Seminar - Programmierung verteilter Systeme

Thema: Oz

Eine Multi-Paradigmen-Sprache für verteilte Anwendungen

Author: Markus Velt
Studienfach: Informatik
Betreuer: Bernd Braßel

30.06.2003

Überblick

1. Einleitung
2. Einblick Mozart-Oz
3. Darstellung der einzelnen Datentypen
4. Programmierung verteilter Anwendungen
5. Fehlerbehandlung
6. Beispiel: Eine Chat-Anwendung
7. Zusammenfassung
8. Verwendete Literatur

1 Einleitung

Mozart liegt die Programmiersprache Oz zugrunde, die viele Eigenschaften moderner Programmierkonzepte vereint. Dazu gehören funktionale und objektorientierte, aber auch Logik-, constraint-basierte und nebenläufige Programmierung. Da Mozart ursprünglich für die Implementierung von mobilen Agenten und zur Lösung von Problemen der künstlichen Intelligenz entwickelt wurde, enthält Oz eine konservative netzwerktransparente Erweiterung. Sprachsicherheit und plattformunabhängiger Bytecode, der von einer virtual machine ausgeführt wird, machen Mozart zu einer spezialisierten Umgebung für verteilte Systeme in heterogenen Netzwerken.
Diese Arbeit beschreibt die Programmierung von verteilten Systemen mit Mozart/Oz.

2 Einblick Mozart-Oz

Zum Verständnis der Programmierung von verteilten Systemen in Mozart ist es notwendig, die Sprache Oz ein wenig kennenzulernen und zu verstehen.

2.1 Syntax

Die Sprache Oz läßt sich auf eine sehr kleine, kompakte Kernsprache zurückführen mit folgender Syntax:

```
<Statement> ::= <Statement1> <Statement2>
              | X = f(l1:Y1 ... ln:Yn)
              | X = <number>
              | X = <atom>
              | X = <boolean>
              | {NewName X}
              | X = Y
              | local X1 ... Xn in S1 end
              | proc {X Y1 ... Yn} S1 end
              | {X Y1 ... Yn}
              | {NewCell Y X}
              | {Exchange X Y Z}
              | {Access X Y}
              | if B then S1 else S2 end
              | case E
                 of P1 then S1
                 [] P2 then S2
                 [] ...
                else S end
              | thread S1 end
              | try S1 catch X then S2 end
              | raise X end
```

Ein wichtiges Merkmal der Sprache Oz ist die Art und Weise, wie Variablen definiert werden und lokale Gültigkeit erhalten. Dies geschieht nicht nur bei Definition von Prozeduren, sondern auch explizit an beliebiger Stelle mit dem Ausdruck:

```
local X1 ... Xn in S end
```

Natürlich sind auch beliebig tiefe Verschachtelungen möglich:

```
Beispiel 1. Variablengueltigkeit

local X Y Z in  % noch sind X,Y,Z ungebundene Variablen
  local X Y in
    X = % inneres X
    Z = % aeusseres Z
  end
  Y = % aeusseres Y
end
```

Auf die Bedeutung der restlichen Sprachelemente soll hier nicht näher eingegangen werden, da sie entweder selbsterklärend ist oder im nächsten Abschnitt noch genauer besprochen wird.

2.2 Logische Variablen

Variablen sind in Oz sogenannte logische Variablen, bei denen ganz besondere Regeln gelten:

- Logische Variablen besitzen zwei Zustände: Sie können gebunden oder ungebunden sein.
- Bei Deklaration sind logische Variablen ungebunden, sie zeigen also auf kein anderes Sprachelement (also auch nicht auf Nummernwerte oder Strings).
- Sobald eine Zuweisung geschehen ist, kann keine weitere erfolgen, jeder Versuch endet in einer Exception. Diese Eigenschaft nennt man single-assignment.
- Eine logische Variable A kann auf eine andere B zeigen, ohne dass sich der Status der beiden ändert. Wird danach A oder B gebunden, ist implizit die andere Variable auch auf dasselbe Sprachelement gebunden.

Diese Eigenschaften von logischen Variablen machen Oz zu einer logischen Programmiersprache und bieten indirekt auch den Grundstein für Nebenläufigkeit. Um die tatsächliche Bedeutung zu verstehen, müssen wir uns also damit befassen, wie man Nebenläufigkeit im Sinne von Threads in Oz realisiert.

2.3 Threads

Threads werden in Oz explizit mit dem Ausdruck `thread S end` erzeugt. Deren Implementierung ist in Oz sehr effizient, so dass sich keine Performanceprobleme durch Nebenläufigkeit ergeben. Außerdem sorgt der Scheduler für eine faire Verteilung der Prozessorzeit.
Wie ist nun die Verbindung zwischen logischen Variablen und Nebenläufigkeit zu verstehen? Die Lösung liegt an einer Eigenschaft von Threads in Oz: Genaugenommen handelt es sich um sogenannte $data - flow\ threads$, was bedeutet, dass sie mit der sequentiellen Abarbeitung der nächsten Instruktionen nur dann fortfahren, wenn alle Werte, die dafür notwendig sind, verfügbar sind. Mit anderen Worten: Ein Thread blockiert, solange auf ungebundene Variablen zugegriffen werden müsste. Betrachten wir dazu folgendes Beispiel:

```
Beispiel 2. Synchronisation mit logischen Variablen

declare X0 X1 in   % entspricht einem local ohne end
thread
  local Y0 Y1 in
    Y0 = X0 + 1    % 1
    Y1 = X1 + Y0   % 2
  end
end
{Delay 100}
X0 = 1
{Delay 100}
X1 = 2
```

