Oz: Nebenläufige Programmierung mit Constraints*

Martin Müller[1] und Gert Smolka[2]
Forschungsbereich Programmiersysteme
[1,2] Universität des Saarlandes und
[2] Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken
{mmueller,smolka}@ps.uni-sb.de

Zusammenfassung

Dieser Artikel behandelt die Programmiersprache Oz und das ihr zugrundeliegende Programmiermodell. Oz ist eine nebenläufige Programmiersprache, die Constraintprogrammierung mit funktionaler und objektorientierter Programmierung verbindet. Oz ist als Nachfolger von Hochsprachen wie Lisp, Prolog und Smalltalk entworfen; diese Sprachen sind nur unzureichend für Anwendungen geeignet, die sowohl Problemlösungskomponenten enthalten, als auch Nebenläufigkeit und Reaktivität. Im Vergleich zu Prolog gibt Oz die Idee auf, daß Programme stets auch logische Spezifikationen sein müssen. Andererseits erlaubt Oz die flexible Programmierung von Inferenzmaschinen, deren Leistungsfähigkeit weit über das in Prolog Machbare hinausgeht. Damit steht insbesondere die Funktionalität von CLP-Sprachen wie CHIP bereit.

1 Einleitung

Programmiersprachen dienen dazu, auf Computern ausführbare Berechnungsprozesse zu beschreiben. Das einer Programmiersprache zugrundeliegende Programmiermodell legt dabei die elementaren Berechnungsstrukturen und die prinzipiell möglichen Softwareabstraktionen fest. Programmiermodelle können sich stark in ihrem Abstraktionsgrad unterscheiden. So fallen die Berechnungsstrukturen des imperativen Programmiermodells weitgehend mit denjenigen der üblichen Hardware zusammen, während die Berechnungsstrukturen

*Erschienen in *KI-Künstliche Intelligenz*, Themenheft Logische Programmierung, pp. 55-61. Scientec Publishing, Bad Ems. September 1996.

des funktionalen und logischen Modells aus berechnungsorientierten Logiken (Lambda-Kalkül, Prädikatenlogik mit Resolution) abgeleitet sind.

Logikprogrammierung beruht auf der Entdeckung [10], daß das Hornklauselfragment der Prädikatenlogik mit Hilfe von Resolution so operationalisiert werden kann, daß man ein Programmiermodell erhält, in dem Berechnung als Deduktion erscheint. Aus dieser faszinierenden und wichtigen Einsicht wurde die Konzeption logischer Programmiersprachen abgeleitet, in denen Programme stets auch logische Spezifikationen sein müssen.

Die Entwicklung der Programmiersprache Oz [27] und des ihr zugrundeliegenden Programmiermodells OPM [25] beruht auf folgenden Annahmen.

1. Die constraintorientierten und nebenläufigen Weiterentwicklungen der Logikprogrammierung bilden eine hervorragende Grundlage für ein hochsprachliches Programmiermodell.

2. Zustand ist ein zentraler Aspekt von Programmierung. Das gilt insbesondere für objektorientierte und nebenläufige Programmierabstraktionen.

3. Logische Programmiersprachen unterstützen die Konstruktion von Softwareabstraktionen nur unzureichend. Funktionale Sprachen wie Scheme sind ihnen darin deutlich überlegen.

4. Die Berechnungsstrukturen des logischen Programmiermodells sind mit denen des funktionalen und objektorientierten Modells durchaus verträglich. Allerdings muß dazu die Idee aufgegeben werden, daß Programme immer auch logische Spezifikationen sein müssen.

OPM ist ein nebenläufiges Programmiermodell mit Constraints, das funktionale und objektorientierte Programmierung subsumiert. Es basiert auf der Metapher des Constraintspeichers [19] als ein leistungsfähiges Medium für die Kommunikation und Synchronisation nebenläufiger Berechnungen. Durch die Integration von Zustand und prozeduraler Abstraktion höherer Ordnung liefert OPM eine flexible Grundlage für die Modellierung nebenläufiger Objekte.

OPM sieht eine Erweiterung vor, mit der zudem constraintbasiertes Problemlösen unterstützt wird, wie es in der KI oder im Operations Research eine lange Tradition hat. Diese Erweiterung ist in Oz realisiert und ermöglicht die Programmierung von Inferenzmaschinen, deren Leistungsfähigkeit weit über das in Prolog Machbare hinausgeht. Insbesondere stellt Oz damit auch die Funktionalität von CLP-Sprachen (Constraint Logic Programming) wie CHIP oder Prolog III bereit.

Zwei in Oz realisierte Inferenztechniken sind Constraintpropagierung und Constraintdistribuierung. In Kombination liefern sie eine allgemeine Methode zur Lösung von Constraintproblemen. Ihre Verfügbarkeit in einer Programmiersprache ermöglicht es, auf flexible Weise Inferenzmaschinen zu programmieren, mit ihnen zu experimentieren, und sie elegant in komplexe Anwendungen zu integrieren. So lassen sich etwa Inferenzmaschinen in Oz leicht in nebenläufige Objekte einbetten, um Alternativen unabhängig voneinander explorieren zu können. Das ist auch von unmittelbarer Bedeutung für autonome Agenten, die oftmals – unabhängig voneinander, aber nebenläufig – komplexe Inferenzleistungen erbringen.

