Nebenläufige Programmierung: Alice, Java und CML

Proseminar

angefertigt von

Mina Nikolova

Universität des Saarlandes
Fachrichtung Informatik
Lehrstuhl Programmiersysteme
Prof. Gert Smolka

06.04.2004

Inhaltsverzeichnis

1 Einführung

Die neue Rechnergeneration, die sich ständig weiterentwickelt, soll immer schneller, besser und zuverlässiger arbeiten. Sie wird aber die immer höheren Anforderungen an Leistungen nicht erfüllen können, wenn sie die Vorteile nicht genutzt hätte, die das parallele Abarbeitung von Aufgaben bietet. Deshalb ist Nebenläufigkeit von grosser Bedeutung in herkommlichen Programmiersystemen.

Ein nebenläufiges Programm definiert mehrere Folgen von Aktionen, die gleichzeitig ausgeführt werden. Es gibt zwei Möglichkeiten ein nebenläufiges Programm auszuführen, *Multi-Programmierung* und *Multiprozessor- Programmierug.* Bei *Multi-Programmierung* kann ein physikalischer Prozessor mehrere Prozesse parallel ausführen. Ein Prozess ist eine sequentielle Ausführung von Aktionen. Jeder Prozess wird von einem Prozessor sequentiell bearbeitet. Unter *Multiprozessor-Programmierung* versteht man mehrere Prozessoren, die gemeinsamen Speicher teilen oder in einem Netzwerk verbunden sind. Dabei können Prozesse, die auf den verschiedenen Prozessoren ausgeführt werden, miteinander kommunizieren.

2 Anwendungen der Nebenläufigkeit

Nebenläufigkeit ist ein wichtiges Element natürlicher und künstlicher Systeme. Diese fundamentale Idee der Informatik ist in vielen Bereichen der Wissenschaft zu erkennen: Petri-Netze (Theoret. Informatik), Betriebssysteme und Programmiersprachen (Prakt. Informatik), Rechnerarchitektur und Prozeßdatenverarbeitung (Techn. Informatik), Informationssysteme (Angew. Informatik), Projektunterricht (Didaktik der Informatik). Nebenläufigkeit erlaubt einem Programm zu arbeiten, während ein anderes Programm auf Eingabe wartet, was ein besseres Nutzen des Prozesors nach sich zieht. Ein weiteres Anwendungsbeispiel wäre ein WWW-Browser. Er kann z.B. gleichzeitig eine neue Seite laden, einen Audio-Clip abspielen und die bereits empfangenen Daten anzeigen. Mit *Multiprozessor-Programmierung* können Prozesse auf verschiedenen Prozessoren zusammen arbeiten und Problemme gleichzeitig lösen. Die Zuverlässigkeit der Kommunikation im Netzwerk verbessert sich dadurch, dass ein Prozessor weiterarbeiten darf, falls ein anderer Prozessor abstürtzt. Nebenläufigkeit findet auch im alltäglichen Leben Ansatz und wird zum Alltagsprinzip.

3 Grundlegende Begriffe

Eine sequentielle Ausführung von Aktionen kann als Prozess, Thread oder Task bezeichnet werden. Oft bezieht sich der Begriff Prozess auf Betriebssystemprozesse, die einen eigenen Adressraum besitzen. Im Gegensatz dazu gehört jeder Thread zu genau einem Prozess, wobei alle Threads eines Prozesses denselben Adressraum teilen. In Sprachen, die das nebenläufige Programmierung unterstützen, wird eher das Konzept des Threads benutzt.

Unter Betriebsmitteln (Resourcen) versteht man Komponenten des Rechnersystems, die von den einzelnen Prozessen/Threads während ihres Ablaufs benötigt werden.

Im Zusammenhang mit Nebenläufigkeit sollen einige Begriffe eingeführt werden:

- **Kommunikation**

- **Koordination**
- **Atomarität**
- **Kritischer Abschnitt (kA)**

In den folgenden Abschnitten werden die einzelnen Begriffe detailiert betrachtet.

3.1 Kommunikation

Threads kommunizieren miteinander durch gemeinsam benutzte Variablen, Datenstrukturen oder Dateien.

Unter gemeinsam benutzte Variablen versteht man Variablen, auf denen mehrere Prozesse gleichzeitig Zugriff haben.Das folgende Concurrent Pascal Programm stellt ein Beispiel dafür:

```
x:=0;
cobegin
   begin x:=1;x:=x+1 end;
   begin x:=2;x:=x+1 end;
coend;
```

In dem *cobegin-coend* Block werden alle Aktionen konkurrent ausgeführt. Das hat zur Folge, dass das Programm mehrere mögliche Ausführungsfolgen hat :

```
x:=0;x:=1;x:=2;x:=x+1;x:=x+1;
x:=0;x:=1;x:=x+1;x:=2;x:=x+1;
```

Die Variable x ist eine gemeinsam benutzte Variable, über die parallel laufende Threads kommunizieren.