In diesem Beispiel blockiert der neu erstellte Thread circa 100 Millisekunden bei Zuweisung (1), da `X0` noch nicht gebunden ist. Danach erhält jedoch `X0` den Wert `1`, so dass die Addition durchgeführt und das Ergebnis `Y0` zugewiesen werden kann. Nun gelangt Oz bei der zweiten Instruktion (2) an und kann auch hier nicht fortfahren, weil zwar `Y0` einen Wert besitzt, aber nicht `X1`. Dies geschieht erst nach circa 100 Millisekunden im äußeren Thread.
Bereits an diesem kleinen Beispiel ist ersichtlich, dass logische Variablen in Verbindung mit Threads hervorragend dazu geeignet sind, typische Erzeuger-Verbraucher-Szenarios zu programmieren. In der Tat ist noch wesentlich mehr möglich, wie wir im nächsten Abschnitt noch genauer zeigen werden.

3 Darstellung der einzelnen Datentypen

3.1 Single-assignment Entities

$Single-assignment\ entities$ sind Sprachelemente beziehungsweise Daten eines Typs, denen nur genau einmal ein Wert zugewiesen werden kann. Ein $single-assignment\ entity$ haben wir bereits in Abschnitt Logische Variablen (2.2) kennengelernt.

3.1.1 Futures

Ein $future$ ist eine logische Variable, die nur lesbar ist. Jeder Versuch, einem $future$ einen Wert zuzuweisen, resultiert in einer Exception. Welchen Nutzen hat eine Variable mit solchen Eigenschaften? Mit einem $future$ kann man verhindern, daßVariablen vorzeitig von anderen Sites gebunden werden, wenn man eine Modul hat, welches verschiedene Dienste im Netz zur Verfügungstellt (vgl. auch Abschnitt Lokale Functors auf entfernten Sites starten (4.4)). Dies kann man erreichen, wenn man für jede fragliche logische Variable ein $future$ erstellt, das auf diese zeigt.
Das Verteilungsprotokoll von $futures$ entspricht dem von logischen Variablen.

3.1.2 Streams

Ein $stream$ ist ein asynchroner 1:n Kommunikationskanal. Tatsächlich handelt es sich hierbei um nichts anderes als eine Liste mit einer logischen Variable am Ende. $Streams$ sind der Grundstein für die sogenannte $stream-basierte$ Kommunikation, die vor

allem bei typischen Erzeuger-Verbraucher-Problemen verwendet wird. Betrachten wir dazu folgendes Beispiel:

`Beispiel 3. Ein typisches Erzeuger-Verbraucher-Problem`

```
declare
  fun {Producer N}
     if N > 0 then
        {Delay 1000}
        car|{Producer N-1}
     else nil end
  end
  proc {Consumer Xs}
     case Xs
     of car|Xr then
        {Browse 'a new car is ready...'}
        {Consumer Xr}
     [] nil then
        {Browse 'end of line'}
     end
  end
in
  {Consumer thread{Producer 10} end}
```

Die Funktion `Producer` liefert einen Stream zurück, der alle 1000 Millisekunden um ein Element erweitert wird. Die Prozedur `Consumer` erwartet nun diesen $stream$ als Argument und gibt jedes mal, wenn ein neues Element im $stream$ angekommen ist, eine Meldung aus. Das `case`-Konstrukt kann in Oz für das von funktionalen Programmiersprachen her bekannte $pattern\ matching$ verwendet werden. Man sollte allerdings beachten, dass hier ein $stream$ eigentlich nur $virtuell$ erweitert wird, da durch das Lesen eines neu ankommenden Elements die Liste zerstört wird. Ein $stream$, bei dem der Zustand der Liste gehalten wird, ist ein sogenannter $port$ (siehe Abschnitt namens 3.3.3 Ports).

3.2 Stateless Entities

$Stateless\ entities$ sind Sprachelemente, die keinen Zustand besitzen. Konkret bedeutet dies, dass sich deren Wert niemals ändert, was eine besonders effiziente Verteilung im Netz ermöglicht. Im wesentlichen wird jedes $stateless\ entity$ einfach kopiert (erfordert genau eine Nachricht), wann dies jedoch geschieht, ist abhängig vom Datentyp, was wir im folgenden noch sehen werden.

3.2.1 Records und Nummern

Ein $record$ besteht in Oz aus Paaren von $features$ und Feldern, wobei $features$ die Bezeichner für die Felder sind. Beispielsweise besteht das $record$

```
tree(key:K left:L right:R)
```

aus den Feldern `K`, `L` und `R`, welche logische Variablen sind. Die Werte des $records$ können jedoch nicht über die Namen der logischen Variablen erreicht werden, sondern über die $features$ `key`, `left` und `right`.

Beispiel 4. Beispiel zur Benutzung von Records

```
declare T Key Left Right in
  T     = tree(key:K left:L right:R)
  K     = 43
  L     = nil
  R     = nil
  {Browse T.key}
```

$Records$ und $Nummern$, sowie Listen und Strings, die intern auch nur $records$ sind, sind sehr grundlegende und kleine Datenstrukturen, so dass Mozart sie einfach $eager$ verschickt. Sie werden also unmittelbar nach Definition an alle Proxies verschickt, selbst wenn dort bereits eine Kopie vorhanden ist. Dies geschieht aus Effizenzgründen: Die Einführung eines globalen Identifikators wäre viel zu aufwendig und kontraproduktiv. Nicht vergessen darf jedoch, dass ein $record$ selbstverständlich aus anderen Datentypen bestehen kann und deren verteilte Semantik sich deswegen nicht ändert. Ein $record$ ist somit vergleichbar mit einem Container, der Daten anderen Typs enthält. Im Netz wird nur die Inventarliste des Containers durch einfaches kopieren verteilt, der Inhalt hingegen muss weiterhin mit der dazu passenden Methode verschickt werden. Siehe dazu auch den Abschnitt Objekte (3.3.2).

