Concurrent ML

Vortrag zum Proseminar *Programmiersprachen für das 3.Jahrtausend*
von

David Frese

1 Einleitung

Concurrent Meta Language ist eine Erweiterung der Programmiersprache ML um das Konzept der *Nebenläufigen Programmierung*. Andere gebräuchliche Synonyme dafür sind *Concurrent Programming*, *Multi-tasking* oder *Multi-threading*. Es ist also die *simultane Bearbeitung von Anweisungen innerhalb eines Programms*. *Concurrent Programming* bedeutet nun, daß man die Arbeit die ein Programm verrichten soll in mehrere Teile teilt, die gleichzeitig ausgeführt werden. Diese Teile werden in der Regel *Threads* (engl. Fäden) genannt. Natürlich wird es dabei äußerst selten vorkommen, daß die einzelnen Threads völlig unabhängig voneinander laufen. Deshalb ist die entscheidene Problematik, die Kommunikation zwischen den Threads eines Programms zu ermöglichen, und sicher zu machen. Concurrent ML erweitert ML um Prozeduren und Primitiva, die es dem Programmierer auf sehr einfache und elegante Art ermöglichen die nebenläufige Programmierung zu verwenden.

2 Kurzeinführung in ML

ML bedeutet *Meta Language*, denn ML ist als Beschreibungssprache für Beweisstrategien in automatischen Beweissystemen entstanden. Über die Jahre ist daraus aber eine elegante funktionale Programmiersprache geworden. Sie basiert, genauso wie z.B. Scheme, auf dem Lambda-Kalkül von Church. Der größte Unterschied zu Scheme ist aber das strenge und statische Typsystem von ML. D.h. alle Werte und Variablen haben einen eindeutigen Typ - z.B. Ganzzahlen oder Zeichenketten.

2.1 Einfache Typen und Werte

Typ	Wert/Literal	Beschreibung
`int`	`15` oder `~1000`	Ganzzahlen
`real`	`5.2`	Fließkommazahlen
`string`	`"Hallo Welt"`	Zeichenketten
`bool`	`true` oder `false`	Boolsche Werte
`unit`	`()`	Der leere Typ, vergleichbar mit dem C-Typ `void`
`'a list`	`[1,2,3]` oder `["a","b"]`	Listen können in ML nur einen festen Typ von Werten enthalten. Dieser ist aber beliebig, und dafür steht der Bezeichner `'a`. `'a list` ist ein sog. abstrakter Typ.
`int list`	`[5,6,7,8]`	Explizite Angabe, daß die Liste Ganzzahlen enthalten muß.
`int list list`	`[[1], [], [4,5]]`	Eine Liste von Ganzzahl-Listen

2.2 Tupel

Alle Typen und Werte können in ML zu Tupel fester aber beliebiger Länge kombiniert werden. Diese dienen oft als Parameter von Prozeduren.

Typ	Wert
`int * int`	`(2, 5)`
`int * real`	`(2, 5.0)`
`string * string * int`	`("David", "Frese", 2001)`
`(int * int) * string`	`( (5, 7), "test" )`

Man kann auch direkt auf ein bestimmtes Element eines Tupels zugreifen:

```
val x = (1,2,3);
2 = #2 x;    => true
```

2.3 Funktionen

Funktionen sind in ML Werte wie jeder andere auch. Sie lassen sich mit dem Schlüsselwort `fn` an jeder beliebigen Stelle erzeugen.

Typ	Wert	Erklärung
`int -> int`	`fn a : int => a+5`	Eine Funktion die zu einer Ganzzahl 5 hinzuaddiert. Durch das `+5` merkt ML übrigens automatisch, daß der Parameter eine Ganzzahl sein muß, wie man am nächsten Beispiel sieht.
`int * int -> int`	`fn (a,b) => a + b - 1`	Der Typ wird automatisch erkannt.
`int -> int -> int`	`fn a => fn b => a - b`	Eine geschönfinkelte Funktion.

Standardfunktionen und Ausdrücke Einige häufig benuzte eingebaute Funktionen:

Funktion	Beispiel	Resultat	Erklärung
`::`	`1::[2,3]`	`[1,2,3]`	Listenkonstruktion
`@`	`[9,10]@[2]`	`[9,10,2]`	Listenkonkatenation
`^`	`"M"^"L"`	`"ML"`	Stringkonkatenation

Ausdrucksdefinition	Erklärung
`if <test> then <cons> else <alt>`	Eine Besonderheit in ML ist, daß `<test>` ein Boolscher Wert sein muß, und daß der `else` Teil immer angegeben werden muß.