Eine weitere Form von Kommunikation stellt der *Nachrichtenaustausch* zwischen Threads dar. Das Nachrichtenaustausch-Modell lässt sich nach den folgenden Hauptkriterien klassifizieren:

- **Puffern.**
  In diesem Fall unterscheidet man zwischen *gepufferte* und *ungepufferte* Kommunikation. Wenn die Kommunikation gepuffert ist, bleibt jede gesandte Nachricht bis zu ihrem Empfang verfügbar. Das heisst, dass sogar wenn der Empfänger Prozess nicht bereit ist die Nachricht zu empfangen, geht sie nicht verloren. Bei ungepufferte Kommunikation geht eine gesandte Nachricht verloren, wenn kein Prozess bereit ist sie zu empfangen.
- **Synchronizität.** Hier wird zwischen *synchrone* und *asynchrone* Kommunikation unterschieden. Bei synchrone Kommunikation ist das Senden und Empfangen von Daten koordiniert. Mit anderen Worten, der Sender Prozess darf keine Daten schicken, falls der Receiver nicht bereit ist sie zu empfangen. Bei asynchrone Kommunikation schickt der Sender die Nachricht, auch wenn der Empfänger nicht bereit ist sie zu empfangen.
- **Reihenfolge der Nachrichten**
  Hier unterscheidet man zwischen *geordnete* und *ungeordnete* Kommunikation. Als geordnet bezeichnet man solche Kommunikation, bei der die Reihenfolge in der die Nachrichten gesendet wurden, behalten wird. Wenn die Kommunikation ungeordnet ist, werden die Nachrichten nicht in derselben Reihenfolge empfangen, in der sie gesendet wurden.

3.2 Koordination

Wenn Prozesse unabhängig und isoliert voneinander ablaufen, besteht keine Notwendigkeit zur Koordination. Koordination ist jedoch erforderlich bei:

- **Kooperation:**
  Prozesse arbeiten an einer gemeinsamen Aufgabe und müssen interagieren, z.B. sich synchronisieren.
  Beispiel: Ein Prozess füllt einen Pufferplatz mit Daten, die von einem anderen Prozess über eine Datenleitung übertragen werden sollen.

- **Konkurrenz:**
  Prozesse stehen im Wettbewerb um gemeinsame Betriebsmittel, die exklusiv belegt werden müssen. Auch dafür müssen sie sich synchronisieren.
  Beispiel: Zwei Prozesse wollen je eine Datei auf einem Drucker ausgeben.

3.3 Atomarität

Eine Aktion ist atomar, wenn sie während ihrer Ausführung nicht unterbrochen werden darf. Die Zuweisung x:= x+1 der Variable x besteht aus mehreren Elementaranweisungen (Maschineninstruktionen), die von der CPU ausgeführt werden. Wenn die Zuweisung nicht atomar ist, könnte der ausführende Prozess nach jeder Elementaranweisung unterbrochen werden. Inzwischen könnte der Wert der Variable von einem anderen Prozess verändert werden. Das hat zur Folge, dass der Variable einen anderen Wert zugewiesen wird, als der zuvor erwartete. Damit konkurrente Programme die erwünschten Ergebnisse liefern, sollen Programmiersprachen, die Nebenläufigkeit untertstützen, das Konzept von atomaren Aktionen zur Verfügung stellen.

3.4 Kritischer Abschnitt

Mit *kritischem Abschnitt (kA)* wird ein Abschnitt eines Programms bezeichnet, der aus einer Folge von Elementaranweisungen besteht, die exklusiv durchgelaufen werden müssen. Wenn zwei Prozesse gleichzeitig auf gemeinsame Daten zugreifen, kann es zur Fehler kommen. Deshalb ist es erforderlich, dass nur ein Prozess einen kritischen Abschnitt zu einem Zeitpunkt betreten darf. Das folgende Beispiel zeigt, dass diese Bedingung sehr wichtig ist:

```
procedure sign(person)
begin
   number := number + 1;
   list[nummber] := person;
end;

cobegin
   sign(bill);
   sign(fred);
coend;
```

Da die Funktion *sign* in dem *cobegin-coend* Block gleichzeitig von mehreren Prozessen ausgeführt werden kann, kann vorkommen, dass nur einer der Namen in der Liste eingetragen wird. In dem Beispiel ist der Code der Prozedur *sign* ein kritischer

Abschnitt. Das Programm soll so modifiziert werden, dass zur Zeit nur ein Prozess die Prozedur *sign* ausführt. Es wurden spezielle *Mechanismen zur Synchronisation* entworfen, die dafür sorgen, dass Prozesse die kritischen Abschnitte eines Programms koordiniert betreten. In Kapitel 5 werden einige Mechanismen zur Synchronisation präsentiert.

4 Problemme

Nebenläufige Programmierung ist für große Systeme erforderlich. Es ist jedoch zeitaufwendig und komplex.