3.2.2 Prozeduren, Funktionen, Klassen, Functors, Chunks, Atome und Namen

Alle im Titel dieses Kapitels angeführten Sprachelemente von Oz verbindet folgende Eigenschaften: Sie werden alle $eager$ im Netz verteilt, jedoch immer nur genau einmal, da sie einen global eindeutigen Identifikator besitzen. Konkret bedeutet dies, dass beispielsweise zwei Prozedurdefinitionen mit identischem Quellcode zwei unterschiedliche Identifikatoren besitzen und somit von beiden jeweils eine Kopie verschickt wird. Dies gilt für alle angesprochenen Sprachelemente außer den Atomen. Atome sind symbolische Identifikatoren, die durch einen alphanummerischen Namen unterschieden werden (siehe Beispiel 3. in Abschnitt namens 3.1.2 Streams das Atom `car`), sie sind also im gewissen Sinne Literale vergleichbar mit Nummern.
Der Vollständigkeit halber sollen an dieser Stelle noch einige Begriffe definiert werden, die jedoch für das Thema dieser Arbeit nicht weiter relevant sind: $chunks$ sind $records$, die ihre Struktur nach außen und gegen $Fremdeinfluss$ verbergen. $names$ können nur durch den Befehl `{NewName X}` erzeugt werden und sind garantiert weltweit eindeutig. Sie sind der Grundstein für die Sicherheit von Oz-Programmen. $functors$ werden noch in dem Abschnitt namens 4.4 Lokale Functors auf entfernten Sites starten erklärt.

3.3 Statefull Entities

3.3.1 Cells

Eine $cell$ ist ein $statefull\ entity$ und kann somit ihren Zustand, also quasi ihren $Inhalt$, beliebig oft ändern. Mit anderen Worten ist eine $cell$ mit Variablen aus anderen bekannten Programmiersprachen wie `C` oder `C++` vergleichbar. Allerdings geschieht der Zugriff auf diese nur über spezielle Anweisungen und nicht über die Zuweisungsoperatoren von logischen Variablen.
Die Operationen auf Variablen vom Typ $cell$ sind im einzelnen:

- `{NewCell X C}` erzeugt eine $cell$, wobei `X` der Initialisierungswert und `C` eine logische Variable ist, die auf die neue $cell$ zeigen soll.
- `{Exchange C X Y}` weist der $cell$ `C` den neuen Wert `Y` zu und speichert den alten Wert in `X`, all dies geschieht atomar.
- `{Assign C Y}` weist der $cell$ `C` den neuen Wert `Y` zu.
- `{Access C X}` liefert den Inhalt von `C` in `X` zurück.
- `{IsCell C}` liefert `true`, wenn `C` vom Typ $cell$ ist.

Dadurch, dass selbst für einfache Operationen wie $access$ eine spezielle Operation vorgesehen ist, erleichtert sich Oz eine aufwendige Typisierung und kann somit zielsicher die richtige Semantik anwenden. Dies gilt natürlich insbesondere für die verteilte Semantik. Der Hauptgrund für diese außergewöhnlich anmutende Entscheidung der Oz-Entwickler liegt jedoch am Ziel $network - aware$: Nur wenn es eine wohldefinierte Menge an Operationen an einem Sprachelement gibt und diese nicht vor dem Programmierer verborgen werden, kann dieser das Verhalten im Netz genau steuern.

3.3.2 Objekte

Es gelten in Oz die üblichen Definitionen objektorientierter Programmiersprachen: Eine Klasse besteht in Oz aus Features, Attributen und Methoden. Der Unterschied zwischen Attribut und Feature läuft im wesentlich darauf hinaus, dass Features immer $public$ und Attribute stets $private$ sind. Ein Objekt ist eine Instanz einer Klasse.

- Mobile Objekte
  Betrachtet man die Natur der Bestandteile einer Klasse, so stößt man schnell auf zwei unterschiedliche Typen: Während Attribute $statefull\ entities$ sind, gehören die Features und Methoden eindeutig zu den $stateless\ entities$. Aufgrund dieser Unterschiede ist es naheliegend, dass es keine spezifische verteilte Semantik für Objekte geben kann, vielmehr verhält sich jede einzelne Komponente im Netz so wie es notwendig ist. Auf Implementierungsebene ist ein Objekt aus folgenden Bestandteilen aufbaut:
  - $Objekt - Record$: Dieser Record enthält alle Features eines Objektes. Er wird lazy im Netz verteilt, wird also nur verschickt, wenn die entfernte Site auf irgendein Feature des Records zugreifen will.
  - $interne\ Klasse$: Jedes Objekt führt die Definition ihrer zugrundeliegenden Klasse und somit auch aller Methoden mit. Ebenso wie der Objekt-Record wird die Klasse nur lazy verschickt.
  - $content - edge$: Die Attribute werden in Variablen vom Typ Cell gespeichert, welche durch einen sogenannten state record zusammengefasst werden. Die verteilte Semantik entspricht somit der einer Cell. Die content-edge eines Objektes zeigt immer auf diesen Record.
- Stationäre Objekte
  Im vorherigen Kapitel wurde das Verteilungsprotokoll von sogenannten mobilen Objekten dargestellt. Dies wird eingesetzt, falls für die Erzeugung eines Objektes die Operation New verwendet wurde. Im Gegensatz dazu stehen stationäre

