgedrückt.
- Mit `allDifferent vs` wird erzwungen, dass die FD-Ausdrücke n in der Liste `vs` paarweise verschiedene Werte haben.

```
(=#),(/=#),(<#),(<=#),(>#),(>=#) :: FDExpr -> FDExpr -> FDConstr
(/\),(\/)       :: FDConstr -> FDConstr -> FDConstr
notC            :: FDConstr -> FDConstr
allDifferent    :: [FDExpr] -> FDConstr
sum             :: [FDExpr] -> FDRel    -> FDExpr  -> FDConstr
scalarProduct   :: [FDExpr] -> [FDExpr] -> FDRel   -> FDExpr -> FDConstr
count           :: FDExpr   -> [FDExpr] -> FDRel   -> FDExpr -> FDConstr
true, false     :: FDConstr
andC, orC       :: [FDConstr] -> FDConstr
allC            :: (a -> FDConstr) -> [a] -> FDConstr
```

Listing 6.2: Übersicht über die Finite-Domain-Constraints der Bibliothek

- Die drei Constraints `sum`, `scalarProduct` und `count` machen Aussagen über Listen von Werten.

  Durch `count v vs rel c` wird die Anzahl der Variablen aus `vs`, die gleich `v` sind, über `rel` zu `c` in Relation gesetzt.

- Das immer erfüllte und das immer unerfüllte Constraint heißt `true` beziehungsweise `false`.

Auf die Implementierung der Funktionen wird in Abschnitt 6.2 eingegangen, nachdem die für die Repräsentation benutzten Curry-Datentypen eingeführt wurden.

Zum Lösen eines Constraints sollen die Solver später jeweils drei Funktionen bereitstellen: mit `solveFD` werden die Lösungen nichtdeterministisch berechnet, `solveFDOne` berechnet genau eine Lösung (sofern eine existiert) und `solveFDAll` berechnet eine Liste aller Lösungen.

Mit den bis hier vorgestellten Funktionen betrachten wir erneut das N-Damen-Problem, welches bereits im Grundlagenkapitel in Abschnitt 3.1 behandelt wurde. Die oben beschriebene FD-Bibliothek implementiert die gleichen Schnittstellen wie die bereits in PAKCS vorhandene – erweitert um einige Constraints wie die für Oder-Verknüpfung oder `false` –, so dass für Erläuterung der Hilfsconstraints `allSafe`, `safe` und `noAttack` auf die dortige Beschreibung verwiesen wird. Dies erlaubt später einfaches Austauschen der in dieser Arbeit angeschlossenen Solver auch durch den in PAKCS vorhandenen. In Listing 6.3 wird dieses Mal `solveFDAll` an Stelle von `solveFD` benutzt.

```
queens :: [Option] -> Int -> [[Int]]
queens options n = solveFDAll options qs (allSafe qs)
  where qs = take n (domain 1 n)

allSafe []     = true
allSafe (q:qs) = safe q qs (fd 1) /\ allSafe qs

safe _ []       _ = true
safe q (q1:qs) p = noAttack q q1 p /\ safe q qs (p +# fd 1)

noAttack q1 q2 p = q1 /=# q2 /\ q1 /=# q2+#p /\ q1 /=# q2-#p
```

Listing 6.3: N-Damen-Problem mit neuer Bibliothek

6.2. Repräsentation der Constraints

In den Signaturen der Funktionen in den Listings 6.1 und 6.2 tauchen bereits einige Datentypen auf, die im nachfolgenden Abschnitt beschrieben werden. Diese stellen die interne Repräsentation der Constraints in Curry dar und dienen später gleichzeitig dazu, die Constraints beispielsweise nach SMT-LIB oder DIMACS zu überführen, damit Solver wie Z3 und Lingeling damit arbeiten können.

Unsere Finite-Domain-Constraints werden zu großen Teilen aus FD-Ausdrücken wie Variablen und Integer-Konstanten aufgebaut. Dafür wird der in Listing 6.4 aufgeführte Curry-Datentyp `FDExpr` eingeführt. Ein FD-Ausdruck hat demnach einen der folgenden vier Werte:

- Eine Variable mit einem Namen, oberer und unterer Grenze des Wertebereiches sowie einem Wert,
- ein Wert vom Typ Integer,
- die Verknüpfung zweier FD-Ausdrücke durch Addition, Subtraktion oder Multiplikation oder
- der Betrag eines FD-Ausdruckes.

Mit den vorgestellten Datentypen betrachten wir nun die Implementierung der Funktionen `fd` und `abs`, der arithmetischen Operatoren und der `domain`-Funktion. Die ersten beiden Funktionen nehmen jeweils einen Parameter, den Integer beziehungsweise FD-Ausdruck:

```
fd :: Int -> FDExpr
fd i = FDInt i
```

```
data FDExpr = FDVar     String Int Int Int
            | FDInt     Int
            | FDBinExp  FDOp FDExpr FDExpr
            | FDAbs     FDExpr

data FDOp = Plus | Minus | Times
```

Listing 6.4: Repräsentation der FD-Expressions in Curry

```
abs :: FDExpr -> FDExpr
abs e = FDAbs e
```

Da die arithmetischen Operatoren alle analog zueinander sind, wird hier die Addition stellvertretend für die beiden anderen vorgestellt:

```
(+#) :: FDExpr -> FDExpr -> FDExpr
a +# b = FDBinExp Plus a b
```

Addition, Subtraktion und Multiplikation werden jeweils durch `FDBinExp` als binäre Verknüpfung zweier FD-Ausdrücke mit dem entsprechenden arithmetischen Operatoren dargestellt, die in `FDOp` definiert sind.

Die `domain`-Funktion bekommt die obere und untere Grenze des Wertebereiches als Parameter und liefert eine unendliche Liste von FD-Variablen. Dabei muss – wie bereits erwähnt – auch ein Name vergeben werden. Dieser soll einzigartig sein, da Deklarationen (vom gleichen Typ) in SMT-LIB unterschiedliche Namen haben müssen. Doppelte Deklaration führt zur Ausgabe einer Fehlernachricht.

In Listing 6.5 ist die Implementierung der Funktion `domainWPrefix` aufgeführt. Diese bekommt neben den Werten für die Domain ein Präfix für die Namen der FD-Variablen. Die Namen enthalten zusätzlich die untere und obere Grenze sowie eine laufende Nummer. Mithilfe dieser Funktion ist `domain` folgendermaßen definiert:

```
domain :: Int -> Int -> [FDExpr]
domain = domainWPrefix "fdv_"
```

Für die meisten Fälle ist es ausreichend, mit `domain` zu arbeiten. Es ist lediglich zu beachten, dass verschiedene Aufrufe mit den gleichen Grenzen natürlich gleiche Namen für die Variablen erzeugen. Beim Programmieren muss daher mit Sorgfalt vorgegangen

```
domainWPrefix :: String -> Int -> Int -> [FDExpr]
domainWPrefix prf lo up = genFDVars (nameList prefix 1) lo up
  where
    prefix = prf ++ (show lo) ++ "_" ++ (show up) ++ "_"
    nameList pr i = (pr ++ (show i)) : nameList pr (i+1)
    genFDVars (n:ns) l u = FDVar n l u _ : genFDVars ns l u
    genFDVars []     _ _ = []
```

Listing 6.5: Implementierung der Funktion `domainWPrefix` in Curry

```
data FDRel = Equ | Neq | Lt | Leq | Gt | Geq

data FDConstr = FDTrue
              | FDFalse
              | FDRelCon  FDRel FDExpr FDExpr
              | FDAnd     FDConstr FDConstr
              | FDOr      FDConstr FDConstr
              | FDNot     FDConstr
              | FDAllDiff [FDExpr]
              | FDSum     [FDExpr]          FDRel FDExpr
              | FDScalar  [FDExpr] [FDExpr] FDRel FDExpr
              | FDCount   FDExpr   [FDExpr] FDRel FDExpr
```

Listing 6.6: Repräsentation der Finite-Domain-Constraints in Curry

werden, indem beispielsweise alle benötigten Variablen mit gleichem Wertebereich auf einmal abgegriffen werden. Diese Liste kann dann durchgereicht werden. Ein Beispiel dafür ist in Anhang C beim Sudoku in der Funktion `readSudoku` zu sehen. Alternativ kann `domainWPrefix` direkt genutzt werden.

Mit dem nun vorhandenen Typen `FDExpr` kann die Datenstruktur für die Constraints definiert werden. Diese ist in Listing 6.6 zu finden und umfasst die restlichen der im vorherigen Abschnitt vorgestellten Funktionen.

Für alle Funktionen der Bibliothek, die ein Constraint darstellen, besitzt der Typ `FDConstr` einen adäquaten Konstruktor. Die Vergleichsoperatoren werden durch den `FDRelCon`-Konstruktor dargestellt, der auf die gleiche Weise wie bei den oben beschriebenen arithmetischen Operatoren den zur Relation passenden `FDRel`-Wert erhält. Da in dem in PAKCS bereits enthaltenen Modul jeder der Operatoren `(=#)`, `(/=#)`, `(<#)`, `(<=#)`, `(>#)` und `(>=#)` einen eigenen Wert erhält, haben wir dies hier ebenfalls auf diese Weise implementiert. Die Alternative ist, für die Relationen „größer“ und „größer-gleich“ keine gesonderten Werte zu nutzen, sondern diese durch Vertauschen der Argumente auf `Lt` beziehungsweise `Leq` abzubilden. Beispielhaft wollen wir die Curry-Implementierung von `(=#)` betrachten:

```
(=#) :: FDExpr -> FDExpr -> FDConstr
x =# y = FDRelCon Equ x y
```

Ein Curry-Ausdruck wie `x =# y +# fd 2 where [x,y] = take 2 (domain 1 9)` wird damit auf den folgenden Term abgebildet – die freien Variablen, die die Werte der beiden FD-Variablen `x` und `y` repräsentieren, sind hier mit dem Unterstrich bezeichnet:

```
FDRelCon Equ (FDVar "fdv_1_9_1" 1 9 _)
             (FDBinExp Plus (FDVar "fdv_1_9_2" 1 9 _) (FDInt 2))
```

Die übrigen Constraints bilden dann in gleicher Weise auf die entsprechenden Konstruktoren ab:

```
(/\) :: FDConstr -> FDConstr -> FDConstr
c1 /\ c2 = FDAnd c1 c2

sum :: [FDExpr] -> RDRel -> FDExpr -> FDExpr
sum vs rel v = FDSum vs rel v
```

Abschließend fehlen noch die Informationen über die Operatorrangfolge der Infixoperatoren wie `(*#)`, `(/=#)` und `(/\)`. Für die Arithmetik gilt auch hier die Konvention Punkt-vor-Strichrechnung, wohingegen die relationalen Vergleichsoperatoren alle gleich gewichtet sind. Die Kombination von Constraints mittels Und beziehungsweise Oder betreffend haben wir uns für eine stärkere Bindung der Konjunktion entschieden. In Curry kann die Bindungsstärke sowie Links- beziehungsweise Rechtsassoziativität mit den Schlüsselwörtern `infix`, `infixl` und `infixr` angegeben werden:

```
infixl 7 *#
infixr 3 /\
```

7. Anschluss von SMT-Solvern

Nachdem jetzt eine Bibliothek existiert, mit der Finite-Domain-Constraints in Curry formuliert werden können, wird ein Solver benötigt, der solche Probleme lösen kann. Mit Z3 gibt es einen solchen Solver, der – wie in Abschnitt 4.2 dargestellt – mittels SMT-LIB genutzt werden kann, um Lösungen für FD-Probleme zu finden.

Dieser Abschnitt beschreibt die Entwicklung einer Schnittstelle zu SMT-Solvern. Der erste Schritt ist die Umsetzung von ausgewählten Teilen der SMT-LIB-Sprache als Curry-Datentypen, in die die Finite-Domain-Constraints überführt werden.

Über einen Pretty Printer kann das SMT-LIB-Modell in die Standardeingabe des Solvers geschrieben werden. Für die Ausgaben des Solvers wird ein Parser verwendet und die gefundenen Lösungen werden entsprechend aufbereitet in Curry zurückgegeben.

Erzeugt wird das Programm in SMT-LIB, indem die Darstellung der Finite-Domain-Constraints aus dem vorherigen Kapitel in geeignete Konstrukte überführt wird.

Der letzte Schritt ist dann die Konfiguration eines Solvers und die Kommunikation mit diesem.

Die einzelnen, im Folgenden beschriebenen Module werden in einer hierarchischen Struktur geordnet. Auch für die Implementierung anderer Sprachen als SMT-LIB bietet sich eine solche Hierarchie an, um die Aufgabenbereiche modular zu gestalten:

- Das Modul `XFD.SMTLib` stellt die Funktion bereit, die die Kommunikation mit dem SMT-Solver regelt. Die restlichen Module sind als `XFD.SMTLib.`*Name* eingeordnet.
- Die Implementierung der Datentypen, mit denen SMT-LIB in Curry dargestellt wird, befindet sich in `Types`. Für einfacheres Erzeugen der Datenterme bietet das Modul `Build` entsprechende Operationen an.
- Die Formatierung des in SMT-LIB erzeugten Programmes übernimmt der Pretty Printer `Pretty`, für das Parsen der Ausgaben des Solvers stehen die Module `Scanner` und `Parser` zur Verfügung. Da wir – wie weiter unten ersichtlich wird – zwei verschiedene Parser implementiert haben, wird auf deren Modulnamen dort extra eingegangen.