Eine der größten Schwierigkeiten beim Entwurf und Testen konkurrenter Programme bereitet *Nichtdeterminismus*. Ein Programm ist nichtdeterministisch, wenn sie mit derselben Eingabe verschiedene Ausgaben liefert und somit viele mögliche Ausführungen hat. Es ist schwer für die Programmier an allen diesen Ausführungen zu denken und die Fehler, die nur in einigen von denen auftreten, zu identifizieren. Das Programm mit allen Ausführungsmöglichkeiten zu testen, könnte ausserdem Jahre dauern.

Ein weiteres Problem, das bei konkurrenten Programmen auftreten kann, ist *Deadlock*. Ein *Deadlock* entsteht, wenn ein Prozess wegen eines anderen Prozesses nicht mehr ausgeführt werden kann. Ein Prozess P1 wartet auf bestimmte Resource, damit er seine Ausführung fortsetzen kann. Prozess P2 besitzt diese Resource, braucht aber gleichzeitig eine Resource, die im Besitz von P1 ist. In diesem Fall blockieren sich beide Prozesse gegenseitig und entsteht ein Deadlock.

5 Mechanismen zur Synchronisation von Prozessen

Mechanismen zur Synchronisation sind Techniken , die es dem Betriebssystem erlauben, Prozessen den Zutritt zum kritischen Abschnitt zu verwehren. Sie sollen folgende Qualitätsanforderungen erfüllen:

- **Exklusivität (Sicherheit)**:
  Die Prozesse sollen den kritischen Abschnitt exklusiv durchlaufen.
- **Verklemmung (Blockadefreiheit)**:
  Die Prozesse dürfen sich nicht gegenseitig blockieren, derart, dass keiner mehr ablaufen kann, wenn einer in seinem Fortschreiten aufgehalten wird.
- **Zeitinvarianz**:
  Der exklusive Eintritt in dem kritischen Abschnitt darf für keinen Prozess an irgendwelche Zeitbedingungen geknüpft sein.

Im folgenden werden die drei wichtigsten *Mechanismen zur Synchronisation* näher betrachtet, nämlich *Semaphoren*, *Locks* und *Monitore*.

5.1 Semaphore

Semaphoren wurden erst von Edsger W. Dijkstra im Jahre 1968 eingeführt. Jeder zu schützenden Datenmenge wird eine Variable (Semaphore) zugeordnet. Ein Semaphore (Steuervariable) signalisiert einen Zustand (Eintritt eines Ereignisses) und gibt in Abhängigkeit von diesem Zustand den weiteren Prozeßablauf frei oder versetzt den betreffenden Prozeß in den Wartezustand. Beim Nutzen von unteilbaren Ressourcen

(z. B. den Prozessor) wird ein binärer Semaphore verwendet, bei N Teil-Ressourcen (z. B. Arbeitsspeicher-Segmente oder Plätze in einem Puffer) kommen Werte von 1 bis N vor.

Semaphoren für den gegenseitigen Ausschluß sind dem jeweiligen exklusiv nutzbaren Betriebsmittel zugeordnet und verwalten eine Warteliste von Prozessen für dieses Betriebsmittel. Sie sind allen Prozessen zugänglich. Prozessen, die voneinander datenabhängig sind, werden Semaphore für die Ereignissynchronisation direkt zugeordnet. Sie dienen zur Übergabe einer Meldung über das Ereignis zwischen den Prozessen.

Semaphoren werden durch zwei atomar auszuführende Operationen manipuliert:

- **wait-Operation** (Anfrage-Operation): wird ausgeführt, wenn ein Prozess auf das von dem Semaphore überwachte Betribsmittel zugreifen will
  `if (s > 0)` /* s - Semaphore Variable*/
  `s = s - l;` /* Prozeß kann weiterlaufen, Zugriff für andere Prozesse wird gesperrt*/
  `else`
  /* der die wait-Operation ausführende Prozess wird wartend; Eintrag des Prozesses in die vom Semaphor verwaltete Warteliste; */

- **signal-Operation** (Freigabe-Operation): wird ausgeführt, wenn ein Prozess das Bertiebsmittel freigibt
  `if (Warteliste leer)`
  `s = s + l;` /* Zugriff für andere, noch nicht wartende Prozesse wird freigegeben */
  `else`
  /* nächster Prozess in Warteliste wird bereit; Zugriff für wartenden Prozess wird freigegeben */

5.2 Locks

Die grundlegende Idee bei diesem Mechanismus ist, dass jeder Prozess eine *wait* und eine *signal* Operation ausführt, bevor bzw. nach er gemeinsame Daten benutzt. Es wird eine *Lock*-Variable benötigt, die durch die beiden Operationen getestet, gesetzt oder freigegeben wird. Die *wait* Prozedur testet die Lock-Variable. Ist sie gesetzt, dann wurde auf den Daten zugegriffen und der *wait* aufrufende Prozess soll warten. Signalisiert die Lock-Variable, dass kein Prozess auf den Daten momentan Zugriff hat, läuft der *wait* aufrufende Prozess weiter . Wenn der Prozess den kritischen Abschnitt verlässt, führt er die *signal* Prozedur aus, wobei die Lock-Variable freigegeben wird und somit ein auf diese Lock-Varible wartender Prozess aktiviert wird.