Objekte, die nicht im Netz verteilt werden, sondern auf der Site verbleiben, auf der sie erzeugt wurden (die sogenannte owner site). Sie sind also Grundstein einer jeden Client-Server-Anwendungen. Man unterscheidet hierbei zwischen zwei unterschiedlichen Typen:

- $stationary\ objects$: Mit NewStat erzeugte Objekte verbleiben immer auf ihrer owner site. Falls eine Methode eines solchen Objektes aufgerufen wird, wird auf der entfernten Site ein neuer Thread erzeugt und darin die Methode ausgeführt. Hierfür werden nur zwei Nachrichten benötigt - eine für den Start der Methode und eine für die Synchronisation mit dem Aufrufer (bis zur Beendigung der Methode wird der aufrufende Thread angehalten). Exceptions werden beim Aufrufer ausgelöst.
- $sequential\ asynchronous\ stationary\ object$: Bei diesen mit NewSASO erzeugbaren Objekten geschieht keine Synchronisation mit dem Aufrufer. Mozart benötigt also nur genau eine Nachricht, da der aufrufende Thread nicht blockiert wird und somit auch keine Exceptions erhält. Außerdem wird für die Methodenaufrufe kein neuer Thread auf Serverseite erzeugt.

3.3.3 Ports

Ein $port$ entspricht einem asynchronen 1:n Kommunikationskanal. Im Gegensatz zu einem Stream speichert er jedoch die ankommenden Werte. Dazu hält er auf der $owner\ site$ einen Stream. $ports$ sind sogenannte $sited\ entities$, bleiben also auf der Site, die sie erstellt haben.
Während Streams lediglich eine geschickte Ausnutzung von Listen sind, stellt Mozart für $ports$ eigene Operatoren zur Verfügung, hier sind die wichtigsten:

- `{Port.New S P}` erstellt einen neuen $port$ `P`, wobei `S` eine logische Variable ist, die den Stream enthält.
- `{Port.Send P M}` sendet `M` an den $port$ `P`.

3.3.4 Thread-reentrante Locks

In nebenläufigen Programmen kommt es zwangsläufig zu Situationen, in denen auf bestimmte Programmbereiche von Threads nur exklusiver Zugriff möglich sein sollte. Oz unterstützt den Programmierer bei der Bewältigung solcher schwierigen Programmieraufgaben durch sogenannte $thread-reentrante\ Locks$, die sehr einfach zu handhaben sind: Nachdem man mit `{NewLock L}` einen neuen Lock erzeugt hat, muss man nur noch den kritischen Bereich mit der Anweisung `lock L then ... end` klammern. Oz sorgt dann dafür, dass nur jeweils ein Thread (dieser aber beliebig oft) auf diese Region Zugriff hat - sogar netzwerkweit. Die verteilte Semantik von Locks entspricht natürlich der von Variablen vom Typ Cell.
Folgendes Beispiel verdeutlicht die Anwendung von Locks:

```
Beispiel 5. Wie benutzt man einen Lock?

local X Y in
  {NewLock L}
  {NewCell hallo C}
```

```
  thread
    {Delay 1000}
lock L then {Assign C hello} end
  end

  thread
    lock L then {Assign C bye} {Delay 2000} end
  end

  % C hat 2000 Millisekunden lang den Wert 'hello' und danach den Wert 'bye'
end
```

3.4 Sited Entities

Sited entities sind Sprachelemente, die auf der *owner site* verbleiben und auch nur dort eingesetzt werden können. Dies sind in Oz im wesentliche Module. Module fassen mehrere Prozeduren und Variablen zu einer Einheit zusammen. *Functors* sind eine Spezialform davon. Es sind Module, bei denen in der Definition angegeben wird, welche Ressourcen benötigt werden (mit anderen Worten: welche anderen Module) und welche Prozeduren und Variablen nach aussen als Schnittstelle zur Verfügung stehen. Ein Beispiel dazu gibt es in dem Abschnitt namens 4.4 Lokale Functors auf entfernten Sites starten.

4 Programmierung verteilter Anwendungen

Nachdem wir nun alle Datentypen von Oz behandelt und auch deren verteilte Semantik mehr oder weniger detailiert besprochen haben, wissen wir trotzdem noch nicht, wie man eigentlich zwei Sites so miteinander verbindet, dass auch tatsächlich eine verteilte Anwendung entsteht. In diesem Kapitel wollen wir uns mit dieser Thematik befassen.

4.1 Tickets

In verteilten Anwendungen ist wohl die Notwendigkeit eines global eindeutigen Identifikators für jedes Sprachelement relativ einleuchtend. Natürlich gäbe es hierfür mehrere Möglichkeiten: Beispielsweise könnte man die IP-Adresse plus Variablennamen verwenden. Allerdings würde dies die Verwendung von zwei Mozart-Engines auf einem Rechner ausschließen und noch dazu keinerlei Netzwerktransparenz bieten. Die Lösung von Mozart ist damit zwar verwandt, umgeht jedoch die genannten Nachteile: Mozart bildet einfach aus vielen verschiedenen Eingabedaten einen global eindeutigen String, der Ticket genannt wird. Wie dieser entsteht, muss man nicht wissen, da er in der Praxis auf irgendeine Art und Weise publik gemacht wird, so dass sich Oz-Programme die Tickets selbst holen können (beispielsweise via EMail oder über Pickles, siehe dazu das nächste Kapitel).
Um also ein Sprachelement `X` anderen Sites zur Verfügung zu stellen, muss man zuerst ein Ticket T für dieses erstellen. Dies geschieht mit `{Connection.offer X T}`, falls das Ticket nur einmal benutzbar sein soll, oder mit
`{Connection.offerUnlimited X T}`, falls das Ticket unbegrenzt gelten soll. Um nun dieses Sprachelement auf einer anderen Site zu benutzen, muss sich die entfernte Site das Ticket besorgen und dann `{Connection.take T X}` aufrufen.

```
Beispiel 6. Tickets erstellen

declare Stream Tkt in
{Connection.offerUnlimited Stream Tkt}
{Show Tkt}
```

