The Mozart Programming System

Tobias Ruf, Jan Tammen

Wissensbasierte Systeme SS 07,
Fakultät Informatik,
HTWG Konstanz

24. April 2007

Einleitung

Mozart - Einleitung

- Mozart implementiert die multiparadigmische Programmiersprache Oz
- Forschungsprojekt seit ca. 1995 (B, D, S)
- Oz ist plattformunabhängig (Byte-Code ähnlich wie Java)
- Open-Source-Lizenz

Mozart - Features

- Unterstützung mehrerer Programmierparadigmen
- Nebenläufigkeit (Threads, Synchronisation)
- Transparente Verteilung
- Dynamische Typisierung

Oz - Programmiermodell (OPM)

- Berechnungen laufen in sog. „Spaces“ (Berechnungsraum) ab
- *Aktoren* kommunizieren darin über einen gemeinsamen *Speicher*, können Information ablegen und abrufen
- Grundlage von Oz: Concurrent Constraint Programming, Erweiterungen durch *syntactic sugar*

Oz - Datentypen

- Basisdatentypen: `Number: Int, Float, Char`
- Zusammengesetzte Typen: `Record: Tuple, Literal, Bool`
- `Chunk`: erlaubt Erstellung abstrakter Datentypen, vordefiniert z. B. `Array, Dictionary, Class`
- `Thread, Space, ByteString, ...`

Constraint Programming

- Constraint: logische Formel, welche den möglichen Wertebereich von Variablen einschränkt (engl. to constrain)
- Elementarer Constraint: z. B.
  $X = Y \quad X = 23 \quad X = \mathrm{pair}(Y\ Z)$
- *finite domain* Constraint: $X :: 1\#42$, Beschränkung von X auf ganze Zahlen zwischen 1 und 42

Constraint Programming: finite domain problems

- Lösung kombinatorischer Probleme, z. B. „N Damen“-, „Graphfärbe“-Problem mithilfe von *Constraint Propagation*
- Hier: Speicher $\equiv$ *Constraint Store*, Aktoren $\equiv$ *Propagators*
- Constraint Store speichert Konjunktion aus elementaren Constraints, z. B. $X \in 0\#5 \wedge Y = 8 \wedge Z \in 13\#23$
- Propagators führen komplexte Constraints ein, z. B. $X < Y$ oder $X^2 + Y^2 = Z^2$

Constraint Programming: Beispiel Propagation

- Constraint Store enthält: $X \in 0\#9 \wedge Y \in 0\#9$
- Propagator 1 (P_1): $X + Y = 9$, Propagator 2 (P_2): $2 \cdot X + 4 \cdot Y = 24$

1. P_2 schränkt ein: $X \in 0\#8 \quad Y \in 2\#6$
2. P_1 schränkt ein: $X \in 3\#7 \quad Y \in 2\#6$
3. ...
4. P_2 stellt fest: $X = 6 \quad Y = 3$

Constraint Programming: Beispiel Propagation

- Constraint Store enthält: $X \in 0\#9 \wedge Y \in 0\#9$
- Propagator 1 (P_1): $X + Y = 9$, Propagator 2 (P_2): $2 \cdot X + 4 \cdot Y = 24$

1. P_2 schränkt ein: $X \in 0\#8 \quad Y \in 2\#6$
2. P_1 schränkt ein: $X \in 3\#7 \quad Y \in 2\#6$
3. ...
4. P_2 stellt fest: $X = 6 \quad Y = 3$

Constraint Programming: Beispiel Propagation

- Constraint Store enthält: $X \in 0\#9 \wedge Y \in 0\#9$
- Propagator 1 (P_1): $X + Y = 9$, Propagator 2 (P_2): $2 \cdot X + 4 \cdot Y = 24$

1. P_2 schränkt ein: $X \in 0\#8 \quad Y \in 2\#6$
2. P_1 schränkt ein: $X \in 3\#7 \quad Y \in 2\#6$
3. ...
4. P_2 stellt fest: $X = 6 \quad Y = 3$

Constraint Programming: Beispiel Propagation

- Constraint Store enthält: $X \in 0\#9 \wedge Y \in 0\#9$
- Propagator 1 (P_1): $X + Y = 9$, Propagator 2 (P_2): $2 \cdot X + 4 \cdot Y = 24$

1. P_2 schränkt ein: $X \in 0\#8 \quad Y \in 2\#6$
2. P_1 schränkt ein: $X \in 3\#7 \quad Y \in 2\#6$
3. ...
4. P_2 stellt fest: $X = 6 \quad Y = 3$

Constraint Programming: Beispiel Propagation

- Constraint Store enthält: $X \in 0\#9 \wedge Y \in 0\#9$
- Propagator 1 (P_1): $X + Y = 9$, Propagator 2 (P_2): $2 \cdot X + 4 \cdot Y = 24$