Das Problem, das bei Nutzung von Locks entsteht, ist dass das Testen und das Setzen der Lock-Vriable zwei verschiedene Operationen sind. Nachdem ein Prozess die Variable getestet hat und bevor er sie gesetzt hat, könnte er unterbrochen werden. Ein anderer Prozess setzt die Variable und bekommt Zugriff auf den Daten. Schliesslich wird der unterbrochene Prozess fortgesetzt und er setzt seinerseits die Variable. So haben zwei Prozesse gleichzeitig Zugriff auf den Daten und entstehen unerwünschte Inkonsistenzen.

5.3 Monitore

In einem Monitor werden die zu den kritischen Abschnitten gehörenden Daten und Prozeduren zusammengepackt. Der Monitor achtet darauf, daß maximal ein Prozeß gleichzeitig Zugriff auf die in ihm enthaltenen Komponenten hat. Normalerweise gibt es mehrere Monitore gleichzeitig. Prozesse dürfen aus jedem Monitor Prozeduren aufrufen, soweit dieser Monitor noch nicht von einem anderen Prozeß belegt ist. Möchte ein Prozess auf einen schon besetzten Monitor zugreifen, so muß er warten. Die Organisation von wartenden Prozessen wird über eine Bedingungsvariable und zwei Operationen, die zu der Bedingungsvariablen gehören, realisiert. Eine Bedingungsvariable *c* gehört zu einer Synchronisationsbedingung A(c) (z.B. „es darf maximal ein Prozeß in Datei x schreiben“). *c* wird normalerweise als Menge aller durch *c* blockierten Prozesse beschrieben. Wenn diese Menge leer ist, blockiert *c* keinen Prozeß. Die Operation *wait* wird von einem Monitor ausgeführt, falls die zu dem anfragenden Prozeß gehörige Synchronisationsbedingung A(c) nicht erfüllt ist. Der Prozeß wird in die Menge der wartenden Prozesse eingefügt und blockiert. Die *signal* Operation wird wieder vom Monitor ausgeführt, sobald eine Synchronisationsbedingung A(c) erfüllt wird, die vorher nicht erfüllt war. Der Aufruf von *signal* hat zur Folge, daß ein wegen A(c) blockierter Prozeß die Kontrolle über den Monitor erhält und somit nicht mehr blockiert ist. Gäbe es keinen blockierten Prozeß, hat *signal* keine Bedeutung. Obwohl Monitore durch Senmaphore implementierbar sind und umgekehrt, sind Monitore ein mächtigerer Mechanismus zur Synchronisation.

6 Anforderungen an Programmiersprachen

In Programmiersprachen, die Nebenläufigkeit direkt unterstützen, lassen sich konkurrente Programme viel leichter und effektiver entwerfen. Solche Programmiersprachen bieten verschiedene Konzepte, die speziell für nebenläufiges Programmieren entworfen sind. Beispiele dafür sind Threads, Kommunikationsabstraktionen und Synchronisationsprimitiven. Unter Kommunikationsabstraktionen versteht man die Art und Weise, auf der die Kommunikation zwischen Threads in den verschiedenen Programmiersprachen realisiert ist. Durch Synchronisationsprimitiven, wie z.B Semaphoren, Locks, wird Koordination zwischen den Aktionen von Threads erreicht.

Concurrent ML, Alice, Java, Erlang, MPD, ADA, Actors, Concurrent Pascal, das Cobegin/Coend Modell sind Beispiele für Programmiersprachen, die Nebenläufigkeit unterstützen. In dieser Arbeit werden die wichtigsten Aspekte von CML, Alice und Java eingeführt.

6.1 Concurrent ML

Concurrent ML ist eine konkurrente Erweiterung von Standard ML. Die Sprache wurde von John Reppy entworfen. Der wichtigste Aspekt in CML ist das Konzept von Events, das dem Programmierer die Möglichket gibt, die Kommunikation und die Synchronisation zwischen Threads zu definieren.

6.1.1 Threads

Ein Prozess in CML wird als Thread bezeichnet. Anfänglich besteht jedes Programm aus einem einzigen Thread, wobei neue Threads durch die Prozedur *spawn* erzeugt

werden. Die Signatur der Funktion *spawn* sieht wie folgt aus:

```
spawn: (unit ⟶ unit) ⟶ thread_id
```

Wenn *spawn f* evaluiert wird, wird der Aufruf *f()* als getrennter Thread ausgeführt, wobei der neue Thread mit anderen Threads durch Kanäle kommuniziert. Der Thread, der *spawn f* evaluiert, heisst *parent* Thread. Der *f()* ausführene Thread wird als *child* Thread bezeichnet. Der child Thread darf, nachdem der parent Thread terminiert hat, weiterexistieren.