Das erzeugte Ticket `Tkt` hat die folgende Gestalt:

```
   'x-ozticket://127.0.0.1:9000:8ctSqy:Aey/y:w:x:m:Qhnybv'
```

4.2 Zugriff auf entfernte Datenstrukturen via URL: Pickles

Wie bereits im vorherigen Kapitel angesprochen, können Tickets sehr einfach gespeichert und geladen werden. Die einfachste Lösung bieten die sogenannten Pickles. Mit ihrer Hilfe kann ein Oz-Programm beliebige *stateless entities* in einer Datei speichern und sie wieder daraus laden. Da auch Tickets *stateless* sind, bilden Pickles eine einfache Methode, um Sprachelemente auf anderen Sites zu referenzieren.
Folgende Anweisungen stehen für Pickles zur Verfügung: `{Pickle.save X F}` speichert das Sprachelement `X` in der Datei `F`, während `{Pickle.load F X}` ein solches aus der angegebenen Datei lädt. In der Praxis wird ein Pickle oft in dem lokalen Verzeichnis gespeichert, das über die eigene Homepage via HTTP erreichbar ist - geladen kann es dann über eine URL wie beispielsweise http://www.informatik.uni-kiel.de/~mve/pickle_file werden. Betrachten wir dazu ein Beispiel:

```
Beispiel 7. Ein Hello-Server

declare Str Prt Srv in
{NewPort Str Prt}
thread
   {ForAll Str proc {$ S} S='Hello World' end}
end
proc {Srv X}
   {Send Prt X}
end
{Pickle.save {Connection.offerUnlimited Srv}
  "/home/mve/public_html/myhw"}

Beispiel 8. Ein Hello-Client

declare Srv in
Srv={Connection.take {Pickle.load
  "http://www.informatik.uni-kiel.de/~mve/myhw"}}

local X in
   {Srv X}
   {Browse X}
end
```

4.3 Stateless Entities als Pickles

Neben Tickets können auch einfache *stateless entities* als Pickle gespeichert werden. Sichert man beispielsweise eine Prozedur in einer Datei, kann eine entfernte Site deren

Quellcode laden und diese dann lokal ausführen. Man beachte, dass hier lediglich der Prozedur-Code zur Verfügung gestellt. Ausgeführt wird er jeweils auf der anfordernden Site.

Beispiel 9. Eine Prozedur wird als Pickle gespeichert

```
declare
fun {Fact N}
   if N=<1 then 1 else N*{Fact N-1} end
end
{Pickle.save Fact "~/mve/public_html/fact"}
```

Beispiel 10. Eine Prozedur wird als Pickle geholt

```
declare
Fact={Pickle.load "http://www.informatik.uni-kiel.de/~mve/fact"}
{Browse {Fact 12}}
```

4.4 Lokale Functors auf entfernten Sites starten

Mit der Hilfe von Pickles und Tickets können wir nur die Ressourcen von entfernten Sites nutzen, die der Programmierer auf der anderen Seite auch als solche vorgesehen hat. Oft möchte man jedoch einfach die Prozessorzeit eines anderen Rechners nutzen, ohne auf der entfernten Site irgendeine Server-Komponente implementiert zu haben. Auch dies ist in Mozart möglich, nämlich mit dem Aufruf von `{New Remote.manager init(host:"rechner.domain.de")}`. Hiermit erstellen wir einen neuen, für uns exklusiv nutzbaren, Mozart-Prozess auf dem entfernten Rechner. Dieser kann nun jeden beliebigen Functor ausführen, er muss nur durch die Methode `apply` dem Prozess übergeben werden. Das folgende Beispiel verdeutlicht die Anwendung:

Beispiel 10. Einen Functor auf einer entfernten Site ausfuehren

```
declare
fun {Fact N}
   if N=<1 then 1 else N*{Fact N-1} end
end

declare R F M in
  % neuen Mozart-Prozess auf einem entfernten Rechner erzeugen
  R={New Remote.manager init(host:"medoc.informatik.uni-kiel.de")}

  % Functor definieren (x ist das Feature der Variable X)
  F=functor export x:X define X={Fact 30} end

  % Functor-Quellcode an medoc uebergeben und
  % Referenz fuer entfernten Aufruf in M speichern
  {R apply(F M)}

  % Functor auf mdeoc ausfuehren und
  % Ergebnis ueber Feature x ausgeben
```

```
{Browse M.x}
```

5 Fehlerbehandlung

Fehler können in jedem Programm auftreten, fast zwangsläufig geschieht dies jedoch in verteilten Anwendungen - hier spielen einfach viel zu viele Fehlerquellen eine Rolle. Die typischen Fehler, die in einer verteilten Anwendungen auftreten können, sind häufig nur partieller Natur. Da oft nur ein Teil des gesamten Systems ausfällt, kann der Rest meist ungehindert fortfahren. Diese Eigenschaft kann sich der Programmierer zu Nutze machen und absichtlich Redundanz erzeugen, so dass im Falle eines Fehlers ein Ersatz einspringen kann. So positiv dieses Verfahren erscheint, so schwierig ist es zu realisieren, da die Komplexität einer Anwendung leider mit dem Grad der Verteilung zunimmt. Oz versucht hierbei jedoch helfend zur Hand zu gehen, indem Fehler feinkörnig unterschieden werden.