- Funktionen zum Überführen des FD-Modells in das SMT-LIB-Modell werden in dem Modul `FromFD` umgesetzt.

Durch diese Struktur ist das Wiederverwenden der einzelnen Module an anderer Stelle einfach möglich.

7.1. Darstellung von SMT-LIB

Anstatt ein komplettes Modell von SMT-LIBin Curry zu erzeugen, haben wir uns lediglich auf einen Sprachausschnitt (und die Theorie der Integer) beschränkt, der zum Modellieren von FD-Problemen nötig ist. Der SMT-LIB Standard [3] gibt die konkrete Syntax in Form einer EBNF an.

Bei der Recherche des Themas SMT sind wir auf das Haskell-Paket Hsmtlib[6] gestoßen, SMT-LIB nutzt, um Funktionen zur Interaktion mit einigen Solvern bereitzustellen, darunter auch Z3. Von diesem wiederum wird ein von den gleichen Autoren geschriebenes Paket namens Smtlib[7] benutzt, das Parser für SMT-LIB sowie Haskell-Datenstrukturen für die Sprache implementiert. Da Haskell und Curry einander sehr ähnlich sind, konnten wir uns daran orientieren.

Die Darstellung erfolgt durch passend eingeführte Curry-Datentypen und folgt lose der gegebenen Grammatik. An davon abweichenden Stellen wird auf den jeweiligen Grund dafür eingegangen.

7.1.1. Commands

Um Anweisungen an einen SMT-Solver schicken zu können, sind Befehle wie die in Abschnitt 4.2.2 benutzten nötig. Der in Anhang A aufgeführte Ausschnitt zeigt die Syntax aller von uns benötigten Befehle. Dazu zählen zum Beispiel `assert` zum Erzeugen von Assertions oder `check-sat` zum Prüfen auf Erfüllbarkeit. Für diese Befehle wird der Datentyp `Command` definiert. Für jeden Befehl gibt es einen Konstruktor, diese sind in Listing 7.1 aufgeführt.
Produktionen des Nichtterminalsymbols ⟨*symbol*⟩ stellen wir einfach durch Currys `String` Datentyp dar. Die einzige Option, die für einige Solver benötigt wird, ist `:produce-models`:

```
data Option = ProduceModels Bool
```

[6] https://hackage.haskell.org/package/Hsmtlib
[7] https://hackage.haskell.org/package/SmtLib

```
data Command = Assert Term
             | DeclareConst String Sort
             | CheckSat
             | Echo String
             | Exit
             | GetValue [Term]
             | Pop Int
             | Push Int
             | SetLogic String
             | SetOption SMTOption
```

Listing 7.1: Darstellung der nötigen SMT-LIB Befehle in Curry

Über den Datentyp `Term` wird das eigentliche Finite-Domain-Constraint in SMT-LIB abgebildet, da mit diesen Integer, Variablen und die Anwendung von Operationen wie der Addition oder Prüfung von Gleichheit notiert werden. Die Darstellung in Curry entspricht direkt der von ⟨*term*⟩:

```
data Term = TermSpecConstant SpecConstant
          | TermQualIdentifier QualIdentifier
          | TermQualIdentifierT QualIdentifier [Term]

data QualIdentifier = QIdentifier Identifier
data SpecConstant   = SpecConstantNumeral Int
```

Listing 7.2: Darstellung von SMT-LIB-Termen in Curry

Die Verschachtelung der einzelnen Datentypen produziert allerdings teilweise unpraktisch lange Datenterme. Der SMT-LIB-Ausdruck `(= v 42)` hat folgende Repräsentation:

```
TermQualIdentifierT (QIdentifier (ISymbol "="))
    [ TermQualIdentifier (QIdentifier (ISymbol "v"))
    , TermSpecConstant (SpecConstantNumeral 4)
    ]
```

Um solche Ausdrücke in übersichtlicher Form darzustellen, werden Funktionen eingeführt, die die Konstruktion übernehmen. So lässt sich das gleiche Beispiel in kompakter Form notieren:

```
termQIdentT "=" [termQIdent "v", termSpecConstNum 4]
```

Die Definition von `termQIdentT` zeigt, dass hier lediglich die Datenkonstruktoren entsprechend angewandt werden:

```
termQIdentT :: String -> [Term] -> Term
termQIdentT sym ts = TermQualIdentifierT (qIdent sym) ts
```

7.1.2. Responses

Die Ausgaben des Solvers sollen geparst und entsprechend in eine Curry-Darstellung überführt werden. Anstatt hier die Grammatik eins zu eins umzusetzen, führen wir den Typ `CmdResponse` ein:

```
data CmdResponse = CmdGenResponse GenResponse
                 | CmdCheckSatResponse CheckSatResponse
                 | CmdGetValueResponse GetValueResponse

data GenResponse = Error String

data CheckSatResponse = Sat | Unsat | Unknown

type GetValueResponse = [ValuationPair]
data ValuationPair = ValuationPair Term Term
```

Listing 7.3: Curry-Darstellung der Command Responses

Anhand der eingelesenen Antwort des Solvers wird dann mittels Case-Ausdruck entschieden, wie weiter vorgegangen wird. In Abschnitt 4.2.2 haben wir gesehen, dass es zu einem Fehler kommt, wenn die Werte von Variablen im Falle einer nicht erfüllbaren Formel abgefragt werden. Es muss später also erst sichergestellt werden, dass die Antwort `sat` lautet:

```
case answer of
  CmdGenResponse (Error msg) -> ... handle error ...
  CmdCheckSatResponse Sat    -> ... get values ...
  ⋮
```

7.2. Pretty Printing und Parsing

Das eben beschriebene Modell der SMT-LIB Kommandos in Curry muss formatiert werden, damit es in den Solver eingegeben werden kann. Dies übernimmt ein sogenannter Pretty Printer. Die Antworten des Solvers werden über einen Parser in eine `CmdResponse` umgewandelt. Im folgenden Abschnitt werden der benutzte Pretty Printer und zwei verschiedene Parser vorgestellt. Der erste Parser verwendet das nichtdeterministische Parser-Modul, welches bei KiCS2 und PAKCS mitgeliefert wird. Der Zweite nutzt eine eigene Implementierung für Recursive Descent Parser.

7.2.1. Pretty Printer

Für Pretty Printing bietet das von PAKCS und KiCS2 mitgebrachte `Pretty` bereits alle erforderlichen Konstrukte. Die Funktion

```
pPrint :: Doc -> String
```

wandelt die abstrakte Repräsentation eines Dokuments in eine Zeichenkette um. Erzeugt wird ein solches Konstrukt des Typs `Doc` durch Funktionen wie `text` und `parens`, die eine Zeichenkette beziehungsweise einen von Klammern umschlossenes Dokument darstellen.

Die im vorherigen Abschnitt definierten Curry-Datenkonstruktoren werden über entsprechende `pp` (pretty-print) Funktionen in die abstrakte Beschreibung umgewandelt. Betrachten wir zunächst den Funktion `ppTerm`, welche SMT-LIB-Terme umsetzt:

```
ppTerm :: Term -> Doc
ppTerm term = case term of
  TermSpecConstant    sc        -> ppSpec sc
  TermQualIdentifier  qi        -> ppQi qi
  TermQualIdentifierT qi terms  -> parens $
                                     ppQi qi <+>
                                     joinA (ppTerm) terms
```

Dies entspricht exakt der Form, die durch die konkrete Syntax (siehe Anhang A) vorgegeben ist. Die restlichen Funktionen sind analog hierzu implementiert:

```
ppQi :: QualIdentifier -> Doc
ppQi qi = case qi of
  QIdentifier i -> ppIden i
```

```
ppIden :: Identifier -> Doc
ppIden i = case i of
  ISymbol s -> text s
```

Der Befehl `declare-const` wurde erst in Version 2.5 des SMT-LIB-Standards als Alias für `declare-fun` eingeführt. Da es unter Umständen Solver gibt, die diese Version noch nicht unterstützen, wird der `DeclareConst`-Datentyp folgendermaßen formatiert:

```
ppCmd :: Command -> Doc
ppCmd cmd = parens $ case cmd of
  DeclareConst name sort  -> text "declare-fun" <+> text name
                                <+> text "()" <+> ppSort sort
  ⋮
```

Semantisch besteht kein Unterschied zwischen den beiden Kommandos und es ist sichergestellt, dass es von jedem SMT-LIB v2 kompatiblen Solver verstanden wird.

Abschließend steht die Funktion `showSMTLib` bereit, die eine Liste von Befehlen in einen einzigen String umwandelt:

```
showSMTLib :: [Command] -> String
showSMTLib = unlines . map (pPrint . showCmd)
```

7.2.2. Scanner

Mit dem Pretty Printer existiert nun die Hinrichtung für die Kommunikation mit SMT-Solvern. Für die Rückrichtung ist ein Parser notwendig, der die Ausgaben in den oben beschriebenen Typen `CmdResponse` überführt. Der erste Schritt dafür ist die lexikalische Analyse, durchgeführt von einem sogenannten Scanner. Dieser zerlegt die Antwort des Solvers – eine einfache Zeichenkette – in Lexeme, auch Token genannt. Aus dem Schlüsselwort `sat` wird so das Token `KW_sat`, die Klammern `(` und `)` werden zu `LParen` beziehungsweise `RParen`. Namen von Variablen und Zahlwerte, wie sie als Ausgabe von `get-value` auftreten, werden entsprechen durch die Token `Id` und `Number` dargestellt. Diese nehmen einen String beziehungsweise einen Integer als Argument.

Die Token werden anschließend in der syntaktischen Analyse durch einen Parser verarbeitet. Dieser erkennt aus der Folge der Token die syntaktischen Einheiten einer Sprache.

7.2.3. Nichtdeterministischer Parser

In den System Bibliotheken von KiCS2 und PAKCS ist ein Modul namens `Parser` enthalten. Dieses implementiert Parserkombinatoren nach [7] mithilfe von freien Variablen und Unifikation. Unterschieden wird hier zwischen zwei Arten von Parsern. Parser ohne Repräsentation – im Folgenden nur Parser genannt – konsumieren Token und geben die nicht verarbeiteten Token zurück. Parser mit Repräsentation nehmen zusätzlich ein Argument, welches im Normalfall eine freie Variable ist, an die die Repräsentation nach dem Parsen gebunden wird.

Die Funktion `parse` nimmt als Eingabe die vom Solver ausgegebene Zeichenkette und zerlegt diese zunächst mit dem Scanner in eine Liste von Token. Dann wird diese Liste vom Parser verarbeitet und der gefundene, an `val` gebundene Wert zurückgegeben:

```
parse :: String -> Either ParseError CmdResponse
parse s = let ts  = scan s
              val free
              parseResult = parseCmdResult val ts == []
          in case parseResult of
            False -> Left "incomplete parse"
            True  -> Right val
```

Listing 7.4: `parse`-Funktion des nichtdeterministischen Parsers für SMT-LIB

Ein Problem bei dem auf diese Weise programmierten Parser war die in den Grundlagen angesprochene call time choice. Der Kombinator `some` wendet einen Parser mit Repräsentation einmal oder öfter an und gibt die einzelnen Repräsentationen als Liste zurück. Betrachten wir die folgende beispielhafte Ausgabe des Befehls `(get-value (v1 v2))`:

```
((v1 42) (v2 23))
```

Der in Listing 7.5 abgebildete Parser für eine `GetValueResponse` findet für diese Ausgabe keinen Wert. Beim ersten Anwenden von `parseValuePair` durch `some` werden die Variablen `t1` und `t2` gebunden. Aufgrund der call time choice wird der Parser von allen Aufrufen geteilt und somit auch die Variablen, die nun aber an die ersten Werte gebunden sind. Weitere Werte können dann nur akzeptiert werden, wenn diese gleich den ersten sind. Die Ausgabe `((v1 42) (v1 42))` kann erfolgreich geparst werden.

```
parseGetValueResponse :: ParserRep GetValueResponse Token
parseGetValueResponse =  terminal LParen
                      <*> some parseValuePair vp
                      <*> terminal RParen >>> vp
  where vp free

parseValuePair :: ParserRep ValuationPair Token
parseValuePair =  terminal LParen
              <*> parseTerm t1
              <*> parseTerm t2
              <*> terminal RParen >>> ValuationPair t1 t2
  where t1, t2 free

some :: ParserRep rep token -> ParserRep [rep] token
some p = p x <*> (star p) xs >>> (x:xs)        where x,xs free
```

Listing 7.5: Parser für VPs

Durch eine Modifikation von `some` (und selbstverständlich auch `star`) kann dieser Umstand allerdings behoben werden [11]. Anstelle eines Parsers `p` wird eine Funktion `(\_ -> p)` übergeben, die einen frischen Parser `p` zurückgibt, wenn sie ausgewertet wird. Die modifizierte Implementierung nennen wir `mSome`, wobei `mStar` analog transformiert wird:

```
mSome :: (() -> ParserRep rep token) -> ParserRep [rep] token
mSome p = (p ()) x <*> (mStar p) xs >>> (x:xs)     where x,xs free
```

Das Modul für diese Implementierung des Parsers heißt `NDParser`.