2.4 Variablenbindung und Funktionsdefinition

Einfache Variablenbindungen werden in ML mit dem Schlüsselwort `val` definiert. Bindungen lassen sich *nicht* mutieren!

```
val name : string = "David"
val n : int = 100
val lst : int list = [-1,12,63]
val f : int -> int = (fn x => x*2)
```

Auch hier hätte man die Angaben der Typen weglassen können, weil ML sie automatisch erkennen kann.
Für die Definion von Funktionen gibt es eine spezielle Syntax, die etwas einfacher zu handhaben ist als mit `fn`:

Definition	Typ
`fun f (a,b) = (a + b) mod b`	`int * int -> int`
`fun g a b = [a,b]`	`'a -> 'a -> 'a list`

Pattern Matching Man kann Funktionen in ML auf sehr mathematische Weise für verschiedene Eingabeparameter getrennt definieren. Der Interpreter versucht dann automatisch bei einen Aufruf der Funktion, die übergebenen Operanden in eine der Definition einzupassen. Die bekannte Funktion `map` lässt sich in ML z.B. sehr elegant definieren:

```
fun map f [] = []
  | map f a::l = (f a)::(map f l)
```

Lokale Bindungen Lokale Bindungen lassen sich mithilfe des `let ... in ... end` Ausdrucks realisieren. Als Beispiel eine endrekursive Funktion die eine Liste rumdreht.

```
fun rev lst = let
      fun loop ([], res) = res
        | loop (a::l, res) = loop (l, a::res)
      in
        loop (lst, [])
      end
```

2.5 Neue Datentypen

Typ-Abkürzungen Einfache Abkürzungen bestehender Typen lassen sich mit dem Schlüsselwort `type` definieren

```
type point = int * int
```

Nach einer solchen Deklaration kann man ein Ganzzahl-Tupel auch unter dem Namen `point` verwenden. Das spart einerseits Schreibarbeit, dient aber natürlich gleichzeitig der Abstraktion und der Übersicht.

Datentypen Komplexere Datentypen lassen sich in ML auch sehr einfach definieren. Man benutzt dazu das Schlüsselwort `datatype`. Eine abstrakte Definition ist:

```
datatype <name> = <constructor1> of <type1> | <constr2> of <type2> | ...
```

Einen Wert dieses neuen Typs erzeugt man, indem man einen der Konstruktoren vor einen entsprechenden Wert schreibt. Zum Beispiel:

```
datatype Shape = Circle of int | Rectangle of int * int;

val shape1 : Shape = Circle 5;
val shape2 : Shape = Rectangle (5,6);
```

Funktionen auf diesem neuen Datentyp lassen sich dann z.B. folgendermaßen definieren:

```
fun area (Circle r) = 2*3*r*r
  | area (Rectangle (w, h)) = w*h;
```

Records Oft werden für neue Datentypen auch Records verwendet, daß sind Sammlungen von Werten, die über einem bestimmten Namen gesetzt und gelesen werden können. Der Typ eines Records besteht aus den Namen und Typen der Felder; z.B.

```
{firstname : string, lastname : string}
```

Einen Wert dieses Typs erzeugt man, indem man jedem Feld einen Wert zuweist:

```
val me = {firstname = "David", lastname = "Frese"}
```

Und Funktionen auf diesem Typ kann man auf verschiedene Arten definieren:

```
fun fullname {firstname, lastname} = firstname ^ " " ^ lastname;
fun fullname2 (n : name) = (#firstname n) ^ " " ^ (#lastname n);
```

3 Threads

Wenn ein ML-Programm gestartet wird besteht es zunächst einmal aus einem einzigen Thread. Weitere Threads können von diesem anfänglichen Thread nun mit Hilfe des `spawn` Privitivums gestartet werden. Es hat den Typ

```
val spawn :(unit -> unit) -> thread_id
```