5.1 Permanente und temporäre Fehler

Betrachten wir zuerst die sogenannten lokalen Fehler. Sie werden so genannt, da sie sich unmittelbar auf die aktuelle Site auswirken. Mozart geht bei jedem Zugriff auf ein Sprachelement von drei Szenarien aus:

- Es gibt keine Probleme. Status: OK
- Das Sprachelement kann vorübergehend nicht angesprochen werden, da die entfernte Site momentan aufgrund eines Netzwerkfehlers nicht erreicht werden kann. Status: tempFail
- Das Sprachelement ist permanent nicht erreichbar, da die entfernte Mozart-Engine zusammengebrochen ist, beispielsweise aufgrund eines Server-Crashs. Status: permFail

5.2 Fehler auf entfernten Sites

Während sich die im vorherigen Kapitel vorgestellten Fehlerklassen direkt auf die eigene Site auswirken und eine korrekte Ausführung beeinträchtigen, kann es oft auch interessant sein, auf Fehler auf entfernten Sites hingewiesen zu werden, die sich vorerst nicht unmittelbar bemerkbar machen. Mozart unterscheidet hier zwischen folgenden Fehlertypen:

- remoteProblem(permSome): Mindestens ein Proxy, der das Sprachelement referenziert, ist permanent nicht erreichbar.
- remoteProblem(permAll): Alle referenzierenden Sites können nicht auf das Sprachelement zugreifen.
- remoteProblem(tempSome): Mindestens eine Site, die das Sprachelement referenziert, kann momentan nicht erreicht werden
- remoteProblem(tempAll): Alle Sites, die das Sprachelement referenzieren, sind temporär unerreichbar

Es ist wohl einsichtig, dass diese hier beschriebenen Fehlerklassen nicht immer zuverlässig gemeldet werden können. Beispielsweise kann es vorkommen, dass ein Proxy, von dem der Manager noch gar nichts weiss, ausfällt. Da es hier keine Bezugspunkte zu der eigenen Site gibt, kann der Fehler nicht erkannt werden.

5.3 Fehler abfangen

Wir haben bisher von Fehlerklassen gesprochen, dabei jedoch völlig ausser acht gelassen, wie diese einem Oz-Programm gemeldet werden. Dafür existieren drei Methoden:

1. Die einfachste Methode benutzt Exceptions. Diese haben den Vorteil, dass sie einfach abzufangen sind, erschweren aber Recovery-Versuche und werden nur dann ausgelöst, wenn der Fehler bereits das Fortfahren des aktuellen Threads unterbindet.

2. Einen ganz anderen Ansatz verfolgen sogenannte Handler: Dies sind Prozeduren (genauer gesagt Funktionen), die die fehlerhafte Anweisung im Fehlerfall ersetzen und somit Alternativen ausloten können. Das Programm kann somit normal fortfahren. Nehmen wir zum Beispiel den Hello-Client aus dem Abschnitt namens 4.1 Grundbaustein für Kommunikation: Tickets. Falls hier der Hello-Server ausfällt, kann ein auf die Prozedur Srv angesetzter Handler eigenständig mit einem anderen Backup-Server Verbindung aufnehmen und Srv mit diesem assoziieren.

3. Die Methoden 1 und 2 reagieren synchron auf Fehler, treten also erst in Aktion, wenn sich der Fehler tatsächlich auf das aktuelle Programm auswirkt. Oft kann es jedoch sinnvoll sein, bereits unmittelbar nach Auftreten eines Fehlers davon zu wissen und entsprechend reagieren zu können. Dann könnte beispielsweise beim Hello-Client ein Alternativ-Server gefunden werden, noch bevor der Fehler überhaupt Relevanz bekommen würde. Die Fehler-Abfang-Prozeduren, die in diesem Sinne asynchron aufgerufen werden, nennt Mozart Watcher.

An den kurzen Ausführungen dieses Kapitels kann man bereits erahnen, wie aufwändig eine ausführliche Fehlerbehandlung ist, besonders, wenn man alle Sprachelemente betrachten müsste. Damit der Programmierer mehr Überblick behalten kann und weniger Schreibarbeit hat, muss die Fehlerbehandlung explizit für jedes gewünschte Sprachelement aktiviert werden.

6 Beispiel: Eine Chat-Anwendung

Ein Chat-System ermöglicht Teilnehmern mit verschiedensten Systemen in Echtzeit eine textbasierende Diskussion zu führen. Weiter Teilnehmer können jederzeit sich dem Chat-Forum anschließen oder bisherige können es einfach verlassen. Deshalb muss ein solches Chat-System sehr robust Netzwerkfehlern, Rechner- oder Prozeßabstürze entgegenwirken.
Diese Beispiel zeigt uns, wie einfach man so eine verteilte Anwendung robust realisieren kann.