7.2.4. Recursive Descent Parser

Der gerade vorgestellte Parser ist zwischenzeitlich in Verdacht geraten, den Solveprozess zu verlangsamen. Im Endeffekt lag das „Problem“ allerdings bei Z3 (dieser Punkt wird später noch einmal aufgegriffen). Trotzdem haben wir uns dazu entschieden, mit dem Modul `RDParser` zusätzlich einen Recursive Descent Parser zu implementieren. Grundlage dafür bietet das Modul `XFD.Parser`, welches die Parserkombinatoren für RD Parser bereitstellt und nachfolgend kurz vorgestellt wird.

Ein Recursive Descent Parser baut auf der Idee auf, dass jedem Grammatiksymbol eine Funktion zum Parsen zugeordnet wird. Diese Funktionen verarbeiten die Token-

```
parseTerm :: Parser Token Term
parseTerm [] = eof []
parseTerm (t:ts) = case t of
  Number i -> yield (termSpecConstNum i) ts
  Id  name -> yield (termQIdent name) ts
  ⋮
```

Listing 7.6: Lookahead für `parseTerm`

liste in der gegebenen Reihenfolge von links nach rechts, wobei Terminalsymbole der Grammatik das entsprechende Token konsumieren.

Da wir auch hier an einer Repräsentation interessiert sind, sieht der Datentyp für Parser

```
type Parser token a = [token] -> Either ParseError ([token], a)
```

dies ebenfalls vor. Ein Parser ohne Repräsentation wird einfach mit dem Unit-Datentyp für die Typvariable `a` realisiert. Dieser hat `()` als einzigen Wert. Ein Beispiel ist `terminal`, welcher lediglich das gegebene Token konsumiert:

```
terminal :: token -> Parser token ()
terminal _ [] = eof []
terminal x (t:ts) = case x == t of
  True  -> Right (ts, ())
  False -> unexpected t ts
```

Über die beiden Kombinatoren `(*>)` und `(<*)` kann gesteuert werden, welche Repräsentation ein zusammengesetzter Parser hat. Sind `p` und `q` zwei Parser, so hat `p <* q` die Repräsentation von `p`. Nehmen wir die mit dem neuen Modul definierte Funktion `parseValuePair`, lässt sich sehen, dass die Token für die Klammern konsumiert werden, die Rückgabe allerdings das Ergebnis von `liftP2` ist:

```
parseValuePair :: Parser Token ValuationPair
parseValuePair =  terminal LParen
              *> liftP2 ValuationPair parseTerm parseTerm
              <* terminal RParen
```

Mit `liftP2` wird eine Funktion mit zwei Parametern auf die Ergebnisse von zwei Parsern angewandt. Hier also `ValuationPair` auf die beiden Terme, die jeweils von `parseTerm` stammen.

Hat ein Nichtterminalsymbol mehrere Produktionen, kann anhand des ersten – oder mehrerer – Tokens entschieden werden, welche davon auszuwählen ist. Pattern Matching und ein Case-Ausdruck machen dies leicht möglich, wie in Listing 7.6 für `parseTerm` zu sehen ist.

7.3. Überführung des FD-Modells

Ähnlich wie der Pretty Printer funktioniert auch das Überführen eines Finite-Domain-Constraints nach SMT-LIB. Ein Constraint setzt sich aus den Typen `FDConstr` und `FDExpr` zusammen. Die Funktionen `convertConstr` und `convertExpr` transformieren diese in einen SMT-LIB `Term`:

```
convertConstr :: FDConstr -> Term
convertExpr   :: FDExpr   -> Term
```

Zusammengesetzt werden die Terme unter Zuhilfenahme der in Abschnitt 7.1.1 beschriebenen Helferfunktionen, mit denen die Konstruktion der Curry Datenterme vereinfacht wird. Die Ausschnitte in den Listings 7.7 und 7.8 zeigen das Vorgehen für einige Konstruktoren von `FDConstr` sowie für die möglichen vier FD-Ausdrücke. Aus Platzgründen sind einige Funktionsnamen verkürzt, die komplette Implementierung von `convertConstr` und `convertExpr` ist in Anhang B zu finden.

Die zwei Fälle für `FDRelCon` sind ein gutes Beispiel für überlappende Muster, wie sie in Abschnitt 2.3 behandelt wurden. In SMT-LIB gibt es keinen Operator für Ungleichheit, diese wird in der Form `(not (= ...))` ausgedrückt. Anhand von `Neq` wird der Fall erkannt und das Constraint auf entsprechende Weise umgeformt. Die restlichen Relationen werden mit der Variable `rel` abgedeckt, die Funktion `rs` bildet die Konstruktoren auf den korrespondierenden String ab: `rs Leq = "<="`.

```
convertConstr :: FDConstr -> Term
convertConstr constr = case constr of
  FDTrue               -> tQI "true"
  FDRelCon   Neq e1 e2 -> convertConstr  $ notC $ e1 =# e2
  FDRelCon   rel e1 e2 -> tQItT (rs rel) $ map (convertExpr)   [e1, e2]
  FDAnd       e1 e2    -> tQItT "and"    $ map (convertConstr) [e1, e2]
   ⋮
```

Listing 7.7: Transformation eines Constraints nach SMT-LIB

```
convertExpr :: FDExpr -> Term
convertExpr expr = case expr of
  FDVar     n _ _ _   -> tQI n
  FDInt     i         -> case i < 0 of
    True  -> tQIT "-" [tSCN (-i)]
    False -> tSCN i
  FDBinExp  op e1 e2  -> tQIT (os op) $ map (convertExpr) [e1, e2]
  FDAbs     e         -> tQIT "abs"   $ [convertExpr e]
```

Listing 7.8: Transformation eines Ausdrucks nach SMT-LIB

```
(=>>) :: SMT -> SMT -> SMT
a =>> b = \cmds -> b $ a cmds

showSMT :: SMT -> String
showSMT cmds = showSMTLib $ cmds []
```

Listing 7.9: Funktionen zum Kombinieren und Ausgeben von SMT-LIB Befehlen

Analog dazu ist die Transformation von `FDBinExp`: den String, der den Operator repräsentiert, liefert die Hilfsfunktion `os`. Bei Integern ist die Darstellung davon abhängig, ob der Wert positiv oder negativ ist. Eine positive Zahl wird direkt in den Typ für Zahlen eingepackt. Eine negative Zahl i hingegen hat in SMT-LIB die Form `(- `n`)`, wobei n der Betrag von i ist.

Die eigentlichen Befehle für den SMT-Solver werden durch den Typen `Command` dargestellt. Um eine Folge solcher Befehle zu erzeugen und weitere hinten anzufügen, führen wir den monadischen Typen

```
type SMT = [Command] -> [Command]
```

ein. Das Kommando für eine Assertion eines Finite-Domain-Constraints wird mit der folgenden Funktion hinten an die Liste der bisherigen Befehle angefügt:

```
assert :: FDConstr -> SMT
assert c = \cmds -> cmds ++ [Assert (convertConstr c)]
```

Der Aufruf `assert c` liefert also eine Funktion, die eine Liste von Kommandos als Parameter nimmt und die diese Liste um den Befehl `Assert` erweitert zurückgibt. Auf die gleiche Weise verhält sich die Funktion für die SMT-LIB Anweisung `(check-sat)`:

```
checkSat :: SMT
checkSat = \cmds -> cmds ++ [CheckSat]
```

Um mehrere solcher monadischen Werte miteinander zu kombinieren, führen wir auch einen bind-Operator `(=>>)` (siehe Listing 7.9) ein. Abschließend wird die Funktion `showSMT` benutzt, um die Liste von Befehlen mit dem Pretty Printer zu formatieren. Das Ergebnis des Curry Ausdrucks `showSMT (assert true =>> checkSat)` ist eine Zeichenkette, die diesem SMT-LIB Code entspricht:

```
(assert true)
(check-sat)
```

Bisher wurde die für Variablen nötige Deklaration noch nicht behandelt. Eine FD-Variable vom Typ `FDVar` enthält die Informationen über den Namen und den Wertebereich. Für eine Liste von FD-Variable nutzen wir die Funktion `declare`, um jeweils einen `DeclareConst` Befehl und einen `Assert` Befehl, der die Einschränkung des Wertebereichs erwirkt, zu der Liste der Kommandos hinzuzufügen. Ein kleines Beispiel mit einer einzelnen Variable und keinen sonstigen Befehlen (leere Liste), `declare [x@(FDVar "x" 0 9 _)] []`, erzeugt folgende Liste:

```
[ DeclareConst "x" (SortId (ISymbol "Int"))
, Assert . convertConstr (fd 0 <=# x /\ x <=# fd 9) ]
```

7.4. Solverkonfiguration

Um einen der externen Solver mit dieser Bibliothek nutzen zu können, muss dieser selbstverständlich auf dem System installiert sein. Außerdem müssen eventuell individuelle, auf den jeweiligen Solver zugeschnittene Einstellungen vorgenommen werden, wie zum Beispiel benötigte Kommandozeilenargumente.

Den Grundstein für diese Einstellungen legt das `XFD.Solver`-Modul. Hier werden die folgenden beiden Typsynonyme definiert:

```
type SolverArgs a = [Option] -> [FDExpr] -> FDConstr -> a
type SolverImpl   = SolverConfig -> SolverArgs (IO [[Int]])
```

Mit `SolverArgs a` werden die Parameter bezeichnet, die die Solve-Funktionen erhalten: eine Liste von Optionen, eine Liste von FD-Variablen und ein Finite-Domain-Constraint. Der Ergebnistyp wird über die Typvariable `a` parametrisiert, damit `SolverArgs` an unterschiedlichen Stellen eingesetzt werden kann, wie im Folgenden deutlich wird.

Die eigentliche Solve-Funktion, über die die Kommunikation mit dem externen Programm läuft, bekommt zusätzlich zu den eben aufgelisteten Parametern eine Beschreibung der Konfiguration eines Solvers. Das Typsynonym `SolverImpl` stellt die Signatur für eine solche Funktion dar, deren Rückgabe eine in die I/O-Monade eingepackte Liste der gefundenen Lösungen ist.

Über die Liste der Optionen können drei Einstellungen vorgenommen werden:

Debugging Der Konstruktor `Debug` nimmt einen Integer als Parameter, über den gesteuert werden kann, wie viele Debug-Informationen ausgegeben werden. Je größer das angegebene „Level“ ist, desto mehr Ausgaben gibt es. Ohne Angabe dieser Option wird 0 als Level genommen und es erfolgen keinerlei Ausgaben.

Persistenz Mittels `Persist s` wird das in die Eingabesprache des Solvers übersetzte Programm in eine Datei mit dem Namen `s` geschrieben.

Anzahl der Lösungen Wie viele Lösungen vom Solver gesucht werden sollen, wird über die drei sich gegenseitig ausschließenden Angaben `All`, `First` und `FirstN n` (mit Integer `n`) gesteuert. Später in der Liste der Optionen befindliche Einträge überschreiben den jeweils Vorherigen. Die Namen der Konstruktoren stehen dabei für alle Lösungen, die erste Lösung beziehungsweise die ersten `n` Lösungen (oder alle, wenn es weniger als `n` gibt).

Optimierung Die Optionen `Minimize` und `Maximize` bekommen jeweils eine FD-Variable. Sie veranlassen, dass der minimale beziehungsweise maximale Wert für die Variable gesucht wird, so dass das Constraint noch erfüllt ist.

Die Standardeinstellung ist, dass keine Debug-Ausgaben erfolgen, das Programm nicht in eine Datei geschrieben wird und alle Lösungen gesucht werden. Kleine Abweichungen zur Anzahl der Lösungen werden unten beschrieben.