Je nach Anwendung müssen Inferenzmaschinen sehr verschiedene Berechnungsdienste erbringen. Man betrachte etwa spezielle Probleme wie Scheduling oder constraintbasiertes Chartparsing. Oder man denke an verschiedene Suchstrategien, wie etwa die Suche nach einer, nach allen, oder nach einer besten Lösung; oder an Suchprozesse, die ihre Ergebnisse nach Bedarf berechnen (lazy) und die abgebrochen werden können, wenn ein Zeitlimit überschritten ist oder ein anderer Prozeß schon ein Ergebnis geliefert hat.

Übersicht. Abschnitt 2 faßt die Entwicklung von Oz zusammen. Abschnitt 3 skizziert verschiedene Programmierparadigmen und vergleicht diese mit Oz. Die Abschnitte 4 bis 7 beschreiben die Sprache Oz und ihre Implementierung DFKI Oz. Die Abschnitte 8 und 9 illustrieren konkrete Anwendungen und derzeitige Forschungsvorhaben im Kontext von Oz.

Dank. Wir danken Norbert Fuchs, Martin Henz, Joachim Niehren, Tobias Müller, Konstantin Popow, Joachim Walser und Jörg Würtz für Kommentare, und allen Kollegen aus dem Forschungsbereich Programmiersysteme für die Verwendung ihrer Materalien.

2 Projekte

Die Entwicklung von Oz begann 1990 mit der Gründung des Forschungsbereichs Programmiersysteme am DFKI Saarbrücken. Der erste Oz-Prototyp war im September 1992 lauffähig. DFKI Oz 1.0 wurde im Januar 1994 freigegeben, Version 1.1 folgte im Mai 1994. Im November 1995 wurde in der Schweiz der erste Internationale Workshop zur Oz-Programmierung (WOz'95) durchgeführt [5], organisiert vom IDIAP, dem Institut Dalle Molle für Perzeptive Künstliche Intelligenz. Im Herbst 1996 wird die erweiterte und verbesserte Version DFKI Oz 2.0 freigegeben werden.

Die bisherige Forschung wurde am DFKI Saarbrücken und der Universität des Saarlandes durchgeführt, mit Förderung im HYDRA-Projekt durch das BMBF, der Esprit Basic Research Action ACCLAIM, und der Esprit Working Group CCL. Die Entwicklung von Oz hat von einer engen Kooperation mit den Entwicklern von LIFE [2] am Digital PRL und von AKL [9] am SICS profitiert.

Seit 1996 finden Forschung und Entwicklung an Oz im Projekt PERDIO [26] am DFKI und im Rahmen des neuen SFB 378 an der Universität des Saarlandes statt [21]. PERDIO („Persistent and Distributed Programming in Oz") wird gemeinsam mit dem Schwedischen Institut für Informatik SICS durchgeführt. Im SFB werden das Programmiermodell von Oz sowie seine theoretischen Grundlagen weiterentwickelt (Projekt NEP, „Nebenläufige Programmiermodelle"), und es sollen, basierend auf nebenläufiger Constraintprogrammierung, neue Modelle für die Grammatikverarbeitung entwickelt werden (Projekt NEGRA, „Nebenläufige grammatische Verarbeitung").

3 Programmiersprachen im Vergleich

Seit den 70er Jahren ist die Entwicklung höherer Programmiersprachen ein Feld aktiver Forschung. Basierend auf verschiedenen Modellen von Berechnung haben sich verschiedene Traditionen herausgebildet.

Funktionale Programmierung basiert auf der Idee von Berechnung als *Auswertung von Ausdrücken* (z. B. arithmetischen Ausdrücken). Die Hauptströmungen sind durch ungetypte Sprachen wie Lisp und Scheme, bzw. durch getypte Sprachen wie SML und Haskell beschrieben. Zu den wichtigsten Beiträgen funktionaler Programmierung gehört es, die Universalität funktionaler Abstraktion nachzuweisen. Das gilt insbesondere für Scheme, das schon frühzeitig zur Modellierung komplexer (z. B. objektorientierter) Datenstrukturen benutzt wurde [1].

Logikprogrammierung im Stil von Prolog basiert auf dem Prinzip von *Berechnung als Deduktion*. Zu den wesentlichen Beiträgen zum Programmiersprachdesign gehören die logische Variable und der Einsatz von generischen Suchverfahren. Die Pionierleistung von Prolog, Berechnung mit unvollständiger Information zu ermöglichen, wurde später zur Constraintprogrammierung verallgemeinert. Die nebenläufige Logikprogrammierung [22] erkennt, daß mit logischen Variablen komplexe Kommunikations- und Synchronisationsprotokolle elegant realisiert werden können. Gleichzeitig nimmt sie Abstand von der Identifikation von Berechnung und Deduktion. Die nebenläufige Constraintprogrammierung [11, 19, 25] mit Sprachen wie AKL oder Oz führt die nebenläufige und die constraintbasierte Logikprogrammierung wieder zusammen, ergänzt um Ideen aus der Theorie kommunizierender Prozesse.

Objektorientierte Programmiersprachen wie etwa Smalltalk organisieren Berechnung mithilfe von *Objekten*. Objekte haben einen veränderlichen Zustand, auf den andere Objekte nur über gewisse objektspezifische Methoden einwirken können (Datenabstraktion). Oft wird durch „Vererbung" die Wiederverwendung von Code unterstützt. Objekte eignen sich gut zur Strukturierung von imperativer Berechnung, insbesondere in Anwendungen, die mit der externen Umgebung interagieren (z. B. über Dateien oder eine graphische Benutzerschnittstelle).