6.1 Der Chat-Server

Die Server-Anwendung erzeugt einen `NewsPort`, der mit Hilfe eines `Tickets` erreicht werden kann. Das Ticket selbst wird in einer Datei gspeichert, welche durch die Clients über eine URL geladen werden kann. Wenn eine Client sich an den Server binden will, wird der `NewsPort` nicht nur für die gesendeten Nachrichten des einen Clients benutzt. Vielmehr ergibt sich ein Strom aus Nachrichten aller Clients zu einem `NewsPort`, der die Nachrichten an alle Clients verteilt, wo sie angeszeigt werden.

```
Beispiel-Anwendung: Der Chat-Server
% file: chat-server.oz

functor
import
   Application(getCmdArgs) Connection(gate) Pickle(save)
define
   Args = {Application.getCmdArgs
           record(ticketfile(single type:string optional:false))}
   NewsPort
   local Ticket in
      {New Connection.gate init(NewsPort Ticket) _}
      {Pickle.save Ticket Args.ticketfile}
   end
   {List.forAllTail {Port.new $ NewsPort}
    proc {$ H|T}
       case H of connect(Messages) then Messages=T else skip end
    end}
end
```

Der Quellcode der Server-Anwendung wird in der Datei `chat-server.oz` gespeichert und kann mit folgender Anweisung kompiliert werden.

```
ozc -x chat-server.oz
```

Gestartet wird der Server durch die folgende Anweisung, wobei die Option $File$ den Zielort des Tickets darstellt. (z. B. http://www.informatik.uni-kiel.de/~mve/chatApp).

```
chat-server --ticketfile File
```

6.2 Der Chat-Client

Der Chat-Client besteht aus zwei Teilen, einem $user\ interface\ agent$ und einem $messagestreamprocessor$.

```
Beispiel-Anwendung: Der Chat-Client
% file: chat-client.oz

functor
import
   Application(getCmdArgs) Pickle(load) Connection(take)
   Viewer(chatWindow) at 'chat-gui.ozf'
define
   Args = {Application.getCmdArgs
```

```
            record(url(single type:string optional:false)
                   name(single type:string optional:false)
                  )}
   NewsPort={Connection.take {Pickle.load Args.url}}
   SelfPort
   % 1. obtain and process message stream
   ...
   % 2. create user interface agent
   ...
   % 3. process message stream
   ...
end
```

Der Chat-Client erhält die Nachrichten vom Server, indem er die Anweisung `connect(...)` sendet. Jetzt wird jede Nachricht von Nachrichtenstrom `NewsPort` in den internen Strom `SelfPort` weitergeleitet. Das $user\ interface$ wird ebenfalls alle Nachrichten an den internen Port schicken.

```
% 1. obtain and process message stream
   thread
      {ForAll {Port.send NewsPort connect($)}
       proc {$ Msg} {Port.send SelfPort Msg} end}
   end
```

Wenn eine $user\ interface$ erzeugt wird, erhält es den internen `SelfPort`. In dieser einfachen Implementierung sendet das $user\ interface$ Nachrichten in Form `say(`$String$`)`, um diese Nachricht später vom globalen Nachrichtenstrom `NewsPort` zu erhalten.

```
% 2. create user interface agent
   Chat = {New Viewer.chatWindow init(SelfPort)}
```

Hier werden nun alle Nachrichten im internen Nachrichtenstrom `SelfPort` verarbeitet. Eine Nachricht der Form `msg(FROM TEXT)` ist schon formatiert und wird in des Chat-GUI angezeigt. Eine Nachricht der Form `say(TEXT)` wird in eine Nachricht `msg(NAME TEXT)` transformiert, wobei `NAME` den Namen des $users$ repräsentiert. Der Inhalt von `TEXT` wird in eine kompakten $byte\ string$ konvertiert, um eine effizientere übermittlung zu ermöglichen. Anschließend wird die transformierte Nachricht in der globalen Nachrichtenstrom `NewsPort` geschickt.

```
% 3. process message stream
   NAME = Args.name
   {ForAll {Port.new $ SelfPort}
    proc {$ Msg}
       case Msg of msg(FROM TEXT) then
          {Chat show(FROM#':\t'#TEXT)}
       elseof say(TEXT) then
          {Port.send NewsPort msg(NAME {ByteString.make TEXT})}
       else skip end
    end}
```

Der Quellcode der Server-Anwendung wird in der Datei `chat-client.oz` gespeichert und kann mit folgender Anweisung kompiliert werden.

```
ozc -x chat-client.oz
```

Gestartet wird der Server durch die folgende Anweisung, wobei die Option `MyName` den Namen des Chat-Teilnehmers repräsentiert und `URL` die URL des Tickets darstellt, das durch den Chat-Server erzeugt wurde (z. B. http://www.informatik.uni-kiel.de/~mve/chatApp).

```
chat-server --name MyName --url URL
```

6.3 Das graphische Oberfläche

Das $graphical\ user\ interface$ beinhaltet im wesentlichen eine Tk-Anwendung, auf die im Rahmen des Seminars nicht weiter eingegangen werden soll. Der wichtigste Teil diese Anwendung ist das `@entry`-Widget. Es wartet auf das Event $< KeyPress - Return >$, welches dann die `post`-Methode auslöst. Ist das der Fall, sendet die `post`-Methode eine `say(`$Text$`)` Nachricht an den internen Port `SelfPort`, wobei $Text$ den eingegebenen Text des `@entry`-Widget enthält. Wie zuvor in de Client-Anwendung schon beschrieben wurde, wird dort des Text konvertiert und mit der Namen des $users$ in die Form `msg(FROM TEXT)` transformatiert. Die `show(TEXT)`-Methode wird vom Chat-Client aufgerufen, wenn dort eine Nachricht der Form `msg(FROM TEXT)` eintroffen ist. Diese Methode schreibt den überlieferten `TEXT` in eine Canvas.

