Tiny Constraint Modelling Language

Marco Kuhlmann

Überblick

Tiny Constraint Modelling Language

- Modellierungssprache für Constraint-Probleme (CSPs)
  - Alternative zum direkten Programmieren in einem Constraint-System
  - ähnliche Ansätze: OPL, AMPL
- Benutzerschnittstelle zu verschiedenen Constraint-Systemen
  - Übersetzung von TCML in Sprache des Zielsystems
  - Beschränkung auf wesentliche Konzepte

Motivation (1)

Modellierungssprachen bieten eine Reihe von Vorzügen:

- für den unerfahrenen Programmierer
  - einfache Sprachen
  - schnelle Erfolgserlebnisse
- für den erfahrenen Programmierer
  - Vergleich verschiedener Constraint-Systeme
  - Benchmarking
- allgemein
  - implementierungsunabhängig
  - unterstützt deklaratives Programmieren

Motivation (2)

Modellierungssprachen sind ein idealer Ansatzpunkt für die automatische Analyse:

- Statische Analyse
  - Type-Checking, auch wenn Zielsprache ungetypt ist
  - Idealfall: Modellierung korrekt $\Rightarrow$ Zielprogramm korrekt
- Constraint-Analyse
  - Erkennen nachteiliger Problemformulierungen
  - Auswahl besserer Constraints und Propagierer

Vorgehen

- Entwurf
  - wichtige Konzepte identifizieren
  - Sprachdesign
- Implementierung
  - statische Analyse
  - Constraint-Analyse
  - Übersetzung für Mozart Oz und ILOG Solver
- Evaluierung
  - ‚Wie funktioniert so etwas?‘
  - Lohnt es sich, diese Ideen weiter zu verfolgen?

Beispiel: Send More Money

```
int[] money() {
   int s,e,n,d,m,o,r,y in {0..9};
   int[8] solution = [s,e,n,d,m,o,r,y];
   allDifferent(solution);
   s :!= 0;
   m :!= 0;
                1000*s + 100*e + 10*n + d
              + 1000*m + 100*o + 10*r + e
   := 10000*m + 1000*o + 100*n + 10*e + y;
   distribute(ff,solution);
   return solution;
}

void main() {
   search(all,money);
}
```

Send More Money: Übersetzung der Constraints (1)

TCML

```
allDifferent(solution);
s :!= 0;
m :!= 0;
             1000*s + 100*e + 10*n + d
           + 1000*m + 100*o + 10*r + e
:= 10000*m + 1000*o + 100*n + 10*e + y;
distribute(ff,solution);
```

Mozart Oz

```
{TCML.allDifferent Osolution}
Os \=: 0
Om \=: 0
{FD.sumC
  [1 91 ~9000 ~90 ~900 10 1000 ~1] [Od Oe Om On Oo Or Os Oy]
  '=:' 0}
{TCML.distribute ff Osolution}
```

Send More Money: Übersetzung der Constraints (2)

TCML

```
allDifferent(solution);
s :!= 0;
m :!= 0;
             1000*s + 100*e + 10*n + d
           + 1000*m + 100*o + 10*r + e
:= 10000*m + 1000*o + 100*n + 10*e + y;
distribute(ff,solution);
```

ILOG Solver

```
model.add(IloAllDiff(env,Isolution));
model.add(Is!=0);
model.add(Im!=0);
model.add(
  1*Id+91*Ie+(-9000)*Im+(-90)*In+(-900)*Io+10*Ir+1000*Is+(-1)*Iy
  == 0);
IloGoal goal = IloGenerate(env,Isolution,IlcChooseMinSizeInt);
```

Golomb-Lineale

```
int[][] golomb5() {
   int n = 5;
   int nn = n*n;
   int[n] k in {0..nn};
   int[][] d in {0..nn};
   k[0] := 0;
   foreach i in {0..n-2} {
      k[i+1] :> k[i];
   }
   foreach i in {0..n-2} {
      foreach j in {i+1..n-1} {
         d[i][j] := k[j]-k[i];
      }
   }
   allDifferent(d);
   distribute(naive,d);
   return d;
}
```

Entwurf

Sprachdesign: Überblick

- Datentypen
  - `int`
  - `array(t)` (statische und dynamische Arrays)
- Ausdrücke

  `* + - && || == != < > <= >= sum prod min max`
- Anweisungen
  - Zuweisung, Prozeduraufruf, `return`, `skip`
  - Konditional
  - Iteratoren
  - Constraints

Sprachdesign: Dynamische Arrays

- dynamisch = nur Dimension bekannt, nicht Größe

  ```
  int[][] foo in {0..nn};
  ```

- wichtiges Ausdrucksmittel
  - Modellierung von Diagonalmatrizen
  - Beispiel Golomb:

    ```
    foreach i in {0..n-2} {
       foreach j in {i+1..n-1} {
          d[i][j] := k[j]-k[i];
       }
    }
    ```