1. P_2 schränkt ein: $X \in 0\#8 \quad Y \in 2\#6$
2. P_1 schränkt ein: $X \in 3\#7 \quad Y \in 2\#6$
3. ...
4. P_2 stellt fest: $X = 6 \quad Y = 3$

Constraint Programming: Distribuierung

- Constraint Propagation ist keine vollständige Lösungsmethode
- Abhilfe: „Distribution“
- Unterteilung des Suchraums in mehrere Spaces (mithilfe eines Suchbaums)
- Dazu: „Erfinden“ eines neuen Constraints für den linken Teilbaum; dessen Negation für den rechten Teilbaum

Constraint Programming in Oz: „Send More Money“

Das „Send More Money“-Problem

Gegeben	$SEND + MORE = MONEY$
Gesucht	Ziffern (0-9) für die Buchstaben
Bedingungen	Ziffern paarweise verschieden, $S \neq 0, M \neq 0$
Lösung	$9567 + 1085 = 10652$

Constraint Programming in Oz

```
2  local
3     proc {Money Root}
4        S E N D M O R Y % Variablen deklarieren
5     in
6        Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y) % Datenstruktur
7        Root ::: 0#9                                % Domainconstraint
8        {FD.distinct Root}                          % paarweise versch.
9        S \=: 0                                     % S und M != 0
10       M \=: 0
11       1000*S + 100*E + 10*N + D +                 % Lsg. der Gleichung
12       1000*M + 100*O + 10*R + E =:
13       10000*M + 1000*O + 100*N + 10*E + Y
14       {FD.distribute ff Root}                     % Distribuierung
15    end
16 in
17    {Browse {SearchAll Money}}                     % Loesung berechnen
18    {ExploreOne Money}                             % Suchbaum anzeigen
19 end
```

▸ OPI starten

Funktionale Programmierung

- Grundidee: Auswertung (mathematischer) Ausdrücke
- Ziel: Programmverifikation vereinfachen (s. λ-Kalkül)
- Gewöhnungsbedürftig: Keine Schleifen und Zuweisungen, sondern Rekursion!

Funktionale Programmierung in Oz

```
1  declare
2  fun {Map List Function}
3     case List of nil then nil
4     [] Head|Tail then {Function Head}|{Map Tail Function}
5     end
6  end
7
8  {Browse {Map [1 2 3 4] fun {$ X} (X+X)*2 end}}
```

▸ OPI starten

Klassen und Objekte

- Klasse: Datenstruktur mit Methodentabelle und Attributnamen
- Objekt: Datenstruktur mit Komponenten, darunter u. a.
  - Klasse
  - Zustand
- Besonderheiten: Unterstützung für Mehrfachvererbung, sog. „Features“

Objektorientierte Programmierung in Oz

```
1  declare Counter
2  class Counter
3     attr val
4     meth browse
5        {Browse @val}
6     end
7     meth inc(Value)
8        val := @val + Value
9     end
10    meth init(Value)
11       val := Value
12    end
13 end
14
15 declare C in
16 C = {New Counter init(23)}
17 {C inc(1)}
18 {C browse}
```

▸ OPI starten

Logische Programmierung

- Grundidee: Berechnung als *Deduktion*
- Jedem bekannt: PROLOG
- Aus der Idee der logischen Programmierung ging Constraintprogrammierung hervor

Logische Programmierung in Oz

```
1  declare
2  proc {Append Xs Ys Zs}
3     case Xs of nil then Zs = Ys % 'ask': case, 'tell': Zs=Ys
4     [] X|Xr then Zr in
5        Zs = X|Zr {Append Xr Ys Zr}
6     end
7  end
8
9  declare X Y in
10 {Browse {Append [1 23 abc] [X Y]}}
```

▸ OPI starten

System Module - Überblick

- Erweiterung der Oz Base Environment
- Ermöglichen effektiveres und effizienteres Entwickeln
- Module für verschiedene Einsatzgebiete
- Sozusagen die Standardbibliothek von Oz

Application Programming

- 2 Module für Applikationen und das Laden von Modulen
- Module

  `Application` Zugriff auf die Argumente einer Applikation
  - Vergleichbar mit `String args[]` in Java
  - Parsen der übergebenen Argumente

  `Module` Laden von Modulen

Constraint Programming

- Erweiterungen für Constraint Programming
- Insgesamt 7 Module, die Semantik und Syntax erweitern
- Module

`Search`	Unterstützung von verschiedenen Suchmaschinen
`FD`	Erweiterungen für finite domain Constraints
`Schedule`	Unterstützung für Scheduling-Anwendungen
`FS`	Erweiterung von finite Domain Constraints auf Mengen
`RecordC`	Records als Constraints
`Combinator`	Kombinatoren für Constraints, z.B. `or`
`Space`	Erweiterung für die Spaces