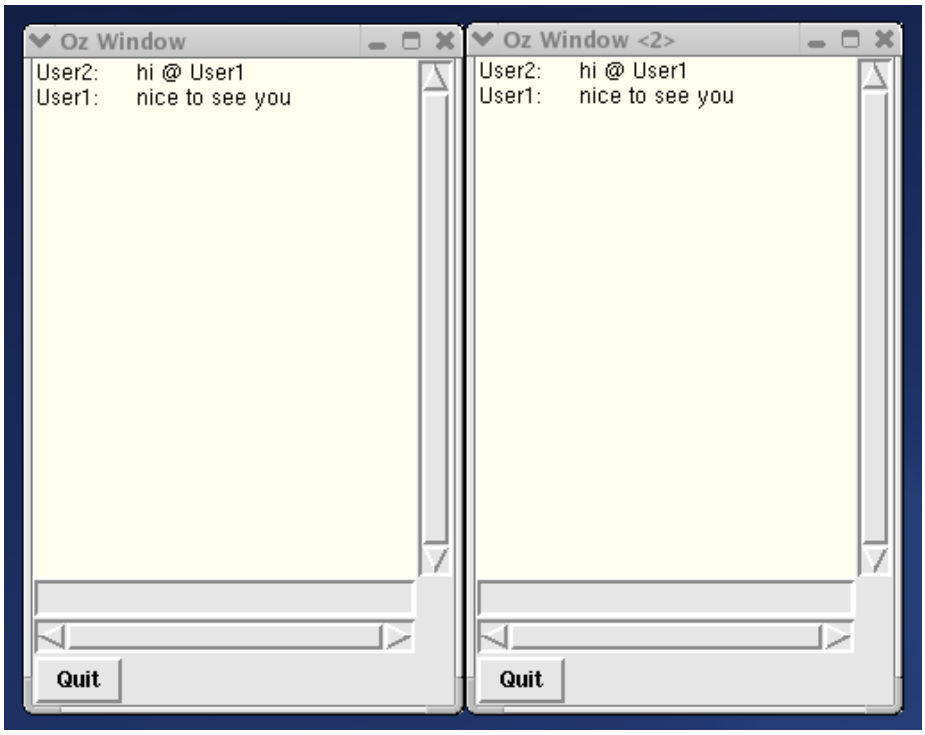
```
Beispiel-Anwendung: Die Chat-GUI
% file: chat-gui.oz

functor
import
   Tk Application(exit:Exit)
export
   ChatWindow
define
   class ChatWindow from Tk.toplevel
      attr canvas y:0 vscroll hscroll tag:0 selfPort entry quit
      meth init(SelfPort)
         Tk.toplevel,tkInit
         selfPort <- SelfPort
         canvas <- {New Tk.canvas
                    tkInit(parent:self bg:ivory width:400 height:300)}
         vscroll <- {New Tk.scrollbar tkInit(parent:self orient:v)}
         hscroll <- {New Tk.scrollbar tkInit(parent:self orient:h)}
         entry   <- {New Tk.entry     tkInit(parent:self)}
         quit    <- {New Tk.button    tkInit(parent:self text:'Quit'
                                             action:proc{$} {Exit 0} end)}
         {Tk.addYScrollbar @canvas @vscroll}
         {Tk.addXScrollbar @canvas @hscroll}
         {@canvas tk(configure scrollregion:q(0 0 200 0))}
         {@entry tkBind(event:'<KeyPress-Return>'
                        action:proc {$} {self post} end)}
         {Tk.batch [grid(row:0 column:0 @canvas sticky:ns)
                    grid(row:1 column:0 @entry sticky:ew)
                    grid(row:0 column:1 @vscroll sticky:ns)
```

```
                  grid(row:2 column:0 @hscroll sticky:ew)
                  grid(row:3 column:0 @quit sticky:w)
                  grid(columnconfigure self 0 weight:1)
                  grid(rowconfigure self 0 weight:1)]}
      end
      meth show(TEXT)
         {@canvas tk(create text 0 @y text:TEXT anchor:nw tags:@tag)}
         local
            [X1 Y1 X2 Y2] = {@canvas tkReturnListInt(bbox all $)}
         in
            y<-Y2
            {@canvas tk(configure scrollregion:q(X1 Y1 X2 Y2))}
         end
      end
      meth post
         {Port.send @selfPort say({@entry tkReturn(get $)})}
         {@entry tk(delete 0 'end')}
      end
   end
end
```

Der Quellcode der Server-Anwendung wird in der Datei `chat-gui.oz` gespeichert und kann mit folgender Anweisung kompiliert werden.

```
ozc -c chat-client.oz
```

Die vom Oz-Compiler erzeugte Datei heißt `chat-gui.ozf`. Da der Chat-Client diese Datei benutzt, muss diese Datei im für den Client in diesem Beispiellokal zu Verfügung stehen.



Screenshot von zwei Chat-Clients

7 Zusammenfassung

Auch wenn diese Arbeit nur einen kleinen Einblick in die Fähigkeiten von Mozart-Oz geben kann, zeigt sie dennoch bereits das Potenzial, das hinter dieser kompakten und mächtigen Sprache steckt, besonders im Bereich der verteilten Programmierung. Mozart-Oz stellt dem Entwickler eine große Bandbreite von Features und Hilfsmitteln zur Verfügung und ermöglicht somit einfach robuste und komplexe Applikationen zu entwickeln. Es vereint eine Vielzahl von Möglichkeiten anderer Programmiersprachen in einem zusammenhängenden Design, da es die Möglichkeiten objekt-orrientierter, funktionaler und logischer Programmierung unterstützt.

8 Verwendete Literatur

1. Mozart-Oz Documentation, http://www.mozart-oz.org/documentation/index.html
2. Gert Smolka, $The\ Oz\ Programming\ Model$, Programming Systems Lab, German Research Center for Artifiacial Intelligence (DFKI), Germany, July 1995