- Konzept fehlt in Mozart Oz und ILOG Solver

Sprachdesign: Semantik dynamischer Arrays

- Initialisierung
  - Dimension
  - Initialisierungstag: `int`, `range`
- Einfügen
  - Testen, ob Array-Index schon besetzt
  - ggf. einfügen und ‚nachinitialisieren‘
  - sonst überschreiben
- Zugriff
  - Testen, ob Array-Index schon besetzt
  - ggf. neu erzeugen und initialisieren
  - sonst entsprechenden Eintrag zurückgeben

Sprachdesign: Such-Skripte

Unterschiedliche Systeme bieten unterschiedliche Mittel zur Modellierung von CSPs:

- in Mozart Oz: Skripte und enkapsulierte Suche
- in ILOG: Objekte und Methoden
- in TCML hybride Lösung in Anlehnung an Java:

```
int[] money() {
   ...
   return solution;
}

void main() {
   search(all,money);
}
```

Skripte sind also voll emanzipiert.

Sprachdesign: Built-ins

- Built-ins für Constraints und Suche
  - `allDifferent`
  - `distribute`
  - `search`
- Linearisierung
  - Beispiel: `allDifferent(foo)` statt `allDifferent(foo[1],foo[2],foo[3])`
  - vermeidet die Notwendigkeit von Überladung
  - Einführung ‚linearisierbarer Typen‘:

```
procspec(id:allDifferent
         type:proc([lin(array(int))] void))
```

Sprachdesign: Iteratoren

- Iteratoren wichtig beim Absetzen von Constraints:

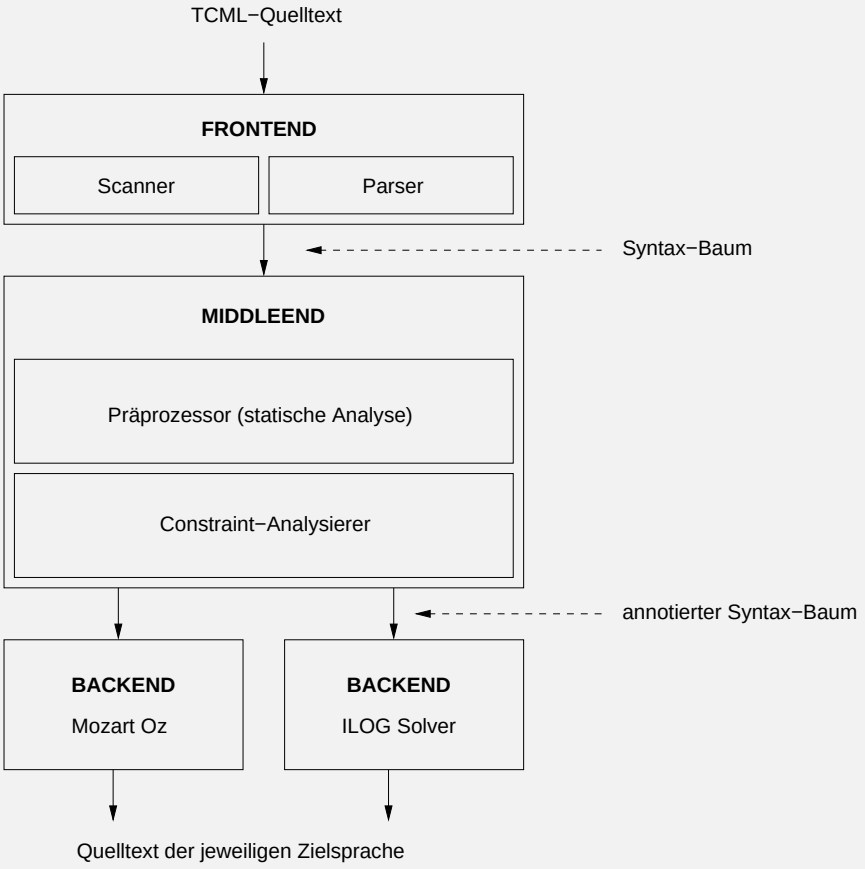
  ```
  foreach @i in code { code[i] :!= i+1; }
  ```

- in TCML unterstützte Iteratoren:
  - `for` – wie in C und Java
  - `foreach x in foo` – Iteration über Elemente
  - `foreach @i in foo` – Iteration über Indizes
  - `foreach x@i in foo` – Iteration über Elemente und Indizes

  (letzterer besonders interessant für Mengen)

Implementierung

Systemarchitektur



Implementierung: Constraint-Normalisierung

Transformation arithmetischer Constraints in Normalform

- allgemeine Normalform: $\sum_{i=1}^{n} a_i \prod_{j=1}^{m_i} x_{ij}^{p_{ij}} \operatorname{op} d$
- Motivation
  - Klassifizierung notwendig für Constraint-Analyse
  - Erleichterung bei der Code-Generierung
- Ein Problem
  - Im Idealfall: Variablen paarweise verschieden machen
  - Wann aber sind zwei Variablen gleich?