Aktoren und Prozesse beschreiben nebenläufige Berechnung als radikale Verallgemeinerung von sequentieller Berechnung. Es war Hewitts Vision [8], Berechnung allein durch ein dynamisches Geflecht von Aktoren bzw. Prozessen zu modellieren, die durch Nachrichten miteinander kommunizieren, und Milner mit Kollegen gelang es, eine elegante formale Grundlage für ein solches Modell zu finden (Pi-Kalkül). Eine jüngere Entwicklung ist die Sprache Pict [18], deren formale Grundlage der Pi-Kalkül ist.

Oz verbindet Konzepte aus allen genannten Sprachfamilien, nämlich Prozeduren höherer Ordnung (die Funktionen höherer Ordnung subsumieren), logische Variablen und Constraints, sowie nebenläufige Objekte. Zu den wichtigsten Unterschieden gehören im Einzelnen:

1. Funktionale Programmierung wird um das Konzept partieller Information erweitert: In Oz kann mit Referenzen gerechnet werden, deren eindeutiger Wert noch nicht (oder nicht vollständig) feststeht.

2. Von der Logikprogrammierung unterscheidet Oz die Möglichkeit, verschiedene Suchprobleme nebenläufig aber unabhängig voneinander bearbeiten zu können. Zudem stehen Prozeduren höherer Ordnung und ein leistungsfähiges Objektsystem zur Verfügung.

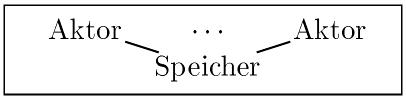
3. Das Objektmodell von Oz [6] basiert auf Kommunikation über gemeinsame Datenstrukturen anstelle von Nachrichten. Die logische Variable garantiert dabei die Stabilität von Synchronisationsbedingungen. Daß heißt, die Möglichkeit von Kommunikation ist nicht abhängig von der Reihenfolge oder dem Zeitpunkt nebenläufiger Aktivitäten.

Für formale Grundlagen von Oz siehe den Gamma-Kalkül [23], das Papier [15] für einen exakten Vergleich mit Lambda- und Pi-Kalkül (Programmierung mit Funktionen und Prozessen), und [16] für eine Erweiterung des Gamma-Kalküls um Constraints. Constraintprogrammierung und die Integration von Suche sind in [20, 24] beschrieben.

4 Das Oz Programmiermodell

Dieser und die folgenden zwei Abschnitte skizzieren die Sprache Oz und das ihr zugrundeliegende Programmiermodell OPM [25]. Für ausführliche Darstellungen siehe [25, 12, 24, 27] und die enthaltenen Referenzen. Alle Angaben beziehen sich auf die Oz Version 2, die DFKI Oz 2.0 zugrunde liegt.

Das Berechnungsmodell. Berechnung im Oz Programmiermodell OPM gliedert sich in sogenannte *Berechnungsräume*.



Ein Berechnungsraum enthält Berechnungseinheiten namens *Aktoren* (oder *tasks*), die über einen gemeinsamen (abstrakten) *Speicher* miteinander kommunizieren. Berechnung schreitet durch *Reduktion von Aktoren* voran. Aktoren werden nacheinander reduziert. Ein Aktor kann bei Reduktion den Speicher verändern und neue Aktoren erzeugen, er selbst verschwindet.

Unter den Aktoren gibt es solche, die Information wie etwa Constraints oder Prozeduren in den Speicher eintragen (*tell*) und solche, die erst dann reduzierbar werden, wenn bestimmte Information im Speicher verfügbar wird (*ask*), zum Beispiel Konditionale oder Prozedurapplikationen.

Constraints und Konditionale. Constraints sind logische Formeln, die die möglichen Werte von Variablen beschreiben. Constraintsprachen verwenden Constraints zur Modellierung von Datenstrukturen. Die elementaren Constraints in Oz sind Gleichungen zwischen Variablen (durch Großbuchstaben denotiert), oder zwischen Variablen und Strukturen wie Zahlen, Tupel oder Verbunden.

```
X=Y   X=1   X=3.14
X=oz  X=pair(Y Z)
X=employee(name:N phone:P manager:M)
```

Desweiteren unterstützt Oz Constraints über endlichen Bereichen ganzer Zahlen (*finite domains*) und Merkmalconstraints. Zum Beispiel beschränkt

```
X :: 1#100
```

die Variable `X` auf ganze Zahlen zwischen 1 und 100 ($X \in \{1, \ldots, 100\}$), und die folgenden Merkmalsconstraints beschreiben Pfadgleichungen, wie sie für Constraintgrammatiken typisch sind.

```
X^head^agreement^number = singular
VP^head^subject = NP^head
```

Konditionale warten solange, bis ihr Bedingungsteil (*guard*) oder dessen Negation von den Constraints im Speicher erfüllt wird, und reduzieren dann zu einem ihrer Zweige. Diese automatische Sychronisation ist nützlich etwa zur Programmierung interaktiver und reaktiver Anwendungen. Zum Beispiel reduziert das folgende Konditional zum Aktor E_1, wenn der Speicher den Constraint `X=pair(1 2)` enthält, und zu E_2, wenn er `X=1` enthält.

if `X = pair(_ _)` **then** E_1 **else** E_2 **end**

Erfüllung von Bedingungen durch den Speicher wird durch logische Implikation (*entailment*) modelliert [11].