6.1.2 Kommunikation zwischen Threads

Die Kommunikation zwischen den Threads wird in CML durch Kanäle realisiert. Ein Kanal wird von dem *channel* Konstruktor erzeugt:

```
channel: unit ⟶ ´a channel
```

channel() produziert einen Kanal, der mit Werten vom Typ *´a* kommuniziert. Zwei Operationen werden auf Kanäle durchgeführt: *send*, wenn an diesem Kanal Werte geschickt werden, und *receive*, wenn Werte von diesem Kanal empfangen werden:

```
receive: ´a chan ⟶ ´a

send: (´a chan * ´a) ⟶ unit
```

Wenn ein Thread *receive* auf einem Kanal *c* ausführt, der mit Werten vom Typ *´a* kommuniziert, kommt als Ergebnis der Wert, den der Thread von *c* bekommen hat. Führt ein Thread *send(c,x)* aus, dann schickt er Variable x an Kanal c.
Der Nachrichtenaustausch in CML ist synchron (s. Kapitel 3.1), d.h. dass eine Nachricht nur dann gesendet wird, wenn beide der Empfänger und der Sender bereit sind zu kommunizieren. Falls ein Thread eine Nachricht einem Kanal schickt, kein Thread aber bereit ist *receive* auf denselben Kanal auszuführen, blockert der Sender, bis ein Thread *receive* durchführt.

Das folgende Beispiel zeigt Synchronisation von Threads durch einen Kanal:

```
c = channel();
spawn (fn() ⟹ ... <A> ... send(c, 3); ... <B> ...);
spawn (fn() ⟹ ... <C> ... receive c; ... <D> ...);
```

Es wird erreicht, dass die Threads B und D nur dann gestartet werden dürfen, wenn beide A und C schon ausgeführt sind. Dabei ist die Reihenfolge bei A und C, sowie auch bei B und D beliebig.

Der synchrone Nachrichtenaustausch bereitet in bestimmten Kooperationsmustern Probleme: typisches Beispiel dafür ist, wenn mehrere Produzenten Aufträge an mehrere Konsumenten verteilen. Deshalb bietet CML auch *Multicast Kanäle.* Durch Multicast Kanäle kann eine gesandte Nachricht mehrere Empfänger erreichen.

6.1.3 Events

Die *send* Funktion kann in zwei Teile zerlegt werden: der Code, der die Ausführung von einer *send* Operation hervorruft und die eigentliche Ausführung von *send*. Wir können es auch so auffasen: der Code ist eine Funktion, die *send* ausführt und die eigentliche Ausführung von *send* ist der Aufruf dieser Funktion. Die *send* ausführende Funktion ist als *send Event* bezeichnet. Der Aufruf dieser Funktion nennt man *synchronisieren auf diesem Event*. In CML stellt der Typ *´a event* den Typ einer Aktion dar, die einen Wert vom Typ *´a* liefert, wenn sie ausgeführt wird, oder in anderen Worten, wenn ein Thread auf der Aktion synchronisiert. Die Funktion *sync* bekommt ein Event als Argument und synchronisiert auf ihm:

```
sync: ´a event ⟶ ´a
```

Es gibt zwei Arten von Events: *receive Events (recvEvt)* und *send Events (sendEvt)*, die auch basis Events genannt werden. Wenn ein Thread auf einem *receive Event* synchronisiert, indem er *sync(recvEvt)* aufruft, ist das Ergebnis der Wert, den der Thread vom Kanal bekommt. Die Funktion *receive* auf Kanäle kann durch *recvEvt* und *sync* definiert werden:

```
fun receive(ch) = sync(recvEvt(ch))
```

Ein Thread kann Werte von einem Kanal dadurch empfangen, dass er auf receive Events für diesen Kanal synchronisiert. Analog kann *send* durch *sendEvt* und *sync* definiert werden:

```
fun send(ch,x) = sync(sendEvt(ch,x))
```

Ein Thread schickt Werte an einem Kanal, indem er auf send Events für diesen Kanal synchronisiert.

Die grosse Bedeutung der Events kommt von Operatoren, die komplizierte Events von den basis Events bilden. Ein Operator für Events ist die *wrap* Funktion, die ein Event, das einen Wert vom Typ *´a* produziert, mit Funktion vom Typ `(´a →´b)` kombiniert, damit sie ein Event bekommt, das Werte vom Typ *´b* produziert. Gegeben ein Event *e*, das *x* vom Typ *´a* liefert, und f vom Typ `(´a →´b)` , dann kommt von *wrap(e,f) f(x)* als Ergebnis.