Die Konfiguration eines Solvers wird durch den Datentyp `SolverConfig` beschrieben, der einige Einstellungen zulässt. Da der Datentyp für Optionen von SMT-LIB ebenfalls `Option` heißt, wird das `Types` Modul qualifiziert importiert. So kommt es nicht zu Konflikten bei den Typen und die SMT-LIB Typen müssen alle explizit gekennzeichnet werden:

```
import qualified XFD.SMTLib.Types as SMT
data SolverConfig = Config
                  { executable  :: String
                  , flags       :: [String]
```

```
  , solveWith   :: SolverImpl
  , smtOptions  :: Maybe [SMT.Option]
  , smtLogic    :: String
  }
```

Hier sind Informationen über den Solver wie die ausführbare Datei und die Kommandozeilenargumente gespeichert. Darüber hinaus wird die konkrete Funktion `solveWith` angegeben, die die Implementierung des eigentlichen Lösens und der Kommunikation mit dem Solver darstellt. Auf diese wird im nächsten Abschnitt eingegangen. Die beiden Felder `smtOptions` und `smtLogic` enthalten die Optionen und die Logik, die verwendet werden sollen. Durch den Typen `Maybe [SMT.Option]` kann mit dem Konstruktor `Nothing` festgelegt werden, dass keine Optionen gesetzt werden sollen. Eine Standardkonfiguration, in der nur `solveWith` und `executable` gesetzt werden müssen, ist als `defaultConfig` vorhanden (wie oben muss auch der Konstruktor `ProduceModels` qualifiziert werden):

```
defaultConfig :: SolverConfig
defaultConfig = Config
                { flags      = []
                , smtOptions = Just [SMT.ProduceModels True]
                , smtLogic   = "QF_LIA"
                }
```

Aufbauend darauf können die Konfigurationen für die tatsächlichen Solver angegeben werden. Für Z3 beispielsweise müssen die Binary `z3` und die Flags `-in` und `-smt2` angegeben werden, die dafür sorgen, dass Z3 von der Standardeingabe Formeln in SMT-LIB v2 Format einliest. Da es sich hier um einen SMT-Solver handelt, wird außerdem die Funktion `solveSMT` eingestellt, die im nächsten Abschnitt genauer behandelt wird. Wie in Abschnitt 4.2 beschrieben, kann Z3 auch mit nicht linearer Integer-Arithmetik umgehen, so dass die Logik entsprechend eingestellt wird:

```
solverConfig = defaultConfig
               { executable  = "z3"
               , flags       = ["-smt2", "-in"]
               , solveWith   = solveSMT
               , smtLogic    = "QF_NIA"
               }
```

```
solveFDwith :: SolverConfig -> SolverArgs [Int]
solveFDwith cfg opt vars constr =
  let solveF    = solveWith cfg
      solutions = unsafePerformIO $ solveF cfg opt vars constr
  in foldr1 (?) solutions
```

Listing 7.10: Hilfsfunktion zur Implementierung der `solveFD`-Funktion

Jeder Solver soll in einem eigenen Modul implementiert sein, dessen Name die Art des Solvers widerspiegelt. In dem Modul `XFD.Solvers.SMT.Z3` befindet sich demnach der SMT-Solver Z3. Importieren dieses Moduls stellt einen Solver dann wie nachfolgend beschrieben zur Verfügung.

Die eigentliche Benutzung eines Solvers findet über eine Funktion `solveFD` – beziehungsweise die Varianten `solveFDOne` und `solveFDAll` – statt. Jeder Solver muss diese Funktionen implementieren, wobei dies einfach über Definitionen wie

```
solveFD :: SolverArgs [Int]
solveFD = solveFDwith solverConfig
```

vorgenommen werden kann. Neben `solveFDwith` gibt es dementsprechend die analog benannten Funktionen `solveFDAllwith` und `solveFDOnewith`. Die Implementierung ist in Listing 7.10 zu sehen. Aus der Solverkonfiguration wird jeweils die unter `solveWith` abgelegte Funktion neben der Konfiguration noch mit den restlichen Parametern aufgerufen: `opt` ist eine Liste aus den oben beschriebenen Optionen, `vars` sind die Variablen, an deren Lösung wir interessiert sind und `constr` ist das Constraint, welches gelöst werden soll.

Wie bereits am Ende von Abschnitt 6.1 angemerkt, soll `solveFD` – somit auch `solveFDwith` nichtdeterministisch sein. Dies wird hier über den `(?)`-Operator bewerkstelligt, indem `foldr1` genutzt wird, um die gefundenen Lösungen damit als Alternativen zu verknüpfen. Wurde keine Lösung gefunden, ist die Liste `solutions` leer und es wird die dem Curry-System entsprechende Meldung ausgegeben, dass kein Wert gefunden wurde.

Funktionen mit dem Typen `SolverImpl`, der oben eingeführt wurde, geben einen I/O-Wert zurück. Mit der Funktion `unsafePerformIO` aus dem Curry Modul `Unsafe` wird der Wert aus der Monade herausgeholt.

Ähnlich verhält es sich bei `solveFDOnewith`: mit der Funktion `head` wird das erste Element der Lösungsliste zurückgegeben. Ist diese leer, kommt es wiederum zur Meldung,

dass kein Wert gefunden wurde. Zusätzlich wird hier hinten an die Liste `opt` noch die Option `First` geschrieben, die dafür sorgt, dass der Solver auch nur eine Lösung sucht.

Die Funktion `solveFDAllwith` schließlich gibt die Liste der Lösungen unverändert zurück. In dem Fall, dass es keine Lösung gibt, ist die Liste einfach leer. Mit `All` wird hier analog zu `solveFDOnewith` Sorge dafür getragen, dass der Solver versucht, alle Lösungen zu finden. In Anhang B sind die Implementierungen von `solveFDOnewith` und `solveFDAllwith` angegeben.

7.5. Kommunikation mit Solvern

In diesem Abschnitt wird das Herzstück der Implementierung beschrieben, das die Kommunikation mit einem externen SMT-Solver übernimmt. Wie in Kapitel 5 kurz angedeutet, gibt es zwei grundsätzliche Möglichkeiten, mit der Solver-Binary zu arbeiten:

1. Das generierte Programm wird in eine Datei geschrieben, mit der die entsprechende Binary aufgerufen werden kann. Die Ausgaben werden auf die Standardausgabe geschrieben und können geparst und verarbeitet werden. Möchte man eine weitere Lösung finden, muss das im vorherigen Schritt generierte Programm erneut in eine Datei geschrieben werden, dieses Mal allerdings erweitert um Constraints, die die zuletzt gefundene Lösung ausschließen. Dies wird wiederholt, bis der Solver meldet, dass die eingegebenen Formeln unerfüllbar sind.

2. Die Solver-Binary wird so aufgerufen, dass auf Eingaben über die Standardeingabe gewartet wird. Hier kann das generierte Programm eingegeben, nach einer Lösung gesucht und diese von der Standardausgabe gelesen werden. Da die SMT-Solver wie in Abschnitt 4.2.2 beschrieben alle Assertions auf einem internen Stack legen, können die neuen, die gefundene Lösung verbietenden Constraints wiederum über die Standardeingabe eingegeben werden. Somit wird die Binary nur ein einziges Mal aufgerufen, um die gewünschte Anzahl Lösungen zu suchen.

Wir haben uns für den zweiten, interaktiven Ansatz entschieden. Das generelle Vorgehen ist als Pseudocode in Algorithmus 1 skizziert, das Starten des Solvers in einem eigenen Prozess ist dort ausgelassen. Es wird stattdessen angenommen, dass dieser zur Verfügung steht. Auf die konkrete Implementierung wird weiter unten eingegangen.

Zunächst wird das Finite-Domain-Constraint$C$ nach SMT-LIB übersetzt. Dies geschieht wie in Abschnitt 7.3 dargestellt. Dazu gehört auf der einen Seite das Deklarieren aller in dem Constraint vorkommenden Variablen und der Variablen aus der Liste $\mathcal{V}$

Algorithmus 1 Pseudocode, der den generellen Ablauf für die Kommunikation mit einem SMT-Solver darstellt

```
1:  procedure SolveSMT(V, C)
2:      S ← Smtlib(V, C)          ▷ FD-Constraint nach SMT-LIB übersetzen
3:      WriteCode(S)              ▷ starte Solver mit Programm S
4:      R ← ∅                     ▷ Ergebnismenge leer setzen
5:      A ← CheckSat              ▷ prüfe Erfüllbarkeit
6:      while A = sat do          ▷ solange es Lösungen gibt
7:          A ← GetValue(V)       ▷ frage Werte für die Variablen V ab
8:          R ← R ∪ {A}           ▷ füge Lösung zur Ergebnismenge hinzu
9:          S ← Exclude(A)        ▷ verbiete die neue Lösung
10:         WriteCode(S)          ▷ füge Verbot dem Stack des Solvers hinzu
11:         A ← CheckSat          ▷ prüfe Erfüllbarkeit erneut
12:     end while
13:     return FD(R)              ▷ gib Menge der Lösungen zurück
14: end procedure
```

inklusive der Assertions für die Domain und auf der anderen Seite das Übersetzen des gegebenen Constraints. Der so erzeugte Code wird dem Solver geschickt. Daraufhin wird durch Senden des `check-sat` Befehl die Erfüllbarkeit getestet. Ist die Ausgabe `sat`, dann existiert eine Interpretation und für die Variablen $\mathcal{V}$ können mittels `get-value` die Werte abgefragt werden. Diese Lösung wird der – zu Beginn leeren – Menge der Lösungen hinzugefügt. Darauf folgend wird eine Assertion erzeugt und dem Solver mitgeteilt, welche die gerade gefundene Lösung beziehungsweise Kombination der Variablenbelegungen ausschließt. Es wird dann erneut auf Erfüllbarkeit geprüft und der gesamte Prozess des Abfragens und Ausschließens einer Lösung wiederholt, bis nicht mehr `sat` als Antwort zurückgegeben wird. Abschließend wird die Menge der Lösungen zurückgegeben.

Wie bereits bei der Definition des Typs `SolverImpl` deutlich wurde, nutzt die Implementierung statt einer Menge eine Liste für die Lösungen. Eine Lösung ist dabei eine Liste von Integer-Werten in der Reihenfolge, in der die FD-Variablen der Solve-Funktion gegeben werden. Im Fall des schon betrachteten 4-Damen-Problems liefert der Ausdruck

```
solveFDAll [] qs (allSafe qs)
  where qs@[q1,q2,q3,q4] = take 4 (domain 1 4)
```

das Ergebnis `[[3,1,4,2],[2,4,1,3]]`, wobei die Werte in den Listen die gefundenen Werte für die Variablen `q1` bis `q4` in der in `qs` gegebenen Reihenfolge sind.

```
when (dbg > 0) $ do
  putStrLn $ "using solver " ++ (executable cfg)
  putStrLn $ "  called with options " ++ unwords (flags cfg)
  putStrLn $ "  with logic " ++ (smtLogic cfg)
when pers $ do
  putStrLn $ "saving program in file '" ++ fname ++ "'"
  writeFile fname $ showSMT smt
```

Listing 7.11: I/O-Aktionen, die bei bestimmten Option ausgeführt werden

Im Folgenden wird nun die Implementierung ausschnittweise vorgestellt. Eine vollständige Angabe der Funktionen findet sich in Anhang B.

An die Funktion `solveSMT`, die in der Konfiguration für einen SMT-Solver angegeben wird, reicht die `solveFD` Funktion die eigenen Parameter durch und übergibt zusätzlich die Solverkonfiguration:

```
solveSMT :: SolverImpl
solveSMT cfg options vars constr = do ...
```

Zunächst werden diese verarbeitet; es werden die Einstellungen aus `cfg` ausgelesen und die SMT-LIB Befehle zusammengesetzt sowie die Optionen gesetzt:

```
pre  = setOptions (smtOptions cfg) =>> setLogic (smtLogic cfg)
cmds = declare (allFDVars constr)  =>> assert ( constr )
smt  = pre =>> cmds
exec = unwords $ (executable cfg):(flags cfg)
(dbg, ns, (pers, fname), mini, maxi) = getSolverOptions options
```

Die Optionen werden wie in Listing 7.11 dargestellt verarbeitet: wenn ein bestimmtes Level für die Debug-Ausgabe gesetzt ist, werden Informationen bezüglich des Lösevorganges ausgegeben. Wird die Option `Persist` genutzt, wird der erzeugte SMT-LIB Code in eine Datei mit dem angegebenen Namen geschrieben. Im Weiteren wird nicht mehr auf das Debugging und die damit verbundenen Ausgaben eingegangen, für weitere Beispiele sei daher an Anhang B verwiesen.

Der Ausdruck `exec` setzt sich aus dem Namen der ausführbaren Datei des Solvers sowie den nötigen Argumenten zusammen. Über die Funktion `execCmd` aus dem Curry Modul `IOExts` wird dieser Befehl in einem neuen Prozess ausgeführt und gibt die Handles für die drei Standard-Datenströme stdin, stdout und stderr des Prozesses

zurück. Letzterer wird hier nicht benötigt, es genügt die Standardeingabe zum Senden von SMT-LIB Programmen und die Standardausgabe um die Ausgaben des Solvers zu erhalten:

```
(inH, outH, _) <- execCmd exec
```

Über die zwei Funktionen `hPutStr` und `hGetContents` kann ein String geschrieben beziehungsweise der Inhalt eines Datenstroms gelesen werden. Allerdings schließt `hGetContents` das Handle bevor der Inhalt zurückgegeben wird, so dass nicht mehr mit dem Datenstrom gearbeitet werden kann. Um die Antworten des Solvers einzeln lesen zu können, nutzen wir daher ein selbst implementiertes `hGetContentsUntil`, das neben einem Handle auch einen String als Parameter erhält. Es werden dann so lange einzelne Zeilen gelesen, bis der gegebene String gelesen wird. Eine solche Begrenzung kann in SMT-LIB leicht mit dem `echo` Befehl erzeugt werden. So sorgt

```
(check-sat)
(echo "===STOP-READING===")
```

dafür, dass erst die Ausgabe des `check-sat` Befehls erfolgt und danach eine Zeile, die die angegebene Zeichenkette enthält. Mit

```
hGetContentsUntil outH "===STOP-READING==="
```

wird dann lediglich die erste Zeile zurückgegeben (diese enthält im obigen Fall `sat`, `unsat` oder `unknown`) und das Handle kann für weitere Lesevorgänge genutzt werden.