Prozeduren. In allen Programmiersprachen sind Prozeduren der grundlegende Abstraktionsmechanismus. Während Prolog die Verwendung von Prozeduren (bzw. Prädikaten) beschränkt, unterstützt Oz – wie funktionale Sprachen – Prozeduren in voller Allgemeinheit. Das bedeutet, daß Prozeduren andere Prozeduren als Argumente nehmen und als Resultat erzeugen können. Die folgende Prozedur erster Stufe implementiert die Längenfunktion im relationalen Stil. Ist ihr erstes Argument `Xs` eine endliche Liste `X|Y|...|nil` der Länge n, wird das zweite Argument `N` zu einem Baum `succ(succ(...))` der Tiefe n eingeschränkt.

```
proc {Len Xs N}
   if Xs=nil then N=0 else
      if X Xr in Xs=X|Xr then
         local M in N=succ(M) {Len Xr M} end
      end
   end
end
```

Syntaktischer Zucker erlaubt eine äquivalente funktionale Darstellung wie folgt.

```
fun {Len Xs}
   case Xs of nil then 0
   [] X|Xr then succ({Len Xr})
   end
end
```

Eine typische funktionale Prozedur in Oz ist die Prozedur `MakeMultiplier`. Sie nimmt ein zahlwertiges Argument `N`, und liefert eine Prozedur `TimesN`, die jedes ihrer Argumente mit `N` multipliziert. Man beachte, daß Variablen in Oz an Prozeduren gebunden werden können (hier `TimesN`).

```
fun {MakeMultiplier N}
   fun {TimesN M} N*M end  in  TimesN
end
```

Objekte und Zustand. Oz-Objekte sind benutzerdefinierte abstrakte Datenstrukturen mit synchronisiertem Zustand. Der Zugriff auf den Zustand eines Objekts ist durch seine sogenannten Methoden definiert und auf diese beschränkt. Dadurch wird Datenabstraktion sichergestellt. Die Ausführung einer Methode wird durch Applikation des Objekts auf eine Nachricht angefordert („Nachrichtensendung“).

```
{O Msg}
```

Nachrichten werden durch Constraints im gemeinsamen Speicher modelliert.

```
Msg = send(X Y)
```

Das macht die Formulierung von komplexen Synchronisationsbedingungen und Kommunikationsprotokollen möglich.

Ein Objekt serialisiert alle eintreffenden Nachrichten, so daß zu jedem Zeitpunkt höchstens eine Methode Zugriff auf den Zustand hat. So wird die Konsistenz des Objektzustands im nebenläufigen Kontext garantiert. Darin sind Objekte Hoares *Monitoren* ähnlich, einer Programmierabstraktion für Betriebssysteme, mit denen die Betriebsmittel eines Computers modelliert werden können.

Der Zustand von Objekten wird in Oz durch *Zellen* modelliert. Zellen sind umsetzbare Referenzen auf Datenstrukturen. Das Umsetzen einer Zelle erfolgt in einer atomaren Operation, die den momentanen Inhalt der Zelle liest und ihn durch einen neuen ersetzt. Damit sind Zellen in der Lage, den klassischen imperativen Speicher zu modellieren. Prozeduren und Zellen reichen aus, um das gesamte Objektsystem von Oz zu definieren, das auf Klassen basiert und sowohl Mehrfachvererbung als auch Delegation unterstützt [24, 6].

5 Constraintprogrammierung

Propagierer. Der Constraintspeicher von Oz enthält nur „einfache“ Constraints, darunter alle oben eingeführten. Einfache Constraints sind solche, die effizient und inkrementell gelöst werden können, bzw. deren Gültigkeit oder Unlösbarkeit effizient getestet werden kann. Da Constraints in Oz zur Modellierung von Datenstrukturen und als elementares Kommunikationsmedium dienen, ist ihre effiziente Handhabung notwendig.

Komplexe Constraints wie etwa arithmetische Gleichungen werden durch *Propagierer* realisiert. Propagierer haben eine deklarative Semantik, die durch ihre operationale Semantik korrekt aber unvollständig implementiert wird. Während seiner Reduktion macht ein Propagierer solche einfachen Constraints explizit, die seine deklarative Semantik impliziert. Die deklarative Semantik eines Propagierers sagt demnach, um *welchen* Constraint der Speicher höchstens verstärkt werden kann, während die operationale Semantik beschreibt, *wann* genau das passiert. Typische Propagierer sind die folgenden.

```
X <: Y
2*X + 4*Y =: 3*Z
X*X + Y*Y =: Z*Z
```

Propagierer können mit Prozeduren implementiert werden. So ist zum Beispiel die Prozedur `Len` oben ein (funktionaler) Propagierer für die Relation „die Liste Xs hat die Länge N“. Die logische Semantik dieser Prozedur ist, daß für alle Xs und N, die Relation $Len(Xs\ N)$ genau dann gilt, wenn entweder

1. $Xs = nil \wedge N = 0$ oder

2. es X, Xr, und M gibt, so daß $Xs = X|Xr \wedge N = succ(M) \wedge Len(Xr\, M)$.

(Man beachte, daß dies genau die Clarksche Vervollständigung der üblichen Kodierung des Längenprädikats in Prolog ist.)

Es kommt vor, daß dieselbe deklarative Semantik durch Propagierer mit unterschiedlicher operationaler Semantik realisiert wird. Die Wahl das Propagierers mit dem richtigen operationalen Verhalten kann für die Komplexität einer Lösung (Zeit und Speicher) entscheidend sein. Schwierige kombinatorische Probleme sind oft erst nach Wahl des richtigen Propagierers praktisch handhabbar (z. B. Scheduling, siehe Abschnitt 8).

Die Stärke von Propagierung besteht darin, daß sie ein *deterministisches*, nichtsuchendes Verfahren ist. Im allgemeinen ist Propagierung jedoch unvollständig. Es kann passieren, daß durch Propagierung allein eine eindeutige Lösung oder die Unlösbarkeit von Constraints nicht erkannt wird.