6.1.4 Synchronisierter gemeinsamer Speicher

In CML dürfen Threads Speicherzelle miteinander teilen, indem sie ihnen Werte zuweisen oder ihren Inhalt lesen. Die gemeinsame Nutzung von Speicherzellen soll koordiniert sein, damit Fehler vermieden werden können. Koordinierte Zuweisungen von genmeinsam benutzten Variablen können durch das Konzept des synchronisierten gemeinsamen Speichers realisiert werden. Es gibt einige mögliche Implementierungen von diesem Konzept. Eine davon ist M-Struktur. Eine M-Struktur ist eine Speicherzelle, die sich in zwei Zustände befinden kann: leer und voll. Am Anfang ist jede M-Struktur leer. Ihr darf einen Wert durch die Prozedur *put* zugewiesen werden. Wenn sie voll ist, kann ein Thread ihren Wert durch die Prozedur *take* ablesen und sie danach leer lassen. Es existieren zwei Exceptions für die Fälle, dass aus einer leeren M-Struktur gelesen wird oder in einer vollen Zelle geschrieben wird. Durch die zweite Exception

wird vermieden, dass der Wert einer Zelle überschrieben wird.

6.2 Alice

Alice ist eine auf Standard ML basierte funktionale Sprache, die nebenläufige Programmierung, Constraintprogrammierung und Distributedprogrammierung unterstützt.

6.2.1 Threads und Futures

Nebenläufigkeit ist in Alice light-weight, d.h. die aktive Einheiten sind Threads. Mehrere Threads können paralell ablaufen. Sie werden wie in CML durch die Prozedur *spawn* erzeugt, bei der eine Future als Ergebnis geliefert wird. Eine Future ist ein Platzhalter für das Resultat einer nebenläufigen Berechnung. Sobald das eigentliche Ergebnis berechnet ist, wird die Future durch den Wert des Ergebnisses ersetzt. Das folgende Beispiel zeigt die Implementierung der Prozedur *fib*, die die n-te fibonacci Zahl liefert, durch Futures:

```
fun fib (0 | 1) = 1
  | fib n = fib (n-1) + fib (n-2)
  val fib : int → int = _fn
  val n = spawn fib 35
  val n : int = _future
  inspect n
  val it : unit = ()
```

Der Variable n wird das Ergebnis des Ausdrucks *spawn fib 35* zugewiesen, was in dem Fall eine Future ist. Der Wert der Future kann durch einen sogenannten Inspektor beobachtet werden, der die Ergebnisse auf den Display zeigt. Zuerst erscheint die 35-te fibonacci Zahl als Future auf dem Display. Erst, wenn die Berechnung zum Ende ist und der Thread terminiert hat, wird die Future durch das fertige Ergebnis ersetzt und auf den Display gezeigt.

6.2.2 Data-flow Synchronisation

Data-flow Synchronisation ist ein mächtiger Mechanismus für nebenläufige Programmierung. Die Hauptidee besteht darin, dass Futures als Werte übergeben werden können. Wenn ein Thread den Wert einer Future fordert, blockiert er und wartet bis dieser Wert berechnet wird und die Future durch ihn ersetzt wird.

Wenn eine Berechnung wegen Fehler unterbrochen wird, wird ein Exception geworfen. In diesem Fall spricht man von einer *failed Future*. Eine *failed Future* enthält ein Exception mit der Ursache der Fehler. Eine *failed Future* verursacht jedoch keinen Fehler bei dem die Berechnung ausführenden Thread. Erst wenn der Wert, für den eine *failed Future* steht, von einem Thread gefordert wird, wird die Exception geworfen, die in der Future enthalten ist.

Als eine Sprache, die Nebenläufigkeit untrestützt, bietet Alice atomare Operationen für Zugriff auf Referenzen:

```
  val r = ref 10
  val r : int ref = ref 10
  Ref.exchange (r, 20)
```

```
val it : int = 10
```

Der Wert der Referenz wird in einer Operation gelesen und durch einen neuen Wert ersetzt. So können Synchronisationsprimitiven realisiert werden, bei denen atomares Testen und Setzen von Lock Variablen benötigt wird.

6.3 Java

Nebenläufigkeit ist ein interner Teil von Java und von dem run-time System. Threads werden in Java erzeugt, indem eine Klasse entweder von der Klasse *Thread* abgeleitet wird, oder direkt das *Runnable* interface implementiert. Zusätzlich muss die abgeleitete Klasse die ererbte *run()* Methode definieren. Java Threads kommunizieren miteinander durch Aufrufen von Methoden gemeinsam benutzter Objekte. Da die gemeinsame Nutzung von Objekten Koordination braucht, unterstützt Java verschiedene Synchronisationsprimitiven, wie z.B Semaphore, Monitore, etc.