Um einen neuen Thread zu starten muß man also der `spawn` Funktion einen Thunk übergeben, der seinerseits nichts zurückgibt. Der Rückgabewert der `spawn` Funktion ist ein ID des gerade erzeugten Threads, die ihn eindeutig identifiziert. Der Aufruf von `spawn` kehrt nun sofort zurück und die Ausführung des Threads geht ganz normal weiter. Der übergebene Thunk wird parallel dazu bis zum Ende seines Rumpfes, oder einer frühzeitigen Beendigung ausgeführt. Frühzeitig beenden kann man einen Thread mithilfe der Funktion

```
val exit : unit -> 'a
```

die keinen Parameter erwartet und einen parametrisierten Rückgabewert hat, da sie ohnehin niemals zurückkehrt. Threads sind in ML sehr einfache Datenstrukturen die wenig Speicher und Resourcen verbrauchen. Daher kann man in einem CML-Programm, im Gegensatz zu andern Programmierspracehn, sehr freizügig mit dem Benutzen von Threads umgehen. In größeren Programmen sind viele Hundert Threads keine Seltenheit.

4 Channels

Um mehrere Threads sinnvoll einsetzen zu können gibt es in Concurrent ML die sogenannten Channels. Sie geben zwei oder mehreren Threads die Möglichkeit Daten untereinander auszutauschen. Die wichtigste Art dieser Kommunikation funktioniert über *synchronous message passing on typed channels*. Dazu benötigt man zunächst einmal eine neue Art von Wert - einen Channel. Der parametrisierte Typ

```
type 'a chan
```

wird benutzt um neue Typen von Channels zu erzeugen. Das `'a` steht dabei für den Typ von Werten die über den Channel übertragen werden sollen. Desweiteren sind die zwei wichtigsten Funktionen für die Kommunikation `recv` und `send`:

```
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

Ein Channel kann also nur Werte eines bestimmten Typs übertragen. Deswegen heißen sie *typed channels*. *Synchonous message passing* bedeutet hingegen, daß z.b. die Anwendung der `recv` Funktion auf einen Channel erst dann zurückkehrt, wenn ein anderer Thread die `send` Funktion auf den selben Channel anwendet. Das gleiche gilt natürlich auch wenn ein Thread versucht zu senden, aber noch kein anderer Thread den Wert

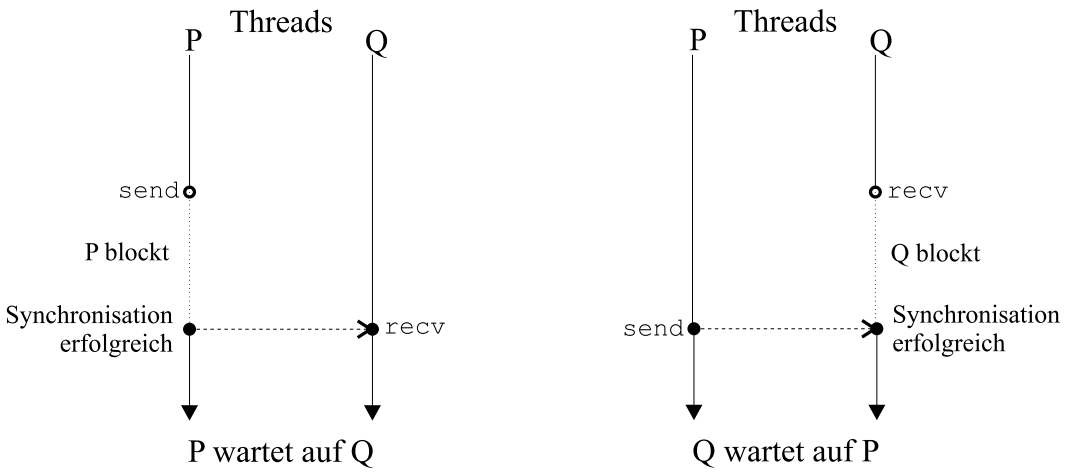

Abbildung 1: Synchronisierung mir `send` und `recv`

empfangen will. Man spricht daher auch von einer Synchronisierung der beiden beteiligten Threads. Eine graphische Darstellung dieser Synchronisierung ist in Abbildung 1 abgedruckt.

Jetzt braucht man noch eine Funktion die einen Channel erzeugt. Diese Funktion ist folgendermaßen definiert:

```
val channel : unit -> 'a chan
```

4.1 Beispiel: Sieb des Eratosthenes

Als einfaches Beispiel für den Gebrauch von Channels und Threads ist ein Channel aus dem man Primzahlen auslesen kann. Dazu benutze ich das sogenannte *Sieb des Eratosthenes*. Das Prinzip besteht darin, aus allen natürlichen Zahlen (ab 2) die Vielfachen von Primzahlen (außer der Primzahl selbst) herauszufiltern. Übrig beleiben dann automatisch die Primzahlen selbst. In CML realisiert man dies durch mehrere Threads, die verschiedene Aufgaben erledigen, und eine Verkettung dieser Threads durch Channels. Man braucht zuerst einen Thread, der fortwährend versucht eine ganze Zahl nach der anderen in einen Channel zu senden. Dann benötigen wir noch je einen Thread für jede bisher gefundene Primzahl, der Zahlen aus einem Channel ausliest und nur Zahlen weiterschickt, die nicht Vielfache dieser Primzahl sind. Am letzten Channel braucht man dann nur noch die Primzahlen auszulesen. Eine Veranschaulichung des Siebs ist in Abbildung 2 dargestellt.

Implementation Zunächst brauchen wir einen Channel, der alle natürlichen Zahlen, angefangen mit `init`, liefert.

```
fun counter init = let
```

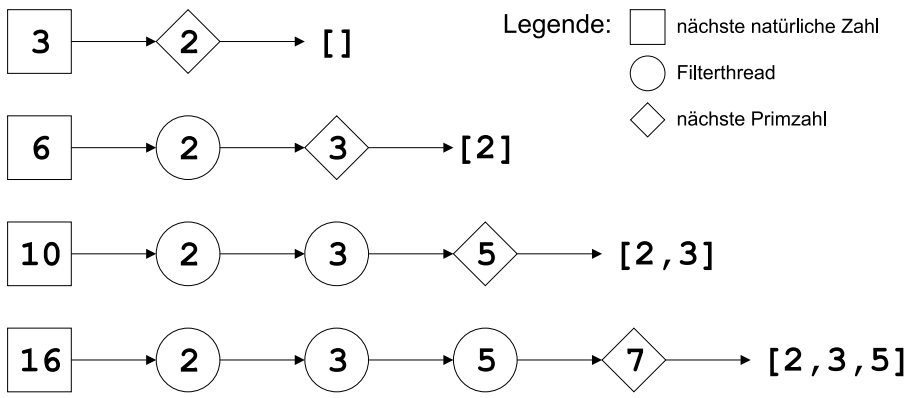

Abbildung 2: Grafische Darstellung des Siebs des Eratosthenes

```
      val outCh = channel()
      fun loop n = ( send(outCh, n); loop n+1 )
      in
        spawn ignore( loop init );
        outCh
      end
```

Die Funktion `filter` erzeugt einen neuen Channel, aus dem nur noch Zahlen ausgelesen werden können, die kein Vielfaches von `p` sind:

```
fun filter (p, inCh) = let
      val outCh = channel()
      fun loop () = let
            val i = recv inCh
            in
              if ((i mod p) <> 0) then send (outCh, i) else ();
              loop ()
            end
      in
        spawn loop;
        outCh
      end
```

Jetzt müssen wir nur noch gewährleisten, daß für jede neue Primzahl ein neuer filter-Thread erzeugt, und in die Kette eingehängt wird. Dazu benutzen wir folgende Prozedur:

```
fun sieve () = let
      val primes = channel()
      fun head ch = let
           val p = recv ch
```

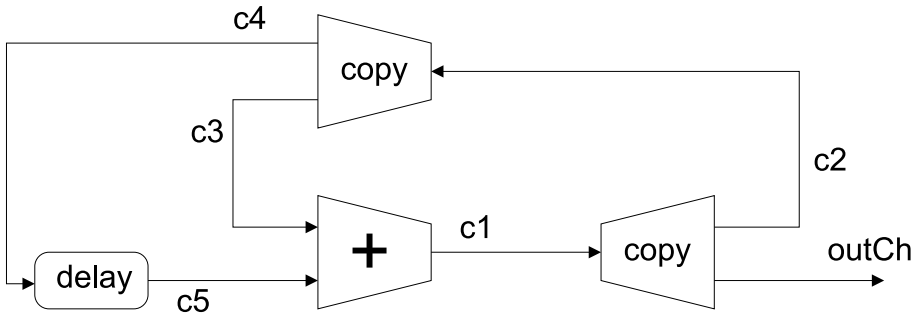

Abbildung 3: Netzwerk zur Berechnung der Fibonacci-Folge