Distribuierung. Zur vollständigen Lösung von Problemen muß man auf Distributierung zurückgreifen. Jeder Distribuierungsschritt entspricht einer vollständigen Fallunterscheidung. Sobald Propagierung nicht mehr zur Lösung eines Problems P beitragen kann, wird dieses in komplementäre Teilprobleme P_1 und P_2 so aufgespalten oder *distribuiert*, daß P äquivalent zur Konjunktion $P_1 \wedge P_2$ ist. Das geschieht wenn möglich so, daß in P_1 und P_2 wieder Propagierung möglich wird. Die Teilprobleme werden unabhängig voneinander gelöst.

Oft kann man einen Constraint C bestimmen, dessen Negation $\neg C$ wieder ein Constraint ist, und zu $P \wedge C$ und $P \wedge \neg C$ übergehen. Legt P zum Beispiel fest, daß $X \in \{1, 2, 3\}$, so kann C zu $X = 1$ und $\neg C$ zu $x \neq 1$ gewählt werden. Die Suche nach einem geeigneten C kann durch einmalige Wahl einer *Distribuierungsstrategie* für ein ganzes Problem festgelegt werden.

Wo das nicht geht, kann man durch explizites Setzen eines Auswahlpunktes (*choice points*) einen Distribuierungsschritt festlegen. Zum Beispiel denotiert der folgende Ausdruck die Disjunktion $D = (NP\hat{}agr\hat{}person = first) \vee (NP\hat{}agr\hat{}person = second)$ und bestimmt eine Distribuierung von $P \wedge D$ in $P \wedge (NP\hat{}agr\hat{}person = first) \vee (NP\hat{}agr\hat{}person = second)$ und $P \wedge (NP\hat{}agr\hat{}person = first) \vee (NP\hat{}agr\hat{}person = second)$.

```
choice NP^agr^person = first
[] NP^agr^person = second
end
```

Die unabhängige Bearbeitung der Teilprobleme P_1 und P_2 ist dadurch gewährleistet, daß für sie je ein neuer *Berechnungsraum* (siehe Abschnitt 4) erzeugt wird. Dieser Datentyp macht Berechnungsräume mit lokalem Speicher und lokalen Aktoren als *first-class citizens* verfügbar [27]. (Er vereinfacht und subsumiert das frühere Konzept von eingebetteter Suche [20]). Die Kapselung von Problemen in Berechnungsräume ermöglicht es auch, autonome Agenten auf elegante Weise mit Problemlösungskomponenten auszustatten.

6 Nebenläufigkeit

Nebenläufigkeit und Parallelität. OPM ist nebenläufig, nicht aber parallel. *Nebenläufigkeit* ist eine *Programmierabstraktion*, die bedeutet, daß verschiedene Berechnungsaktivitäten unabhängig voneinander voranschreiten, miteinander kommunizieren, und sich aufeinander synchronisieren können. *Parallelität* ist demgegenüber ein *physikalisches Phänomen*, das besagt, daß sich Operationen der Hardware zeitlich überlappen.

Nebenläufigkeit schließt selbstverständlich Parallelität nicht aus. Eine nebenläufige Sprache erfüllt bereits wichtige Voraussetzungen für Parallelisierung, wie auch für *Verteilung*, welche Nebenläufigkeit mit Parallelität verbindet. Aber die nebenläufige Semantik von Oz erleichtert die Programmierung, indem sie Parallelität ganz auf die Implementierung beschränkt (siehe auch Abschnitt 9).

Reduktion von Aktoren. Typischerweise sind mehrere Aktoren zur selben Zeit reduzierbar (d.h., in der Lage, einen Berechnungsschritt zu machen). Eine *Reduktionsstrategie* entscheidet, welcher von ihnen als nächster reduziert wird. Die Wahl der Reduktionsstrategie hat wesentlichen Einfluß auf Fairness, Reaktivität und Effizienz, und damit auch auf den Programmierstil.

Die Reduktionsstrategie kann zum Beispiel festlegen, daß alle Aktoren fair reduziert werden müssen, so daß jeder reduzierbare Aktor auch tatsächlich reduziert wird. Dieser Ansatz erzeugt feinkörnige Nebenläufigkeit (Hewitt spricht von *ultraconcurrency* [8]). Er hat angenehme theoretische Eigenschaften, ist aber praktisch schwer zu handhaben. Die Erfahrung mit feinkörniger Nebenläufigkeit in Oz hat gezeigt, daß eine stärker sequentielle Organisation von Programmen nötig ist.

In Oz werden Aktoren durch *sequentielle Komposition* verknüpft. Der Aktor

$E\ ;\ F$

zum Beispiel legt fest, daß E vor F reduziert wird. (Wir weichen hier mit der expliziten Notation ; von konkreter Oz-Syntax ab, in der es kein Symbol für sequentielle Komposition gibt). Durch sequentielle Komposition werden Aktoren zu sequentiellen sogenannten *Threads* gruppiert. Um die Ausdrücke E und F nebenläufig zu reduzieren, muß für E *explizit* ein neuer Thread erzeugt werden.