    `foo[42]` `foo[2*21]` `foo[bar]` `boo`

Implementierung: Constraint-Klassifizierung

- Beispiele für Constraint-Klassen
  - lineare Constraints: $\sum_{i=1}^{n} d_i x_i \,\mathrm{op}\, d$
  - lineare Constraints mit unitären Koeffizienten: $\sum_{i=1}^{n} \pm x_i \,\mathrm{op}\, d$
  - einfache Multiplikationen: $x_1 = x_2 \cdot x_3$
  - Quadrate: $x_1 = x_2^2$
- wichtig für Constraint-Analysierer und Backends
- ggf. weitere Transformationen in den Backends

Implementierung: Constraint-Analysierer (1)

- Idee: wenn möglich Domain-Propagierer durch Bounds-Propagierer ersetzen
  - Theoretische Grundlage: Schulte & Stuckey 2001
  - Motivation: Effizienzsteigerung
  - Methode: abstrakte Interpretation

Implementierung: Constraint-Analysierer (2)

- Probleme
  - dynamische Arrays
    - ⋆ Welche Variablen sind derzeit im Array enthalten?
  - Prozeduraufrufe
    - ⋆ aufgerufene Prozeduren setzen möglicherweise Constraints ab
    - ⋆ Lösung: multivariante Analyse
  - Schleifen
    - ⋆ Zählervariablen ändern ihren Wert zur Laufzeit
    - ⋆ Lösung: multivariante Analyse
- Status: Constraint-Analysierer fertig, aber noch nicht in Gesamtsystem integriert

Implementierung: Support-Bibliotheken

- Hilfsfunktionen, um die Code-Generierung zu erleichtern
- Implementierung benötigter Datenstrukturen
  - multidimensionale Arrays (für Oz)
  - multidimensionale Dictionaries
- Implementierung des TCML-Sprachdesigns
  - Beispiel: Wrapper für `allDifferent`

```
proc {AllDifferent X}
   if {MultiArray.is X} then
      {FD.distinct {MultiArray.toList X}}
   elseif {MultiDictionary.is X} then
      {FD.distinct {MultiDictionary.toList X}}
   else
      {FD.distinct X}
   end
end
```

Evaluierung

Evaluierung: Sprachdesign (1)

- Ausdrucksmächtigkeit
  - getestete Beispiele ließen sich problemlos modellieren:
    - Send More Money
    - Professor Smart's Safe
    - Grocery Store
    - Alphabet
    - Golomb
  - mehr und komplexere Beispiele notwendig

Evaluierung: Sprachdesign (2)

- Übersetzbarkeit
  - Übersetzung in die beiden Zielsprachen funktioniert
  - Aufwand bei der Übersetzung abhängig vom Zielsystem:
    - ⋆ in Mozart Oz: aufwändige Support-Bibliotheken
    - ⋆ in ILOG Solver: aufwändige Code-Erzeugung
  - Zusammenhang mit Typsystemen der Zielsprachen
  - Beispiel: Einfügen in dynamische Arrays
    - ⋆ in Mozart Oz: Funktion aus der Support-Bibliothek
    - ⋆ in ILOG Solver: Generierung wohltypisierter Funktionen

Evaluierung: Sprachdesign (3)

- CSP-Modellierung mit Skripten
  - bewährt und für Oz-Benutzer vertraut
  - erfordert voll emanzipierte Prozeduren
  - intuitiv auch für Anfänger?
  - Gibt es bessere Alternativen?

Evaluierung: Sprachdesign (4)

– Lösung in ILOG Solver

```
IloEnv env;
IloModel model(env);
...
IloGoal goal = IloGenerate(env,Isolution,IlcChooseMinSizeInt);
IloSolver solver(model);
IlcSearch search = solver.newSearch(goal);
while (search.next()) {
  solver.out() << solver.getIntVarArray(Isolution) << endl;
}
search.end();
env.end();
```

Ausblick

Erprobung der Sprache

- Ausdrucksmächtigkeit erproben
  - mehr Beispiele
  - komplexere Beispiele
- Implementierung testen
  - Ist das System stabil?
  - Ist die gewählte Modularisierung sinnvoll?
  - Ist das System leicht zu erweitern?
- Nützlichkeit der Sprache

Erweiterung der Sprache

- weitere Datentypen, zum Beispiel Mengen
- Erweiterung des Constraint-Analysierers
  - multivariante Analyse für Prozeduraufrufe und Schleifen
  - Analyse von dynamischen Arrays
- weitere Backends, zum Beispiel SICStus Prolog, Alice
- Entwicklungsumgebung
  - interaktive Version wünschenswert
  - Feedback-Komponente für den Constraint-Analysierer
    - ⋆ auf Defizite in der Modellierung hinweisen
    - ⋆ interaktives Lernen von Konzepten der Constraint-Programmierung

Verfügbarkeit des Systems

- Quellen verfügbar im CVS:

  `/services/ps/CVS/tcml`

- umfangreiche Dokumentation (ca. 100 Seiten)
- Verwendung von Literate Programming
  - Einsatz von NOWEB
  - gute Wartbarkeit des Systems