6.3.1 Threads

Java Threads sind Objekte einer Klasse, die von der Klasse *Thread* abgeleitet ist. Von ihr werden einige Methoden geerbt, wie z.B *start*, *interrupt*, *suspend*, *stop*. *Start* ist eine Methode, die einen neuen Thread erzeugt, indem sie die Java virtuelle Maschine (JVM) veranlasst, die *run* Methode parallel mit dem *parent* Thread auszuführen. Durch *interrupt* wird ein Thread suspendiert und in ihm wird ein Exception geworfen. *Suspend* verursacht, dass ein Thread unterbrochen wird, bis er eventuell später wieder aktiviert wird. *Stop* unterbricht einen Thread.

Programmierer haben sich überzeugt, dass diese Methoden zu Problemen führen. Wenn ein Thread suspendiert wird, der sich im kritischen Abschnitt befindet, kommt es zum Deadlock. Ein weiters Problem entsteht, wenn ein Thread durch *stop* unterbrochen wird, und somit alle von ihm gesetzte Locks freigegeben werden. Objekte, die von dieser Locks geschutzt werden, können in einem nicht konsistenten Zustand gelassen werden. So können andere Threads auf diesen beschädigten Objekten zugreifen.

Threads, die in der selben JVM ausgeführt werden, werden nach ihren Prioritäten eingeplant, wobei der Thread mit der höchsten Priorität bei der Ausführung bevorzugt wird.

6.3.2 Kommunikation und Synchronisation

Java Theads kommunizieren miteinander durch Zuweisung von Werten gemeinsam benutzte Variablen oder durch Aufrufen von Methoden gemeinsam benutzter Objekten. Da die Nutzung von gemeinsamen Objekten und Variablen problematisch ist, werden in der Regel drei Synchronisationsmechanismen angesetz, damit die Zugriffe auf den Objekten keine Fehler verursachen. Threads werden in Java durch die folgenden Mechanismen synchronisiert: *Locks*, *Wait sets* und *Thread termination*.

6.3.2.1 Locks

Java Objekte sind eigentlich Resourcen (s. Kapitel 3). Jedes Objekt in Java besitzt ein Lock, das den gegenseitigen Ausschluss bei Nutzung dieses Objektes sichert. Locks werden von dem *synchronized statement* benutzt, die die folgende Form hat:

```
synchronized(objekt) {
   statements
}
```

Durch das *synchronized statement* wird ein Objekt als synchronisiert deklariert, was zur Folge hat, dass nur ein Thread zu einem Zeitpunkt auf diesem Objekt zugreifen darf. Bevor ein Thread die Anweisungen eines Objektes ausführen darf, testet er das Lock dieses Objektes. Ist das Lock schon von einem anderen Thread gesetzt, wartet der testende Thread bis es wieder freigegeben wird. Erst dann darf der Thread die Anweisungen ausführen, wobei er das Lock setzt, damit andere Threads auf denselben Objekt nicht zugreifen dürfen. Nachdem der Thread das gemeinsam benutzte Objekt verlässt, gibt er das Lock wieder frei.

Java Locks können mehrmals von einem Thread gesetzt werden. Das passiert, wenn ein *synchronized statement* im Rahmen eines anderen *synchronized statement* ausgeführt wird, oder wenn zwei synchronisierte Methoden sich gegnseitig aufrufen.

6.3.2.2 Wait sets

Jedes Objekt besitzt ein Wait set, das als eine Warteschlange von Threads betrachtet werden kann, die auf dieses Objekt zugreifen wollen. Die Methoden *wait*, *notify* und *notifyAll* der Klasse *Object* benutzen das Wait set eines Objektes, damit die Zugriffe auf das Objekt synchronisiert werden. Ein Thread kann sich dadurch suspendieren, dass er die Methode *wait* aufruft. Danach führt er keine Anweisungen mehr aus, bis ihn ein anderer Thread durch *notify* weckt. Die Beziehung zwischen Threads, die die Methoden *wait* und *notify* aufrufen, kann als eine Produzent/Konsument Beziehung betrachtet werden. Der Konsument suspendiert sich, da ihm keine Eingabe zur Verfügung steht. Der Produzent weckt den Konsumenten, indem er die Eingabe produziert. Die Methode *notifyAll* hat denselben Effekt wie *notify*, mit dem Unterschied, dass alle suspendierte Prozesse das Wait set des Objektes verlassen und zur Zuteilung des Prozessors eingeplant werden.

6.3.2.3 Thread termination

Die dritte Form von Synchronisation in Java stellt Thread termination dar. Darunter versteht man ein Thread, der darauf wartet, dass ein anderer Thread terminiert. Dies wird durch die *join* Methode erreicht:

```
class Compute_thread extends Thread {
   private int result;
   public void run() { result = f(...);}
   public int getResult() { return result;}
}
...
Compute_thread t = new Compute_thread;
t.start()
...
t.join(); x = t.getResult();
...
```

In dem Beispiel berechnet ein Objekt der Klasse *Compute_ thread* ein beliebiges Ergebnis. Das Programm erstellt ein Objekt (neuer Thread) derselbe Klasse und startet

es. Wenn das Ergebnis benötigt wird, wird die *join* Methode aufgerufen, damit der laufende Thread solange wartet, bis die Berechnung des Ergebnisses fertig ist.