thread E **end** ; F

Man beachte, daß das *Potential* an Nebenläufigkeit durch sequentielle Threads nicht eingeschränkt wird. (Die Tatsache andererseits, daß sequentielle Threads durch explizite Synchronisation von Aktoren ausdrückbar sind, ist von theoretischer aber nur geringer praktischer Relevanz.) Sequentielle Threads zerlegen eine Berechnung in größere Teile mit starken Invarianten. Das erleichtert die Fehlersuche, ermöglicht die effizientere Ausführung von Programmen, und erlaubt neue Programmiertechniken. Zum Beispiel kann man ausdrücken, daß ein Aktor erst dann reduzierbar wird, wenn die Reduktion anderer Aktoren abgeschlossen ist. Im folgenden Programm wird E_3 erst nach Terminierung von E_1 und E_2 reduziert, da `{Wait X2}` solange wartet, bis `X2` determiniert ist (hier durch `X2=X1` und `X1=done`).

```
local X1 X2 in
  thread E1 X1=done end ;
  thread E2 X2=X1 end ;
  {Wait X2} ;
end
E3
```

7 DFKI Oz

DFKI Oz ist eine interaktive Implementierung von Oz, deren Effizienz mit der kommerzieller Prolog- und Lispsysteme vergleichbar ist. Die Programmierumgebung von DFKI Oz basiert auf GNU Emacs. Das System besitzt einen nebenläufigen Browser, eine objektorientiertes Graphiksystem basierend auf Tcl/Tk, mächtige Interoperabilitätsmechanismen (Sockets, C, C++), automatische Speicherbereinigung, einen inkrementellen Übersetzer, sowie Unterstützung für *stand-alone* Anwendungen. Der Explorer ist ein interaktives, graphisches Werkzeug zur Analyse von Suchproblemen (Schulte in [5]).

8 Anwendungen

Zu den Anwendungen von Oz gehören Multiagentensysteme, Simulation, Scheduling, Konfigurierung, Frequenzzuweisung für mobile Telekommunikation, Entwicklung graphischer Benutzerschnittstellen, Sprachverarbeitung (Parsing), automatische musikalische Komposition und Programmierung virtueller Realität. Wir gehen hier auf einige näher ein. Mehr Information findet sich unter URL `http://www.ps.uni-sb.de/www/oz/projects/`, sowie in [5].

Multiagentensysteme. In verschiedenen Projekten innerhalb und außerhalb des DFKI werden Multiagentenanwendungen in Oz entwickelt. Besonders geeignet sind dabei solche Szenarien, in denen autonome Agenten komplexe Deduktionsleistungen erbringen müssen. Konkrete Projekte behandeln die Optimierung von Transportproblemen (Flottenmanagement) und die automatische Terminplanung, sowie die Spezifikation von Agenten oder die Entwicklung von Planungskomponenten (siehe auch Rosinus *et al.*, Schmeier und Schupeta, sowie Selvaratanam und Ahmad in [5]).

Scheduling. Schedulingprobleme sind inhärent schwierig. Das heißt, sie sind nicht nur im schlechtesten Fall exponentiell, sondern schon für kleine Instanzen schwierig. Größere Instanzen sind oft praktisch gar nicht mehr lösbar. Eine Teilklasse sind die sogenannten *job-shop*-Probleme. Dabei müssen bestimmte Jobs so auf eine Anzahl von Maschinen verteilt werden, daß die Gesamtlaufzeit der Maschinen minimal ist. Kompliziert wird das Problem dadurch, daß Teiljobs nur auf bestimmten Maschinen und unter Reihenfolgebeschränkungen ausführbar sind. Ein konkretes Job-Shop-Problem, das sich lange als besonders hartnäckig erwies, trägt den Namen MT10 [14]. Erst 1989 wurde MT10 vollständig gelöst, indem der Optimalitätsbeweis für die (bekannte) Lösung erbracht wurde (siehe [4] für Details). Seitdem ist es auf verschiedene Weise mit Constrainttechnologie gelöst worden; mit zufriedenstellender Effizienz von den kommerziellen Systemen CHIP und ILOG, sowie von DFKI Oz.

Stundenplanerstellung. Die Erstellung von Stundenplänen z. B. für Schulen ist ein schwieriges kombinatorisches Problem, das komplexe Constraints und anwendungsspezifische Suchstrategien erfordert. Mit Techniken des Operations Research (OR) und traditionellen Verfahren der Constraintlogikprogrammierung sind beide jedoch nicht auf hinreichend hoher Abstraktionsebene ausdrückbar. Insbesondere sind die OR-Methoden oft nicht ausreichend flexibel, und ihre Anpassung im Einzelfall zu teuer. Hier kann eine programmiersprachlicher Ansatz durch erhöhte Flexibilität von Vorteil sein. In [7] wird das für Oz am Beispiel der Erstellung von Stundenplänen belegt.

Sprachverarbeitung. Von Anfang an war die Computerlinguistik ein wichtiges Anwendungsfeld für die Logikprogrammierung. Insbesondere constraintbasierte Grammatikformalismen wie etwa HPSG erlauben Modellierung und Implementierung von „Parsing als Deduktion“ [17], was mit der Idee von „Berechnung als Deduktion“ korrespondiert; zudem werden in beiden Bereichen Baumconstraints zur Modellierung von Datenstrukturen verwendet werden Zur effizienten Implementierung von Grammatikverarbeitung ist jedoch die eingebaute Inferenzmaschine von Prolog (Tiefensuche mit Rücksetzen) oft zu restriktiv. Es ist wesentlich, daß ohne Effizienzverlust verschiedene Such- (oder Parsing-) strategien eingesetzt und ggfs. anwendungsspezifische entwickelt werden können. Allgemein benötigt man zur Grammatikentwicklung sehr flexible Ablaufmechanismen (z. B. prozedurale Abstraktion, Objekte) und verschiedenartige Constraints (z. B. Endliche Bereiche, getypte Merkmalsstrukturen, Mengenconstraints), sowie Möglichkeiten zur graphischen Präsentation. Viele der genannten Einzelaspekte sind in logischen Programmiersprachen wie z. B. LIFE [2] integriert worden. Ihre Kombination in Oz ist derzeit jedoch einzigartig (siehe auch Erbach in [5]).