Die Suche nach den Lösungen für das Constraint gestaltet sich nun wie zu Beginn dieses Abschnittes beschrieben und in Algorithmus 1 dargestellt. Mit der Funktion `getSolutions` werden die Lösungen gesucht:

```
solutions <- getSolutions inH outH vars ns dbg 1 mini maxi
```

Die eingegebenen Assertions werden auf Erfüllbarkeit geprüft und im Erfolgsfall wird die Belegung der Variablen in `vars` abgefragt. Dem Solver wird dann eine neue Assertion gegeben, durch die die gerade gefundene Lösung ausgeschlossen wird. Sei eine Lösung für Variablen `a`, `b` und `c` gegeben durch

```
((a 2) (b 5) (c 0)),
```

wird diese mittels `assert (a /=# fd 2 \/ b /=# fd 5 \/ c /=# fd 0)` verboten. Schrittweise wird nun eine Liste mit den Lösungen aufgebaut, die so in den einzelnen Aufrufen von `getSolutions` gefunden werden. Liefert der Solver `unsat` oder `unknown`, wird über `return []` die leere Liste zurückgegeben und `getSolutions` nicht erneut rekursiv aufgerufen. Der Wert `ns` aus den Optionen von oben wird genutzt, um die gewünschte Anzahl an Lösungen zu liefern: bei jedem rekursiven Aufruf wird er um Eins verringert und ebenfalls die leere Liste zurückgegeben, wenn wir bei Null angekommen sind. Somit werden höchstens `ns` Lösungen beim Solver abgefragt. Die Solveroption `All` setzt diesen Wert auf Minus-Eins, so dass Null durch Dekrementieren nicht erreicht werden kann.

Die Optimierung mittels `Minimize` und `Maximize` funktioniert auf die Weise wie das Suchen der Lösungen. Die Funktion `optimize` erhält unter anderem eine der Relationen `(<#)` und `(>#)` als Parameter `relop`. Wie eben schon beschrieben, wird nach einer Lösung für die zu optimierende Variable gesucht und der Wert mittels `relop` in einer Assertion als größer oder kleiner festgelegt. Ist die Formel dann unerfüllbar, steht der minimale oder maximale Wert für die Variable fest.

8. Anschluss von SAT-Solvern

Auch für SAT-Solver lässt sich eine Schnittstelle wie die aus dem vorherigen Kapitel realisieren. Der Aufbau dieses Kapitels entspricht dem des vorherigen und behandelt Folgendes: (1) Einführung eines Datentyps zur Repräsentation von Booleschen Formeln mit nummerierten Variablen, (2) Formatierung der Ausgabe und Parsen der Antwort des Solvers, (3) Transformation des FD-Modells in eine Boolesche Formel, (4) die Konfiguration eines SAT-Solvers und (5) die Kommunikation mit solchen Solvern.

Die Struktur der Module ist ebenfalls entsprechend gewählt, es wird lediglich `SMTLib` durch `Dimacs` ersetzt.

8.1. Darstellung von Booleschen Formeln

Den Hauptteil des in Abschnitt 4.1.2 beschriebenen DIMACS Formates macht die Darstellung der Formel aus. Für Boolesche Formeln nutzen wir den Datentyp `Boolean`, dessen Definition in Listing 8.1 gegeben ist. Variablen werden nummeriert. Die Formel

$$\varphi = (x_1 \vee \neg(\overline{x}_2 \wedge x_4)) \wedge (x_3 \wedge (x_1 \vee x_2)) \,,$$

die wir bereits in den Grundlagen zu SAT betrachtet haben, wird mit diesem Datentyp folgendermaßen ausgedrückt:

```
And [ Or  [ Var 1, Not (And [ Not (Var 2), Var 4])]
    , And [ Var 3, Or [ Var 1, Var 2 ] ]
    ]
```

```
data Boolean = Var Int
             | Not Boolean
             | And [Boolean]
             | Or  [Boolean]
```

Listing 8.1: Datentyp zur Repräsentation Boolescher Formeln

Da das DIMACS Format eine Formel in CNF verlangt, muss eine entsprechende Umwandlung mithilfe der Funktion `toCNF` erfolgen. Diese wandelt eine Formel zunächst in Negationsnormalform um und transformiert das Ergebnis anschließend nach CNF. Eine Formel ist in Negationsnormalform, wenn der Negationsoperator lediglich auf Variablen angewandt wird und ansonsten nur logisches Und und Oder als Operatoren erlaubt sind. Über die beiden De Morganschen Regeln werden Negationen nach innen, zu den Variablen geschoben. Umwandeln in CNF ist dann leicht möglich. Zusätzlich werden in diesem Schritt Tautologien herausgefiltert, also Klauseln, die eine Variable sowohl in negierter als auch nicht negierter Form enthalten. Da diese bei einer Oderverknüpfung immer wahr sind, müssen sie nicht betrachtet werden. Die CNF der obigen Formel (die ebenfalls schon in den Grundlagen gezeigt wurde), ist demnach Folgende:

```
And [ Or [Var 1, Var 2, Not (Var 4)]
    , Or [Var 3]
    , Or [Var 1,Var 2] ]
```

Um die Konstruktion wieder zu erleichtern, stehen Smart-Konstruktoren bereit: `(./\)` und `(.\/)` sorgen dafür, dass die Listenparameter für Und und Oder richtig aufgebaut werden. Mit `va` und `nv` werden eine Variable beziehungsweise eine negierte Variable bezeichnet. Die Formel in CNF lässt sich damit folgendermaßen ausdrücken:

```
(va 1 .\/ va 2 .\/ nv 3) ./\ (va 3) ./\ (va 1 .\/ va 2)
```

8.2. Pretty Printing und Parsing

Eine Formel mit einem Pretty Printer gemäß DIMACS zu formatieren, ist aufgrund der Struktur der Formeln in CNF unkompliziert möglich: ganz außen befindet sich ein `And` Konstruktor, der eine Liste von `Or` Konstruktoren enthält. Diese wiederum enthalten Listen mit Einträgen wie `Var 1` und `Not (Var 2)`. Die Funktion in Listing 8.2 zeigt, wie die Klauseln formatiert werden.

Zu dem DIMACS Format gehört neben der Menge der Klauseln auch die Präambel, die das Format, die Anzahl der Klauseln sowie die Anzahl der Variablen angibt. Als Erinnerung: die Präambel für die eben betrachtete Formel ist `p cnf 4 3`. Da die Variablen nummeriert werden – und diese Zahl mitgeführt wird, wie wir weiter unten sehen werden –, ergibt sich die erste Zahl automatisch. Wie viele Klauseln die Formel hat, lässt sich einfach über die Länge der Liste bestimmen, die das äußere `And` enthält.

Das Parsen der Lösung gestaltet sich unkompliziert: im Fall von Solvern, die sich bei der Ausgabe wie Lingeling verhalten, hat die erste Zeile die Form `s SATISFIABLE` oder

```
ppCNF :: Boolean -> Doc
ppCNF b = case b of
  Var  i -> int i
  Not  n -> text "-" <> ppCNF n
  And bs -> vsep $ map ppCNF bs
  Or  bs -> (hsep $ map ppCNF bs) <+> text "0"
```

Listing 8.2: Pretty Printing für eine Formel in CNF gemäß dem DIMACS Format, Klauseln stehen in einer eigenen Zeile

`s UNSATISFIABLE`. Z3 hingegen gibt lediglich `sat` oder `unsat` aus. Ein `s` mit einem folgenden Leerzeichen zu Beginn der Zeile kann ignoriert werden und anschließend kann auf die gerade genannten Schlüsselwörter geprüft werden. Die Zeile mit den Variablenbelegungen hat wieder abhängig vom Solver die Form `v` *Belegungen* `0` oder lediglich *Belegungen*. Das `v` und die `0` können also wieder ignoriert werden. In der Belegung wird ein i (ohne Minus davor) einfach zu `Var` i und $-i$ zu `Not (Var` i`)`. Zurückgegeben wird eine Liste Variablenbelegungen, wobei diese bei Unerfüllbarkeit leer ist. Die Lösung

```
s SATISFIABLE
v 1 2 3 4 0
```

aus Abschnitt 4.1.2 wird nach dem Parsen so dargestellt:

```
[Var 1, Var 2, Var 3, Var 4]
```

8.3. Überführung des FD-Modells

Das Überführen des FD-Modells in eine Formel, die nur Boolesche Variablen enthält, ist der aufwendigste Teil, der für den Anschluss von SAT-Solvern nötig ist. Die FD-Variablen müssen kodiert werden. Dafür bieten sich verschiedene Möglichkeiten, die sowohl Vor- als auch Nachteile mit sich bringen. Der Picat-SAT Compiler nutzt „sign-and-magnitude log encoding“ [16]. Eine Variable mit einem betragsmäßig maximalen Wert von n wird durch $\lceil \log_2(n) \rceil$ Boolesche Variablen dargestellt, mit einer zusätzlichen Variable für das Vorzeichen. Damit können Constraints wie `abs(x) =# y` sehr einfach umgesetzt werden, in diesem Fall kann das Vorzeichen von `y` einfach auf 0 (für eine positive Zahl) gesetzt werden. Allerdings muss die mit dieser Darstellung mögliche

negative Null ausgeschlossen werden. Wird das Zweierkomplement für die Darstellung gewählt, entfällt dies [12] selbstverständlich.

Damit eine FD-Variable immer auf die gleichen Booleschen Variablen abgebildet wird, muss diese Zuordnung in einem Zustand festgehalten werden. Dafür wird eine Zustandsmonade genutzt. Analog zur Umwandlung nach SMT-LIB werden `toDimacsC` und `toDimacsE` genutzt, um ein Constraint in eine Boolesche Formel zu transformieren. Für die Variablen steht darüber hinaus `toDimacsV` bereit. Über den bind-Operator der Zustandsmonade können diese auf folgende Weise kombiniert werden, um eine Liste `vars` von Variablen sowie ein Finite-Domain-Constraint `constr` umzuwandeln:

```
toDimacsV vars `bindS_` toDimacsC constr
```

Dabei speichert der Zustand, welche Zahl für die nächste Variable vergeben werden muss sowie eine Liste von Paaren. Durch diese Paare wird der Name der Variablen mit der Liste der Booleschen Variablen in Verbindung gesetzt. Wie viele Boolesche Variablen benötigt werden, um den Wertebereich einer FD-Variablen zu kodieren, lässt sich einfach anhand der oberen Grenze o und der unteren Grenze u bestimmen. Die Anzahl ergibt sich aus

$$2 + \lfloor \log_2 (\max \{|u - 1|, |o|\}) \rfloor \ .$$

Abhängig davon, welche der Grenzen den betragsmäßig größeren Wert hat , ergibt sich die Zahl der benötigten „Bits“, um diese als Zweierkomplement darzustellen.

In der bisherigen Implementierung ist lediglich das Gleichheitsconstraint `(=#)` umgesetzt. Betrachten wir den einfachen Fall `x =# y` mit zwei FD-Variablen `x` und `y`, die durch die Booleschen Variablen x_2 und x_1 beziehungsweise y_2 und y_1 dargestellt werden können, so repräsentiert die folgende Formel die Gleichheit der beiden Zahlen:

$$(x_2 \wedge y_2 \vee \overline{x}_2 \wedge \overline{y}_2) \wedge (x_1 \wedge y_1 \vee \overline{x}_1 \wedge \overline{y}_1)$$

Allgemein ist also die Gleichheit von zwei Zahlen, die durch Variablen x_i und y_i kodiert werden, gegeben durch

$$\bigwedge_i x_i \wedge y_i \vee \overline{x}_i \wedge \overline{y}_i \, .$$

Um die arithmetischen Operatoren und die restlichen relationalen Operatoren umzusetzen, werden Volladdierer und Multiplizierer nötig, welche im Rahmen dieser Arbeit nur ansatzweise umgesetzt wurden.

8.4. Solverkonfiguration

Das Modul `XFD.Solver` wurde schon in Abschnitt 7.4 behandelt und erlaubt einfaches Beschreiben weiterer Solver. Um Lingeling als Solver bereitzustellen, muss lediglich ein entsprechendes Modul angelegt werden, das die Konfiguration und die `solve` Funktionen bereitstellt, beispielsweise `XFD.Solvers.SAT.Lingeling`:

```
solverConfig = Config { executable  = "lingeling"
                      , flags       = ["-q"]
                      , solveWith   = solveDimacs
                      }
```

Abschließend muss die Implementierung der Solvefunktion `solveDimacs` angegeben werden, die von `solveFD` und den beiden Varianten davon aufgerufen wird.

8.5. Kommunikation mit Solvern

Die Kommunikation mit einem SAT-Solver gestaltet sich sehr ähnlich zu der mit SMT-Solvern, die in Abschnitt 7.5 beschrieben wurde: zunächst werden die Konfiguration des Solvers und die restlichen Parameter verarbeitet und das Constraint in das DIMACS Format transformiert. Der Solver wird ebenfalls mittels `execCmd` gestartet und über die Standard-Datenströme wird kommuniziert. Ein Unterschied ist, dass nicht inkrementell wie mit einem SMT-Solver gearbeitet werden kann. Es wird stattdessen für jede abzufragende Lösung das gesamte Modell auf die Standardeingabe geschrieben – jeweils erweitert um Constraints, die die vorherigen Lösungen ausschließen – und die Antwort gelesen. Anschließend muss durch erneutes Aufrufen von `execCmd` ein neuer Prozess gestartet werden.