Modul Search

- **Basis Suchmaschinen** - Suche nach einer Lösung, allen Lösungen oder den besten Lösungen.
- **Universelle Suchmaschinen** - Parameterisierte Suche
  - Recomputation - Reduzierung des Suchraums (Space)
  - Beenden der Suche
  - Verschiedene Ausgabe-Modi
- **Parallele Suche** - Verteilung der Suche auf Rechnern im Netzwerk
  - Verteilung durch Aufteilung des Suchbaum in Untersuchbäume
  - Nur von Vorteil, wenn Suchbaum groß genug und gut unterteilbar

Finite Domain Constraints: `FD`

- Erweiterung des Telling auf den Constraint Store für Datenstrukturen
- Reflection für Domains
- Verschiedene vordefinierte Propagatoren, z.B. Generic Propagators oder 0/1 Propagators (binär)

Scheduling Unterstützung: `Schedule`

- Propagatoren und Verteiler für Scheduling-Anwendungen
- *Serialization for unary resources* - Anordnung von Tasks, die gleiche Ressource benötigen, ohne zeitliche Überlappung
- Verteilung der Tasks, so dass jede benötigte Ressource *serialized* ist
- Kumulatives Scheduling - Kapazität einer Ressource darf nicht überschritten werden

Finite Set Constraints: FS

- Neue Constraint-Art
- Assoziation einer n-elementigen Menge mit n finite Domain Variablen
- Mengen von Constraint Variablen sehr nützlich bei kombinatorischer Problemlösung und Natural Language Processing
- Prozeduren, Propagator usw. für den Umgang mit Finite Set Constraints

First-class Computation Spaces: Space

- Zur Programmierung von Interferenzmaschinen
- Prozeduren um mit Computation Spaces umzugehen

Verteilte Programmierung

- Zur Entwicklung von verteilten Anwendungen
- Module

`Connection` Ticket-basierter Verbindungsaufbau zwischen unabhängigen Oz Prozessen, auch lokal. Ticket ist ein String, der übertragen wird.

- One-to-one - Einzelverbindungen
- Many-to-one - Mehrere Verbindungen mit gleichem Ticket

`Remote` Remotesteuerung anderer Oz Prozesse über das Netzwerk.

`URL` Erzeugen und Manipulieren von URLs, für das WWW und Dateisystem

Verteilte Programmierung

- Weitere Module:

`Resolve`	Auflösen von URLs zur einfacheren Auffindung von Daten
`Fault`	Erkennung und Behandlung von Fehlern in der Verteilung
`Discovery`	Lokation von Diensten (Oz-Server) im Netzwerk
`DPInint`	Initialisierung und Konfiguration der Verteilungsschicht
`DPStatistics`	Abfrage von statistischen Informationen

Open Programming

- Verbindung zum Rest der *computational world*
- Module

  `Open` Zugriff auf Dateien, Sockets und Pipes

  `OS` Support für Betriebssysteme. Prozeduren zur Interaktion mit dem Betriebssystem, z.B. Exceptions falls Fehler auftretten

System Programmierung

- Weitere Funktionalität für die Mozart Engine
- Module

`Pickle`	Persistente Speicherung von zustandlosen Werten
`Property`	Operationen auf Eigenschaften (Properties)
`Error`	Formatierung von Fehlermeldungen, z.B. Exceptions
`ErrorFormatters`	Vordefinierte Fehlerformatierungen, z.B. `os` Fehler des Betriebssystems
`Finalize`	Automatische Freigabe von Resourcen, bei gekapselten Daten
`System`	Prozeduren für die Mozart Engine, z.B. Drucken

Window Programming

- Programmierung von Benutzeroberflächen mit Tk
- Module

  `Tk` Modul zur GUI Programmierung. Verschiedene Klassen für die Kommunikation zwischen der Graphic Engine und der Mozart Engine

  `TkTools` Graphische Tools, vordefinierte Klassen für Dialoge, Menüs usw.

Strasheela

- Auf Mozart/Oz basierendes Musik-Komponierungs-System
- Deklarative Angabe einer Musiktheorie
- Festlegen einer Menge von Regeln (Constraints), die zur Erzeugung von Musik erfüllt werden müssen.

MozEclipse

- Integration von Mozart/Oz in die Eclipse IDE
- Derzeit Integration von Mozart in Emacs
- **Ziele:**
  - Einfacherer Umgang mit Oz als bisher
  - Erhöhung des Bekanntheitsgrades durch Integration
- **Start:** Februar 2007
- Zukunft des Projekts ungewiss, bisher keine neuen Aktivitäten

Danke.

Fragen?