Verteilte Rollenspiele. In einem Fortgeschrittenenpraktikum an der Universität des Saarlandes wird derzeit ein sogenanntes MUD namens Munchkins in Oz entwickelt. MUDs sind Rollenspiele, die üblicherweise auf einem zentralen Server laufen, an dem sich die Spieler via Telnet anmelden. Ziel des Praktikums ist die Entwicklung eines verteilten MUDs mit einer graphischen Schnittstelle (siehe Henz *et al.* in [5]). Die Einfachheit, mit der autonome Agenten in Oz geschrieben werden können, erleichtert die Entwicklung von komplexen Spielsszenarien. Die Verteilung basiert zunächst auf der Socket-Schnittstelle von Oz und ist daher nicht vollständig transparent. Für PERDIO können verteilte Rollenspiele als geeignete Testszenarien dienen

Virtuelle Realität. Das Schwedische Institut für Informatik SICS entwickelt mit DIVE (*DIstributed Virtual Environment*, [3]) einen Baukasten für verteilte Anwendungen von Virtueller Realität. DIVE erlaubt es, daß verschiedene Benutzer und Anwendungen in einer virtuellen Umgebung und miteinander in Echtzeit kommunizieren und interagieren. Zur Beschreibung interessanter Verhaltensmuster wie denen von Fahrzeugen oder Personen werden schnell komplexe Programme benötigt. Das macht eine Hochsprache wünschenswert, die es erlaubt Anwendungen in nebenläufige kommunizierende Agenten zu strukturieren, und die Reaktivität und Echtzeitkontrolle unterstützt. Im Unterschied zu Lisp, Prolog oder Smalltalk, paßt Oz gut auf dieses Anforderungsprofil. Axling *et al.* beschreiben in [5] ein Interface von Oz zu DIVE, das es erlaubt, DIVE-Applikationen über Oz-Objekte zu steuern.

9 Forschungsaktivitäten

Neben der stetigen Weiterentwicklung der Implementierung und der formalen Methoden zur Analyse von Oz, findet Forschung in den folgenden Bereichen statt.

Constraintprogrammierung. Um Oz für Probleme realistischer Größe in Bereichen wie Stundenplanung, Scheduling oder Konfiguration einsetzen zu können, ist weitere Arbeit erforderlich. Das berührt insbesondere die Entwicklung und Integration von Propagierern, die nichttriviale Algorithmen implementieren, z. B. solche aus dem Operations Research.

Sprachverarbeitung. Im Rahmen des SFB 378 [21] soll Oz zur Grammatikverarbeitung eingesetzt und seine Eignung für die semantischen Modellierung untersucht werden. Zu weiteren Anwendungen in diesem Kontext siehe Ciortuz sowie von Cochard und Vial in [5], die sich mit automatischer Übersetzung bzw. Spracherkennung beschäftigen.

Verteilung. Die Anzahl an Computern und ihre Vernetzung wächst weltweit mit rasantem Tempo. Zukünftige Applikationen werden also zunehmend über ein Rechnernetz verteilt sein. Um diesen Anforderungen gerecht zu werden, muß der Programmieraufwand für verteilte Anwendungen auf das Maß für zentralisierte Anwendungen reduziert werden. Im Projekt PERDIO, das in Kooperation mit dem SICS durchgeführt wird, soll Oz zu einer Hochsprache für die Programmierung von verteilten Anwendungen weiterentwickelt, implementiert und exploriert werden. Dabei soll das Ergebnis einer Berechnung unabhängig von den Rechnerknoten sein, auf denen es erbracht wurde (Transparenz), obgleich der Programmierer die volle Kontrolle über Verteilung und Kommunikation behält.

Parallelisierung. DFKI Oz ist eine sequentielle Implementierung von Oz und kann daher nicht davon profitieren, daß Arbeitsplatzrechner zunehmend als Mehrprozessorsysteme ausgestattet werden. Ein laufendes Forschungsvorhaben versucht, dem durch eine parallele Implementierung von Oz abzuhelfen. Leitlinie dabei ist, daß nur die *explizite* Nebenläufigkeit in Oz (Threads) zur Parallelisierung genutzt werden soll (keine automatische Parallelisierung).

Typanalyse. Oz ist eine dynamisch getypte Sprache wie Scheme oder Smalltalk. Das heißt, die wohlgetypte Verwendung primitiver Operationen wird erst zur Laufzeit getestet und garantiert etwa, daß Addition nur auf Zahlen, aber nicht auf Paare oder Prozeduren angewendet wird. Stark getypte Sprachen wie SML oder Haskell testen Wohlgetyptheit zur Übersetzungszeit und garantieren die Abwesenheit von Typfehlern zur Laufzeit. Typen sind von großem Nutzen für Programmverifikation und -dokumentation, sowie für effiziente Übersetzung. Starke Typisierung kann jedoch die Flexibilität des Programmierens erheblich einschränken. Insbesondere gibt es noch kein befriedigendes Typsystem, das objektorientierte Programmierung in vollem Umfang erlauben würde, und das gleichzeitig einfach und intuitiv wäre. Aus diesem Grund wird für Oz ein „weiches" Typsystem entwickelt (*soft typing*), das Vorteile von Typanalyse für Oz verfügbar machen soll. Typanalyse kann durch ein Erfüllbarkeit gewisser Constraints beschrieben werden [13], was ihre Implementierung in einer Constraintprogrammiersprache wie Oz nahelegt.