9. Erweiterbarkeit

Die beiden vorangegangenen Kapitel haben gezeigt, wie verschiedene Solver in Curry eingebunden werden können. Allgemein kann festgehalten werden, dass folgende Schritte dafür nötig sind:

- ein Modell in Curry für die Zielsprache, die der Solver versteht;
- ein Pretty Printer, der dieses Modell in einem für den Solver verständlichen Format ausgibt, sowie ein Parser, der die Antworten des Solvers wieder in Curry bereitstellt;
- das Transformieren des FD-Modells in das Modell der Zielsprache;
- Angeben einer Konfiguration für den gewünschten Solver;
- die konkrete Implementierung einer entsprechenden Solvefunktion, die die Kommunikation regelt.

Für neue Zielsprachen bietet es sich daher an, der Struktur zu folgen, die bereits für SMT-LIB und DIMACS angelegt ist. Vorgestellt wurde diese zu Beginn von Kapitel 7.

Es ist mit sehr geringem Aufwand möglich, Solver für bereits bestehende Implementierungen von Zielsprachen hinzuzufügen. Für SMT-LIB wurden auf diese Weise neben Z3 auch die Solver CVC4[8], Yices[9] und MathSAT[10] angebunden. Sie sind wie Z3 unter `XFD.Solvers.SMT` zu finden. Es genügt dabei, eine passende Konfiguration (inklusive der Funktionen `solveFD`, `solveFDOne` und `solveFDAll`, analog zu Z3) anzugeben:

```
solverConfig = defaultConfig
                 { executable  = "yices-smt2"
                 , flags       = ["--incremental"]
                 , solveWith   = solveSMT
                 }
```

Auf diese Weise lässt sich die Implementierung beliebig erweitern.

[8] http://cvc4.cs.nyu.edu/
[9] http://yices.csl.sri.com/
[10] http://mathsat.fbk.eu/

Teil IV.

Schlussbetrachtung

10. Evaluation

In diesem Kapitel wird die Performanz der Implementierung und dem auf diese Weise integrierten SMT-Solver Z3 untersucht.

Durchgeführt wurden alle Programmläufe auf einem PC mit einer Intel Core i5-750 CPU (2,66 GHz) und 8 GB Arbeitsspeicher unter Ubuntu 14.04 „Trusty Tahr“. Die Zeitmessung für das Suchen mehrerer Lösungen von Hand ist nicht einfach realisierbar, wenn Z3 direkt über die Standardeingabe benutzt wird. Daher wird zunächst eine Messung durchgeführt, die für Curry die Funktion `solveFDOne` benutzt und für Z3 die mit der Option `Persist` erzeugte Datei als Eingabe verwendet (erweitert um die Befehle `check-sat` und `get-value`). Gemessen wird dabei die Zeit für einige Instanzen des N-Damen-Problems sowie des Programms „Magic Series“ (siehe Anhang C). Es wurde jeweils der Mittelwert von drei Messungen berechnet und gerundet, wobei diese in PAKCS (mit SICStus Prolog 4.3.2) und KiCS2 (mit GHC 7.6.3) mit der `time`-Option durchgeführt wurden. Für Z3 direkt die `time` Funktion der Shell genutzt wurde:

		Solver			
FD-Problem	n	PAKCS/Prolog	PAKCS/Z3	KiCS2/Z3	Z3 direkt
N-Damen	5	50	87	15	13
	10	60	350	57	41
	15	90	903	110	78
	20	6330	2453	230	23
	25	1890	3507	353	241
Magic Series	10	60	277	90	49
	20	70	2060	1250	1030
	30	90	20 353	17 580	16 532

Tabelle 10.1.: Performanz für verschiedene Instanzen des N-Damen-Problems mit n Damen und der Magic Series mit n Folgengliedern, alle Zeiten sind in Millisekunden angegeben

PAKCS/Prolog heißt hier, dass das Modul `CLP.FD` benutzt wurde, wohingegen bei PAKCS/Z3 und KiCS/Z3 jeweils das Modul `XFD.Solvers.SMT.Z3` zum Einsatz kam.

Es ist zu sehen, dass das Lösen mit PAKCS und dem Solver von SICStus Prolog am schnellsten ist, ausgenommen das 20-Damen-Problem. Dieser Wert ließ sich zuverlässig reproduzieren; warum es so lange dauert, diese Instanz zu lösen, konnte nicht festgestellt werden. Ansonsten ist das von KiCS2 erzeugte Programm immer recht nah an der Zeit des direkten Laufs mit Z3, während die PAKCS Variante teilweise erheblich viel länger benötigt. Dies war zu erwarten: KiCS2 braucht länger für das Kompilieren als PAKCS, erzeugt aber schnellere Programme. Bei der Magic Series ist der Prolog Solver Z3 klar überlegen.

Desweiteren wurde die Performanz verglichen, wenn beim N-Damen-Problem nach allen Lösungen gesucht wird. Auch hier schlägt sich PAKCS mit dem Prolog Solver am besten. Mit Z3 als Solver zeigt sich wieder die Überlegenheit von KiCS2 gegenüber PAKCS:

	Solver		
n	PAKCS/Prolog	PAKCS/Z3	KiCS2/Z3
4	50	107	15
5	50	270	33
6	60	210	37
7	70	1287	173
8	90	3370	497
9	230	18 790	2033
10	720	53 750	6097

Tabelle 10.2.: Suchen aller Lösungen des N-Damen-Problems mit $n \in \{4, \ldots, 10\}$ Damen, alle Zeiten sind in Millisekunden angegeben

Mit Z3 ergibt sich der zusätzliche Aufwand, das FD-Modell nach SMT-LIB zu überführen und über die Standard-Datenströme mit dem Solver zu kommunizieren. Wird mit PAKCS das Modul `CLP.FD` benutzt, entfallen diese Schritte.

11. Zusammenfassung

In dieser Masterarbeit wurde eine Bibliothek entwickelt, die das Programmieren mit Finite-Domain-Constraints unabhängig von der verwendeten Curry-Implementierung erlaubt und damit nicht auf das unterliegenden System angewiesen ist, in das die Curry-Programme kompiliert werden.

Dafür wurde zunächst eine Bibliothek zum Modellieren von Finite-Domain-Contraint-Satisfaction-Problemen implementiert. Diese Constraints sollen durch SMT- oder SAT-Solver wie Z3 und Lingeling gelöst werden. Dazu wurden für die Sprachen SMT-LIB beziehungsweise DIMACS Modelle in Curry angelegt, so dass die Finite-Domain-Modelle in äquivalente Modelle in der für den jeweiligen Solver verständlichen Eingabesprache übersetzt werden können. Für die Kommunikation mit den Solvern wurde eine Schnittstelle entwickelt, über die alle – auf Wunsch auch eine bestimmte Anzahl – Lösungen des Problems abgefragt und anschließend aufbereitet in Curry bereitgestellt werden.

Anhand von Zeitmessungen wurde gezeigt, dass die Bibliothek im Zusammenhang mit KiCS2 nur geringen Overhead hat verglichen mit dem direkten Benutzen von Z3. Abhängig von dem Problem ist die Implementierung unter Umständen sogar schneller als die bereits für PAKCS existierende.

Als mögliche Erweiterung in der Zukunft bietet es sich wahrscheinlich an, die Kommunikation mit den Solvern lazy zu gestalten. Die bisherige Implementierung sammelt zuerst die geforderte Anzahl an Lösungen ein (sofern so viele existieren) und liefert diese dann insgesamt als eine Liste von Ergebnissen. Ein lazy Ansatz hätte hier sicher insofern einen Geschwindigkeitsvorteil, als schneller mit den Lösungen weitergearbeitet werden könnte.

Darüber hinaus unterstützt SMT auch Theorien über reelle Zahlen, so dass die Bibliothek für CLPR-Anwendungen erweitert werden kann.

Auch könnten weitere Eingabesprachen für neue oder bereits angeschlossene Solver hinzugefügt werden.

A. Ausschnitt der Grammatik für SMT-LIB

Dieses Kapitel zeigt den umgesetzten Teil von SMT-LIB als EBNF. Die vollständige Grammatik ist in [3] in Anhang B zu finden. Die drei Nichtterminalsymbole ⟨*numeral*⟩, ⟨*symbol*⟩ und ⟨*string*⟩ tauchen unten auf der rechten Regelseite auf. Für diese sind hier keine Produktionen angegeben, da wir diese durch die Datentypen `Int` und `String` darstellen, die bereits in Curry eingebaut sind. Die eigentlichen Definitionen findet der interessierte Leser im SMT-LIB Standard.

S-expressions

⟨*spec_constant*⟩ ::= ⟨*numeral*⟩

Identifiers

⟨*identifier*⟩ ::= ⟨*symbol*⟩

Sorts

⟨*sort*⟩ ::= ⟨*identifier*⟩

Terms

⟨*qual_identifier*⟩ ::= ⟨*identifier*⟩

⟨*term*⟩ ::= ⟨*spec_constant*⟩
| ⟨*qual_identifier*⟩
| `(` ⟨*qual_identifier*⟩ ⟨*term*⟩$^{+}$ `)`

Command options

⟨*b_value*⟩ ::= `true` | `false`

⟨*option*⟩ ::= `:produce-models` ⟨*b_value*⟩

Commands

⟨*command*⟩ ::= `(` `assert` ⟨*term*⟩ `)`
| `(` `check-sat` `)`
| `(` `declare-const` ⟨*symbol*⟩ ⟨*sort*⟩ `)`
| `(` `echo` ⟨*string*⟩ `)`
| `(` `exit` `)`
| `(` `get-value` `(` ⟨*term*⟩$^+$ `)` `)`
| `(` `pop` ⟨*numeral*⟩ `)`
| `(` `push` ⟨*numeral*⟩ `)`
| `(` `set-logic` ⟨*symbol*⟩ `)`
| `(` `set-option` ⟨*option*⟩ `)`

Command responses

⟨*valuation_pair*⟩ ::= `(` ⟨*term*⟩ ⟨*term*⟩ `)`

⟨*check_sat_response*⟩ ::= `sat` | `unsat` | `unknown`

⟨*get_value_response*⟩ ::= `(` ⟨*valuation_pair*⟩$^+$ `)`

⟨*specific_success_response*⟩ ::= ⟨*check_sat_response*⟩ | ⟨*get_value_response*⟩

⟨*general_response*⟩ ::= ⟨*specific_success_response*⟩ | `(` `error` ⟨*string*⟩ `)`

B. Auszüge der Implementierung

Parser-Modul:

```
module XFD.Parser where

type Parser token a = [token] -> Either ParseError ([token], a)
type ParseError = String

--- Combine parsers with resulting representation of first one.
(<*) :: Parser token a -> Parser token b -> Parser token a
a <* b = const <$> a <*> b

--- Combine parsers with resulting representation of second one.
(*>) :: Parser token a -> Parser token b -> Parser token b
a *> b = const id <$> a <*> b

(<*>) :: Parser token (a -> b) -> Parser token a -> Parser token b
a <*> b = \ts -> case a ts of
            Left e         -> Left e
            Right (ts', f) -> case b ts' of
              Left e         -> Left e
              Right (ts2, x) -> Right (ts2, f x)

--- Combines two parsers in an alternative manner.
(<|>) :: Parser token a -> Parser token a -> Parser token a
a <|> b = \ts -> case a ts of
  Right (ts', x) -> Right (ts', x)
  Left e         -> case b ts of
    Left e' -> Left $ "parse error: " ++ e' ++ " | " ++ e
    Right (ts2, y) -> Right (ts2, y)

--- Apply unary function `f` to result of parser `p`
(<$>) :: (a -> b) -> Parser token a -> Parser token b
f <$> p = yield f <*> p

--- Apply binary function `f` to results of parsers `p1` and `p2`
liftP2 :: (a -> b -> r) -> Parser token a -> Parser token b -> Parser token r
liftP2 f p1 p2 = \ts -> case p1 ts of
```

```
                    Left e          -> Left e
                    Right (ts', x) -> case p2 ts' of
                      Left e -> Left e
                      Right (ts2, y) -> Right (ts2, f x y)

--- A parser with `x` as representation while consuming no tokens.
yield :: a -> Parser token a
yield x ts = Right (ts, x)

--- A parser recognizing a particular terminal symbol.
terminal :: token -> Parser token ()
terminal _ [] = eof []
terminal x (t:ts) = case x == t of
  True  -> Right (ts, ())
  False -> unexpected t ts

--- Returns parse error about unexpected end-of-file
eof :: Parser token a
eof _ = Left "unexpected end-of-file"

--- Returns parse error about unexpected token `t`
unexpected :: token -> Parser token a
unexpected t _ = Left $ "unexpected token " ++ show t

--- A star combinator for parsers. The returned parser
--- repeats zero or more times a parser p and
--- returns the representation of all parsers in a list.
star :: Parser token a -> Parser token [a]
star p = (\ts -> case p ts of
  Left e -> Left e
  Right (ts', x) -> (x:) <$> star p $ ts')
  <|> yield []

--- A some combinator for parsers. The returned parser
--- repeats the argument parser at least once.
some :: Parser token a -> Parser token [a]
some p = \ts -> case p ts of
  Left e -> Left e
  Right (ts', x) -> (x:) <$> star p $ ts'
```