6.3.3 Synchronisierte Methoden

Threads kommunizieren, indem sie Methoden gemeinsam benutzter Objekte aufrufen. Beliebige Methoden können nach Bedarf *synchronisiert* werden. Die Deklaration einer Methode als synchronisiert hat denselben Effekt wie das Setzen dieser Methode in dem Körper eines *synchronized statements*. Wenn ein Thread eine synchronisierte Methode aufrufen möchte, muss er das Lock des Objektes testen, das diese Methode enthält. Nur wenn das Lock nicht gesetzt ist, darf der Thread dieser Methode aufrufen. Neben synchronisierten Methoden, enthalten Klassen auch nicht synchronisierte Methoden, die gleichzeitig von mehreren Threads aufgerufen werden dürfen. Die folgende Klasse enthält zwei synchronisierte und eine nicht synchronisierte Methode:

```
class LinkedCell {
   protected double value;
   protected LinkedCell next;
   public LinkedCell (double v, LinkedCell t) {
   ...
   }
   public synchronized double getValue() {
   ...
   }
   public synchronized void setValue(double v) {
   ...
   }
   public LinkedCell next() {
   ...
   }
}
```

Die Klasse implementiert Zellen mit zwei Felder: ein Wert Feld und ein Zeiger auf der nächste Zelle. Wenn die Methode *setValue* nicht synchronisiert wäre, könnten zwei Threads die Methode aufrufen und somit den Wert einer Zelle verändern. In diesem Fall enthält das Wert Feld dieser Zelle weder den Wert, der ihr der erste Thread zuweisen wollte, noch diesen von dem zweiten Thread, sondern eine Mischung aus beiden Werten. Analog dürfen die *setValue* und die *getValue* Methoden auf einer Zelle nicht gleichzeitig ausgeführt werden. In diesem Fall bekommt der *getValue* aufrufende Thread nicht das richtige Ergebnis, sondern eine Mischung aus dem alten Wert und dem neuen zuzuweisenden Wert.

7 Zusammenfassung

In der vorliegenden Arbeit wurden die wichtigsten Aspekten der nebenläufige Programmierung diskutiert. Diese Idee ist von sehr grosse Bedeutung für grosse Komputersysteme. Das Erstellen von nebenläufige Programme ist jedoch kompliziert und zeitaufwendig. Nichtdeterminismus macht das Testen nebenläufiger Programme schwierig, da der Programmierer nicht immer alle mögliche Ausführungen und die dabei auf-

tretenden Fehler berücksichtigen kann. Kommunikation, Koordination, Atomarität, kritischer Abschnitt sind die wichtigsten Begriffe, wenn man von Nebenläufigkeit spricht. Es werden Synchronisationsmechanismen wie Locks, Semaphoren und Monitore angesetzt, um Zugriffe auf gemeinsam benutzte Objekte durch parallel ablaufende Prozesse aufeinander abzustimmen. Der Entwurf nebenläufiger Programme lässt sich leichter und effektiver in Programmiersprachen realisieren, die Nebenläufigkeit unterstützen, da sie spezielle Mechanismen zur Synchronisation und Kommunikation bieten.

Drei Beispiele für solche Programmiersprachen wurden zusammen mit ihren wichtigsten Features in dieser Arbeit detailiert betrachtet: CML, Alice und Java.

CML ist eine konkurrente Erweiterung von Standard ML. In den drei Sprachen wird das Konzept von Threads für eine sequentielle Einheit benutzt. CML Threads kommunizieren miteinander, indem sie synchron Nachrichten durch Kommunikationskanäle senden. Events sind der interessanteste Aspekt von CML. Sie sind synchrone Aktionen, die ausgerführt werden, wenn ein Thread auf ihnen synchronisiert.

Alice ist eine auf SML basierte Programmiersprache, bei der Nebenläufigkeit mit Futures realisiert wird. Eine Future ist ein Platzhalter für einen später kommenden Wert. Futures können als Werte übergeben werden. Ein Thread, der den Wert einer Future fordert, blockiert und wartet, bis der Wert berechnet wird. Das ist als Data-flow Synchronisation bekannt.

Java Threads sind Objekte von Klassen, die von der Klasse *Thread* abgeleitet sind. Sie werden durch die *start* Methode gestartet und ihre Funktionalität wird in der *run* Methode implementiert. Die Kommunikation zwischen Threads wird durch Aufrufe von Methoden gemeinsam benutzter Objekte realisiert. Dabei werden die Aufrufe durch folgende Mechansimen miteinander koordiniert: Locks, Wait sets und Thread termination.