Debugging. Neue Programmiermodelle benötigen neue Modelle zur Fehlersuche und -behandlung. Ein solches vermittelt der Oz-Browser, der es ermöglicht, die inkrementelle Berechnung von Werten durch Constraints nebenläufig zu beobachten (siehe Popow in [5]). Darüber hinaus werden – entsprechend den verschiedenen Aspekten von Oz bzw. OPM – verschiedene Ansätze verfolgt, darunter die folgenden.

- Zur Analyse der Kommunikation *nebenläufiger Objekte* existieren zwei Prototypen, die es erlauben, die dynamische Veränderung von Objektzuständen und die Folge empfangener Nachrichten zu verfolgen (siehe

Henz und Reiter sowie Cochard und Nguyen in [5]).

- Mit dem Explorer (siehe Schulte in [5]) bietet DFKI Oz ein Werkzeug, mit dem Suchbäume visualisiert und schrittweise durchlaufen werden können. Der Explorer bietet komfortable Möglichkeiten, Probleme und deren Suchraum zu analysieren und problemspezifische Suchstrategien zu entwickeln.
- Ausnahmebehandlung (*exception handling*) wird dem Programmierer erlauben, Fehlersituationen zur Laufzeit flexibel zu begegnen. Das ist von besonderer Bedeutung im Kontext eines verteilten und offenen Systems, dessen Fehlverhalten nicht *a priori* klar ist.

Kontaktadressen. DFKI Oz und seine Dokumentation ist frei erhältlich via ftp von `ftp.ps.uni-sb.de` im Verzeichnis `/pub/oz`, oder via WWW von `http://www.ps.uni-sb.de/www/oz`. Über diese Adresse sind auch die übrigen Veröffentlichungen des FB Programmiersysteme des DFKI und der Universität des Saarlandes verfügbar. Das Oz-Team ist unter `oz@ps.uni-sb.de` erreichbar, und für Diskussion unter Benutzern gibt es die Mailingliste `oz-users@ps.uni-sb.de`.

Literatur

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[2] H. Aït-Kaci, B. Dumant, R. Meyer, A. Podelski, and P. Van Roy. The Wild LIFE handbook, 1994. Prepublication edition available as part of the Wild LIFE release. The release is available by anonymous ftp from gatekeeper.dec.com in pub/plan/Life1.01.tar.Z.

[3] M. Andersson, C. Carlsson, O. Hagsand, and O. Stahl. *the Distributed Interactive Virtual Environment, Tutorials and Installation Guide*, Mar. 1993.

[4] J. Carlier and E. Pinson. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[5] J.-L. Cochard, editor. *WOz'95, International Workshop on Oz Programming*, CP 592, CH-1920 Martigny, 29 November – 1 December 1995. IDIAP.

[6] M. Henz, G. Smolka, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, Cambridge, MA, 1995.

[7] M. Henz and J. Würtz. Using Oz for college time tabling. In E.K.Burke and P.Ross, editors, *The Practice and Theory of Automated Time Tabling: The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Time Tabling, Edinburgh 1995*, Lecture Notes in Computer Science, vol. 1153, pages 162–177, 1996.

[8] C. Hewitt, P. Bishop, and R.Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[9] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Logic Programming Symposium*, pages 167–186, 1991.

[10] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[11] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In J.-L. Lassez, editor, *International Conference on Logic Programming*, pages 858–876. The MIT Press, Cambridge, MA, 1987.

[12] M. Müller, T. Müller, and P. Van Roy. Multi-paradigm programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 Dec. 1995. A Workshop in Association with ILPS'95.

[13] M. Müller, J. Niehren, and A. Podelski. Inclusion Constraints over Nonempty Sets of Trees. 1996. Submitted for publication.

[14] J. Muth and G. Thompson. *Industrial Scheduling.* Prentice-Hall, Englewood Cliffs, N.J., 1963.

[15] J. Niehren. Functional computation as concurrent computation. In 23^{th} *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 333–343. The ACM Press, 21–24Jan. 1996. longer version appeared as DFKI Research Report RR-95-14.

[16] J. Niehren and M. Müller. Constraints for Free in Concurrent Computation. In K. Kanchanasut and J.-J. Lévy, editors, *Asian Computing Science*

Conference, Lecture Notes in Computer Science, vol. 1023, pages 171–186, Pathumthani, Thailand, 11–13 Dec. 1995. Springer-Verlag, Berlin, Germany.

[17] F. C. N. Pereira and D. H. D. Warren. Parsing as Deduction. In *Proc. 21st Annual ACL Meeting*, pages 137–244, 1983.

[18] B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming (TPPP)*, Lecture Notes in Computer Science, Sendai, Japan, Nov. 1994. Springer-Verlag, Berlin, Germany.

[19] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[20] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In A. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer-Verlag.

[21] SFB 378 – Ressourcenadaptive kognitive Prozesse, 1996. Sonderforschungsbereich an der Universität des Saarlandes, Finanzierungsperiode 1996–1998.

[22] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, Sept. 1989.

[23] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *First International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 Sept. 1994. Springer-Verlag.

[24] G. Smolka. An Oz primer. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.

[25] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[26] G. Smolka, C. Schulte, and P. Van Roy. Perdio: Persistent and Distributed Programming in Oz – Project Proposal, 1995. Available at `http://ps-www.dfki.uni-sb.de/fbps/perdio.html`.

[27] G. Smolka and R. Treinen, editors. *DFKI Oz Documentation Series.* German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.