SMT-LIB RDParser-Modul:

```
module XFD.SMTLib.RDParser where

import XFD.Parser
import XFD.SMTLib.Scanner
import XFD.SMTLib.Types
import XFD.SMTLib.Build


parse :: String -> Either ParseError CmdResponse
parse s = case parseCmdResult $ scan s of
  Left e        -> Left e
  Right ([], x) -> Right x
  Right _       -> Left "incomplete parse"

parseCmdResult :: Parser Token CmdResponse
parseCmdResult =  parseCmdCheckSatResponse
              <|> parseCmdGetValueResponse
              <|> parseCmdGenResponse
              <* terminal EOF

parseCmdCheckSatResponse :: Parser Token CmdResponse
parseCmdCheckSatResponse []     = eof []
parseCmdCheckSatResponse (t:ts) = case t of
  KW_sat     -> yield (CmdCheckSatResponse Sat) ts
  KW_unsat   -> yield (CmdCheckSatResponse Unsat) ts
  KW_unknown -> yield (CmdCheckSatResponse Unknown) ts
  _          -> unexpected t ts

parseCmdGetValueResponse :: Parser Token CmdResponse
parseCmdGetValueResponse []     = eof []
parseCmdGetValueResponse (t:ts) = case t of
  LParen -> CmdGetValueResponse <$> parseGetValueResponse <* terminal RParen $ ts
  _      -> unexpected t ts

parseGetValueResponse :: Parser Token GetValueResponse
parseGetValueResponse = some parseValuePair

parseValuePair :: Parser Token ValuationPair
parseValuePair = terminal LParen
              *> liftP2 ValuationPair parseTerm parseTerm
              <* terminal RParen

parseTerm :: Parser Token Term
```

```
parseTerm [] = eof []
parseTerm (t:ts) = case t of
  Number i -> yield (termSpecConstNum i) ts
  Id  name -> yield (termQIdent name) ts
  LParen   -> terminal OP_Minus *> (termQIdentT "-" <$> (some parseTerm))
              <* terminal RParen $ ts
  _        -> unexpected t ts

parseCmdGenResponse :: Parser Token CmdResponse
parseCmdGenResponse []     = eof []
parseCmdGenResponse (t:ts) = case t of
  LParen -> CmdGenResponse <$> parseErrorResponse <* terminal RParen $ ts
  _      -> unexpected t ts

parseErrorResponse :: Parser Token GenResponse
parseErrorResponse []     = eof []
parseErrorResponse (t:ts) = case t of
  KW_error -> Error <$> parseString $ ts
  _        -> unexpected t ts

parseString :: Parser Token String
parseString [] = eof []
parseString (t:ts) = case t of
  Str string -> yield string ts
  _          -> unexpected t ts
```

Ausschnitt aus SMT-LIB FromFD-Modul:

```
-- transforms a FD constraint into a SMT-Lib term
convertConstr :: FDConstr -> Term
convertConstr constr =
  case constr of
    FDTrue               -> termQIdent "true"
    FDFalse              -> termQIdent "false"
    FDRelCon   Neq e1 e2 -> convertConstr $ notC $ e1 =# e2 --SMT-Lib has no '/='
    FDRelCon   rel e1 e2 -> termQIdentT (relSymb rel) $ map (convertExpr) [e1, e2]
    FDAnd       e1 e2    -> termQIdentT "and" $ map (convertConstr) [e1, e2]
    FDOr        e1 e2    -> termQIdentT "or"  $ map (convertConstr) [e1, e2]
    FDNot           c    -> termQIdentT "not" $ [convertConstr c]
    FDAllDiff      xs    -> termQIdentT "distinct" $ map (convertExpr) xs
    FDSum       vs rel c -> termQIdentT (relSymb rel)
                                [ termQIdentT "+" (map convertExpr ((fd 0):vs))
                                , convertExpr c]
    FDScalar cs vs rel v -> convertConstr $ sum (scalarProd cs vs) rel v
```

```
    FDCount   v vs rel c -> termQIdentT (relSymb rel) [countSum v vs, convertExpr c]
  where
    relSymb :: FDRel -> String
    relSymb rel = case rel of
      Lt  -> "<"
      Leq -> "<="
      Gt  -> ">"
      Geq -> ">="
      _   -> "=" -- Equ; Neq cannot happen, is caught beforehand
    scalarProd :: [FDExpr] -> [FDExpr] -> [FDExpr]
    scalarProd cs vs = zipWith (*#) cs vs
    countSum :: FDExpr -> [FDExpr] -> Term
    countSum v vs =
      let one  = convertExpr $ fd 1
          zero = convertExpr $ fd 0
          comparisons  = map (\x -> convertConstr $ x =# v) vs
          reifications = map (\x -> termQIdentT "ite" [x, one, zero]) comparisons
      in  termQIdentT "+" reifications

-- transforms a FD expression into a SMT-Lib term
convertExpr :: FDExpr -> Term
convertExpr expr =
  case expr of
    FDVar     n _ _ _   -> termQIdent n
    FDInt     i         -> case i < 0 of
      True  -> termQIdentT "-" [termSpecConstNum (-i)]
      False -> termSpecConstNum i
    FDBinExp  op e1 e2  -> termQIdentT (opSymb op) $ map (convertExpr) [e1, e2]
    FDAbs     e         -> termQIdentT "abs" $ [convertExpr e]
  where
    opSymb op = case op of
      Plus  -> "+"
      Minus -> "-"
      Times -> "*"
```

Solver-Modul:

```
module XFD.Solver where

import Unsafe

import XFD.FD (FDConstr, FDExpr (FDVar))
import qualified XFD.SMTLib.Types as SMT (Option(..))
```

```
--------------------------------------------------------------------------------
-- types and type synonyms for solvers

-- arguments and return type for a solver
type SolverArgs a = [Option] -> [FDExpr] -> FDConstr -> a

-- an implementation additionaly gets the config and has IO return type
type SolverImpl   = SolverConfig -> SolverArgs (IO [[Int]])

-- configuration with fields for executable, command line flags, solver
-- implementation and settings for SMT solvers
data SolverConfig = Config
                  { executable  :: String
                  , flags       :: [String]
                  , solveWith   :: SolverImpl
                  , smtOptions  :: Maybe [SMT.Option]
                  , smtLogic    :: String
                  }

-- options to pass to the solver
data Option = Debug Int
              -- how many solutions?
            | All
            | First
            | FirstN Int
              -- keep the generated program
            | Persist String
              -- optimize a FD variable
            | Minimize FDExpr
            | Maximize FDExpr

-- options for the solver implemtation to work with; extracted from notations above
-- Int: debugging level
-- Int: number of solutions
-- (Bool, String): keep? generated program if yes: in file named filename
-- (Bool, FDExpr): minimize? given variable
-- (Bool, FDExpr): maximize? given varialbe
type ExtractedOptions = (Int, Int , (Bool, String), (Bool, FDExpr), (Bool, FDExpr))

--------------------------------------------------------------------------------
-- default configuration for SMT solvers
defaultConfig :: SolverConfig
defaultConfig = Config
              { flags      = []
```

```
              , smtOptions = Just [SMT.ProduceModels True]
              , smtLogic   = "QF_LIA"
              }

-- default options, can be modified by giving [Option] to solveFD functions
defaultOptions :: ExtractedOptions
defaultOptions =  ( 0           -- default debugging level of 0
                  , (-1)        -- number of solutions, (-1) means All
                  , (False, _)  -- (persist?, filename)
                  , (False, _)  -- (minimize?, var)
                  , (False, _)  -- (maximize?, var)
                  )

--- Takes a list of options and returns a tuple containing the extracted values
--- to work with.
getSolverOptions :: [Option] -> ExtractedOptions
getSolverOptions options = getOpts options defaultOptions
  where
    getOpts []     opts = opts
    getOpts (o:os) (d, n, (p, f), mi, ma) = case o of
      Debug   i   -> getOpts os (i, n   , (p, f), mi, ma)
      All         -> getOpts os (d, (-1), (p, f), mi, ma)
      First       -> getOpts os (d, 1   , (p, f), mi, ma)
      FirstN  i   -> getOpts os (d, i   , (p, f), mi, ma)
      Persist s   -> getOpts os (d, n   , (True, s), mi, ma)
      Minimize v  -> getOpts os (d, n, (p, f), (True, assertVar v), ma)
      Maximize v  -> getOpts os (d, n, (p, f), mi, (True, assertVar v))
    assertVar v = case v of
      FDVar _ _ _ _ -> v
      -- _             -> error "not an FDVar"

--------------------------------------------------------------------------------
-- helpers to build solveFD funtions

--- Get list of solutions from solver and combine them using the (?) operator to
--- imitate non-determinism as `CLP.FD.solveFD` implements it.
solveFDwith :: SolverConfig -> SolverArgs [Int]
solveFDwith cfg opt vars constr =
  let solveF    = solveWith cfg
      solutions = unsafePerformIO $ solveF cfg opt vars constr
  in foldr1 (?) solutions

--- Same as solveFDwith, but append option First to list of options so only one
--- solution is returned.
```

```
solveFDOnewith :: SolverConfig -> SolverArgs [Int]
solveFDOnewith cfg opt vars constr =
  let solveF    = solveWith cfg
      options   = opt ++ [First]
      solutions = unsafePerformIO $ solveF cfg options vars constr
  in head solutions

--- Same as solveFDwith, but append option All to list of options so all
--- solutions are returned; also returns list of solutions non-deterministically.
solveFDAllwith :: SolverConfig -> SolverArgs [[Int]]
solveFDAllwith cfg opt vars constr =
  let solveF    = solveWith cfg
      options   = opt ++ [All]
      solutions = unsafePerformIO $ solveF cfg options vars constr
  in solutions
```

SMTLib-Modul:

```
module XFD.SMTLib
  ( solveSMT
  ) where

import XFD.SMTLib.Pretty
import XFD.SMTLib.Types
import XFD.SMTLib.Build
import XFD.SMTLib.RDParser
import XFD.SMTLib.FromFD

import XFD.FD
import XFD.Solver

import IO
import IOExts
import List (last, init)

-- reads lines from an input handle until a line that equals `s` is read:
hGetContentsUntil :: Handle -> String -> IO String
hGetContentsUntil h s = do line <- hGetLine h
                           let l = unstring line
                           if l == s
                              then return []
                              else do ls <- hGetContentsUntil h s
                                      return (l ++ "\n" ++ ls)
  where
```

```
    -- some solvers put quotes around echo output...
    unstring :: String -> String
    unstring s' = case (s', head s', last s') of
      (_:tr@(_:_), '"', '"') -> init tr
      _                      -> s'

-- combination of `hPutStr` and `hFlush`:
hPutStrFlush :: Handle -> String -> IO ()
hPutStrFlush h s = hPutStr h s >> hFlush h

-- string to delimit answers to read one at a time
outputDelimiter :: String
outputDelimiter = "===STOP-READING==="

-- get contents until `outputDelimiter`
hGetDelimited :: Handle -> IO String
hGetDelimited h = hGetContentsUntil h outputDelimiter

-- show SMT-LIB with delimiter echo at the end
delimSMT :: SMT -> String
delimSMT cmds = showSMT (cmds =>> echo outputDelimiter)


--- Prints output of the external solver given a list of FD variables (second
--- argument) w.r.t. constraint (third argument).
--- The first argument contains the configuration for the solver to use.
solveSMT :: SolverImpl
solveSMT _ _ [] _ = return [[]]
solveSMT cfg options vars@(_:_) constr = do
  let pre  = setOptions (smtOptions cfg) =>> setLogic (smtLogic cfg)
      cmds = declare (allFDVars constr) =>> assert ( constr )
      smt  = pre =>> cmds
      exec = unwords $ (executable cfg):(flags cfg)
      (debug, numSol, (persist, filename), mini, maxi) = getSolverOptions options
  when (debug > 0) $ putStrLn $ "debugging level " ++ (show debug)
  when (debug > 0) $ do
    putStrLn $ "using solver " ++ (executable cfg)
    putStrLn $ "  called with options " ++ unwords (flags cfg)
    putStrLn $ "  with logic " ++ (smtLogic cfg)
  when persist $ do
    putStrLn $ "saving program in file '" ++ filename ++ "'"
    writeFile filename $ showSMT smt
  when (debug > 1) $ putStr "starting solver... "
  (inH, outH, _) <- execCmd exec
```

```
  when (debug > 1) $ putStrLn "done"
  when (debug > 1) $ putStr "writing program to stdin... "
  hPutStrFlush inH $ delimSMT $ smt
  when (debug > 1) $ putStrLn "done"
  checkForError outH debug
  when (debug > 1) $ putStrLn "searching solutions"
  solutions <- getSolutions inH outH vars numSol debug 1 mini maxi
  when (debug > 1) $ putStrLn $ "found " ++ (show (length solutions)) ++ " solutions"
  hPutStrFlush inH $ showSMT exit
  hClose inH
  hClose outH
  return solutions

-- check for any error message from the solver; uses `error` to get cancel I/O
-- computation; use directly after writing commands without check-sat to catch
-- problems with constraint like non linear arithmetic with solvers that only
-- support linear
checkForError :: Handle -> Int -> IO ()
checkForError outH debug = do
  when (debug > 1) $ putStr "checking for errors raised by solver... "
  response <- hGetDelimited outH
  if (response /= "")
    then do
      case parse response of
        Left  e -> error e
        Right p -> case p of
          CmdGenResponse (Error msg) -> do
            when (debug > 1) $ putStrLn ("got error message")
            error msg
          _                          -> error "unexpected response"
    else do
      when (debug > 1) $ putStrLn "all clear"

-- print solution number and values
putSolution :: Int -> [Int] -> IO ()
putSolution n s = putStrLn $ "Solution #" ++ (show n) ++ ": " ++ (show s)

-- check satisfiability and get solution iff answer is sat; exclude said
-- solution and look for next one if `n` is not 0
-- returns list of solutions
getSolutions :: Handle -> Handle -> [FDExpr] -> Int -> Int
             -> Int -> (Bool, FDExpr) -> (Bool, FDExpr) -> IO [[Int]]
getSolutions inH outH fds n debug nSol (mini, miv) (maxi, mav)
 | n == 0 = return []
```

```
  | otherwise = do
      hPutStrFlush inH $ delimSMT checkSat
      when (debug > 1) $ putStr "waiting for checksat response... "
      response <- hGetDelimited outH
      when (debug > 1) $ putStrLn "received"
      case parse response of
        Left  e -> putStr e >> return []
        Right p -> case p of
          CmdGenResponse (Error msg) -> do
            putStrLn $ "error from solver: " ++ msg
            return []
          CmdCheckSatResponse Sat -> do
            when mini $ optimize inH outH debug miv (<#)
            when maxi $ optimize inH outH debug mav (>#)
            hPutStrFlush inH $ delimSMT (getValue fds)
            when (debug > 1) $ putStr "waiting for value response... "
            values <- hGetDelimited outH
            when (debug > 1) $ putStrLn "received"
            let vs  = case parse values of
                        Left e -> error e
                        Right vr -> convertValueResponse vr
                sol = extractValues vs
            when (debug > 1) $ putStr "writing excluded solution... "
            hPutStrFlush inH $ showSMT (excludeSolution vs)
            when (debug > 1) $ putStrLn "done"
            when (debug > 0) $ putSolution nSol sol
            liftIO (sol:) $ getSolutions inH outH fds (n-1) debug (nSol+1) (False, _) (False, _)
          _ -> return []

-- optimize variable according to relation; `(<#)` for minimiing value of
-- variable `ov`, `(>#)` for maximizing it
optimize :: Handle -> Handle -> Int -> FDExpr -> (FDExpr -> FDExpr -> FDConstr) -> IO ()
optimize inH outH debug ov relop = do
    when (debug > 1) $ putStrLn $ "optimizing value for variable "
                                  ++ getFDVarName optv
    hPutStrFlush inH $ delimSMT (getValue [optv])
    when (debug > 1) $ putStr "waiting for value response... "
    values <- hGetDelimited outH
    when (debug > 1) $ putStrLn "received"
    let vs = case parse values of
               Left e -> error e
               Right vr -> convertValueResponse vr
        sol = head $ extractValues vs
    optval <- optimize' sol
```

```
    when (debug > 1) $ putStrLn $ "found optimal value to be " ++ show optval
    hPutStrFlush inH $ showSMT (assert (optv =# fd optval))
    hPutStrFlush inH $ delimSMT checkSat
    when (debug > 1) $ putStr "waiting for checksat response... "
    response <- hGetDelimited outH
    when (debug > 1) $ putStrLn "received"
    case parse response of
      Left  e -> error e
      Right p -> case p of
        CmdGenResponse (Error msg) -> error $ "error from solver: " ++ msg
        CmdCheckSatResponse Sat -> return ()
        _ -> error "error occured"
  where
    optv = case ov of
      FDVar _ _ _ _ -> ov
      _             -> error "optimize: not an FDVar"
    optimize' :: Int -> IO Int
    optimize' val = do
      hPutStrFlush inH $ showSMT $ push 1
      hPutStrFlush inH $ showSMT (assert (optv `relop` fd val))
      hPutStrFlush inH $ delimSMT checkSat
      when (debug > 1) $ putStr "waiting for checksat response... "
      response <- hGetDelimited outH
      when (debug > 1) $ putStrLn "received"
      case parse response of
        Left  e -> error e
        Right p -> case p of
          CmdGenResponse (Error msg) -> error $ "error from solver: " ++ msg
          CmdCheckSatResponse Sat -> do
            hPutStrFlush inH $ delimSMT (getValue [optv])
            when (debug > 1) $ putStr "waiting for value response... "
            values <- hGetDelimited outH
            when (debug > 1) $ putStrLn "received"
            let vs  = case parse values of
                        Left e -> error e
                        Right vr -> convertValueResponse vr
                sol = head $ extractValues vs
            hPutStrFlush inH $ showSMT $ pop 1
            optimize' sol
          _ -> do
            hPutStrFlush inH $ showSMT $ pop 1
            return val

-- convert value response to list of FD variables and integers
```

```
convertValueResponse :: CmdResponse -> [FDExpr]
convertValueResponse rsp = case rsp of
    CmdGetValueResponse vr -> map extractVP vr
    _ -> []
  where
    extractVP (ValuationPair n v) = toFD n v --(name n, value v False)
    toFD n v = case n of
      TermQualIdentifier (QIdentifier (ISymbol s)) -> FDVar s _ _ (value v False)
      TermSpecConstant (SpecConstantNumeral _)     -> fd (value v False)
      TermQualIdentifierT (QIdentifier (ISymbol "-")) [i] -> fd (value i True)
      _ -> error "not a variable name or integer"
    value t neg = case t of
      TermSpecConstant (SpecConstantNumeral v) -> if neg then (-v) else v
      TermQualIdentifierT (QIdentifier (ISymbol "-")) [v] -> value v True
      _ -> error "not a value"

-- get list of values of a list of FD expressions
extractValues :: [FDExpr] -> [Int]
extractValues = map getFDVal

-- produce assertion to exclude values for a list of FD variables
excludeSolution :: [FDExpr] -> SMT
excludeSolution vs = assert (orC $ map exclude (filterVars vs))
  where
    exclude x = case x of
      FDVar _ _ _ v -> x /=# fd v
      _ -> error "solutin of non FDVars cannot be excluded"
```

C. Beispiel FD-Probleme

Magic Series:

```
-----------------------------------------------------------------------------
--- Computing magic series.
--- A series [a_0,a_1,....,a_(n-1)] is called magic iff there are
--- a_i occurrences of i in this series, for all i=1,...,n-1
---
--- Adapted from an example of the TOY(FD) distribution.
-----------------------------------------------------------------------------

module XFD.Examples.MagicSeries (
    magic, magicWo, Option(..)
  ) where

-- import XFD.Solvers.SMT.Z3
import CLP.FD

magic :: Int -> [Int]
magic n = magicWo n []

magicWo :: Int -> [Option] -> [Int]
magicWo n opt =
  let vs = take n $ domain 0 (n-1)
      is = map fd (take n [0..])
  in solveFD opt vs $
       constrain vs vs is /\
       sum vs Equ (fd n) /\
       scalarProduct is vs Equ (fd n)


constrain :: [FDExpr] -> [FDExpr] -> [FDExpr] -> FDConstr
constrain []     _  _      = true
constrain (x:xs) vs (i:is) = count i vs Equ x /\ constrain xs vs is
```

Send More Money:

```
module XFD.Examples.SendMoreMoney where

import XFD.Solvers.SMT.Z3

sendMoreMoney :: [Int]
sendMoreMoney =
 let xs@[s,e,n,d,m,o,r,y] = take 8 (domain 0 9)

     constraints =
        s ># fd 0 /\
        m ># fd 0 /\
        allDifferent xs /\
           fd 1000  *# s +# fd 100  *# e +# fd 10  *# n +# d
        +# fd 1000  *# m +# fd 100  *# o +# fd 10  *# r +# e
        =# fd 10000 *# m +# fd 1000 *# o +# fd 100 *# n +# fd 10 *# e +# y
 in solveFD [] xs constraints
```

Send More Money in SMT-LIB, generiert durch die Bibliothek:

```
(set-option :produce-models true)
(set-logic QF_NIA)
(declare-fun fdv_0_9_1 () Int)
(declare-fun fdv_0_9_5 () Int)
(declare-fun fdv_0_9_2 () Int)
(declare-fun fdv_0_9_3 () Int)
(declare-fun fdv_0_9_4 () Int)
(declare-fun fdv_0_9_6 () Int)
(declare-fun fdv_0_9_7 () Int)
(declare-fun fdv_0_9_8 () Int)
(assert (and (<= 0 fdv_0_9_1) (<= fdv_0_9_1 9)))
(assert (and (<= 0 fdv_0_9_5) (<= fdv_0_9_5 9)))
(assert (and (<= 0 fdv_0_9_2) (<= fdv_0_9_2 9)))
(assert (and (<= 0 fdv_0_9_3) (<= fdv_0_9_3 9)))
(assert (and (<= 0 fdv_0_9_4) (<= fdv_0_9_4 9)))
(assert (and (<= 0 fdv_0_9_6) (<= fdv_0_9_6 9)))
(assert (and (<= 0 fdv_0_9_7) (<= fdv_0_9_7 9)))
(assert (and (<= 0 fdv_0_9_8) (<= fdv_0_9_8 9)))
(assert (and (> fdv_0_9_1 0) (and (> fdv_0_9_5 0) (and (distinct fdv_0_9_1 fdv_0_9_2
 ↪  fdv_0_9_3 fdv_0_9_4 fdv_0_9_5 fdv_0_9_6 fdv_0_9_7 fdv_0_9_8) (= (+ (+ (+ (+ (+ (+
 ↪  (+ (* 1000 fdv_0_9_1) (* 100 fdv_0_9_2)) (* 10 fdv_0_9_3)) fdv_0_9_4) (* 1000
 ↪  fdv_0_9_5)) (* 100 fdv_0_9_6)) (* 10 fdv_0_9_7)) fdv_0_9_2) (+ (+ (+ (+ (* 10000
 ↪  fdv_0_9_5) (* 1000 fdv_0_9_6)) (* 100 fdv_0_9_3)) (* 10 fdv_0_9_2))
 ↪  fdv_0_9_8))))))
```

Literatur

[1] S. Antoy und M. Hanus. *Curry: A Tutorial Introduction*. Available at http://www.curry-language.org. 2014.

[2] C. W. Barrett u. a. „Satisfiability Modulo Theories.“ In: *Handbook of satisfiability* 185 (2009), S. 825–885.

[3] C. Barrett, P. Fontaine und C. Tinelli. *The SMT-LIB Standard: Version 2.5*. Techn. Ber. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2015.

[4] A. Biere. „Lingeling and friends entering the SAT challenge 2012“. In: *Proceedings of SAT Challenge* (2012), S. 33–34.

[5] A. Biere. „Lingeling, plingeling and treengeling entering the sat competition 2013“. In: *Proceedings of SAT Competition* (2013), S. 51–52.

[6] A. Biere. „Yet another local search solver and lingeling and friends entering the SAT competition 2014“. In: *SAT Competition* 2014 (2014), S. 39–40.

[7] R. Caballero und F. J. López-Fraguas. „A functional-logic perspective of parsing“. In: *Fuji International Symposium on Functional and Logic Programming*. Bd. 1722. Springer. 1999, S. 85–99.

[8] D. Challenge. „Satisfiability: Suggested Format“. In: *DIMACS Challenge. DIMACS* (1993).

[9] L. Damas und R. Milner. „Principal type-schemes for functional programs“. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1982, S. 207–212.

[10] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at http://www.curry-language.org. 2016.

[11] M. Hanus. „Adding Plural Arguments to Curry Programs“. In: *Technical Communications of the 29th International Conference on Logic Programming (ICLP 2013)*. Bd. 13. 4-5. Theory und Practice of Logic Programming (Online Supplement), 2013.

[12] J. Huang. „Universal Booleanization of constraint models“. In: *Principles and Practice of Constraint Programming*. Springer. 2008, S. 144–158.

[13] L. de Moura und N. Bjørner. „Z3: An efficient SMT solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, S. 337–340.

[14] N.-F. Zhou. „Combinatorial Search With Picat“. In: *arXiv preprint arXiv:1405.2538* (2014). URL: http://arxiv.org/abs/1405.2538.

[15] N.-F. Zhou und J. Fruhman. *A User's Guide to Picat*. Available at http://picat-lang.org. 2016.

[16] N.-F. Zhou und H. Kjellerstrand. „The Picat-SAT Compiler“. In: *Proceedings of PADL'2016, Practical Aspects of Declarative Languages*. 2016.

Index