```
        in
          send (primes, p);
          head (filter (p, ch))
        end
    in
      spawn (fn () => head (counter 2));
      primes
    end
```

Die Funktion `sieve` gibt nun also also einen Channel zurück, aus dem eine Primzahl nach der anderen ausgelesen werden kann. Diese können wir z.B. noch mit folgender Funktion auslesen:

```
fun primes n = let
      val ch = sieve ()
      fun loop (0, l) = rev l
        | loop (i, l) = loop (i-1, (recv ch)::l)
      in
        loop (n, [])
      end
```

5 Selective communication

Man kann Threads und Channels in ML auch dazu benutzen um Algorithmen zu berechnen. Dazu verbindet man mehrere Channels zu einem ganzen Netzwerk, indem man sie mit verschiedenen Threads verknüpft. Diese Threads können dann z.B. zwei Channels addieren oder eine Channel duplizieren. Als Beispiel betrachten wir die Fibonacci-Zahlen. In Abbildung 3 ist ein Netzwerk abgebildet, daß die Fibanacci-Zahlenfolge berechnen soll. Das Problem, daß man mit solchen Netzwerken bekommen kann, ist daß die festgelegte Reihenfolge der Kommunikation dazu führen kann, daß das ganze Netzwerk blockiert. Beispielsweise könnte es passieren, daß der `copy`-Thread zuerst versucht

über Kanal `c4` zu senden und daher blockiert ist, bis `delay` den Wert empfängt. Wenn aber der `+`-Thread zuerst von Channel `c3` empfangen will, dann könnte das Netzwerk stillstehen.

Die Lösung für dieses Problem ist die sogenannte *selective communication*. Dazu stellt CML zunächst einmal Funktionen bereit, die eine Kommunkation noch nicht wirklich durchführen, sondern nur vorbereiten und in ein *Event* umwandeln:

```
val sendEvt : ('a chan * 'a) -> unit event
val recvEvt : 'a chan -> 'a event
```

`'a event` ist der Typ einer synchronen Operation, die einen Wert vom Typ `'a` zurückgibt, wenn man sie synchronisiert. Tatsächlich sind die bekannten Funktionen `send` und `recv` einfach mithilfe der `sync`-Funktion definiert:

```
val sync : 'a event -> 'a
val send = sync o sendEvt
val recv = sync o recvEvt
```

Die zentrale Rolle spielt die `select`-Funktion. Sie hat den Typ

```
val select = sync o choose
val choose : 'a event list -> 'a event
```

Sie realisiert die nichtdeterministische Auswahl von mehreren möglichen Kommunikationen. D.h. aus der Liste der Events wird diejenige ausgewählt, die als erste realisiert werden kann - daß also an der anderen Seite des Channels eine passende Funktion aufgerufen wurde. Falls mehrere Kommunikationen gleichzeitig ausgeführt werden könnten, wird eine von ihnen nichtdeterministisch ausgewählt.
Damit kann man das Netzwerk also nun so programmieren, daß es nicht so leicht blockieren kann. Um die Implementation etwas einfacher zu machen, gibt es noch die Funktion `wrap`:

```
val wrap : ('a event * 'a -> 'b) -> 'b event
```

Sie liefert ein Event, daß synchnonisiert werden kann, sobald das Event, daß als erster Parameter übergeben wurde, synchronisiert werden kann. Außerdem steckt es das Ergebnis dieser Kommunikation noch in die übergebene Funktion.
Die Implementation des Addierer-Threads kann nun folgendermaßen aussehen:

```
fun add (inCh1, inCh2, outCh) =
      forever () (fn () => let
        val (a, b) = select [
                wrap (recvEvt inCh1, fn a => (a, recv inCh2)),
                wrap (recvEvt inCh2, fn b => (recv inCh1, b))
              ]
        in
          send (outCh, a + b)
        end)
```

Beim Aufruf von `add` mit den beiden Eingangs-Channels wird mit der `forever`-Funktion ein neuer Thread gestartet. Denn die eingebaute Funktion `forever` ist folgendermaßen definiert:

```
fun forever init f = let
      fun loop s = loop (f s)
      in
        ignore (spawn (fn () => loop init))
      end
```

Sie startet einen neuen Thread der die Funktion `f` zuerst mit `init` anwendet und dann fortwährent den Rückgabewert erneut in `f` reinsteckt.
Innerhalb von `add` werden nun zunächsteinmal die beiden Eingangswerte `a` und `b` bestimmt. Dazu wird der Eingangs-Channel ausgewählt, der zuerst einen Wert liefern kann, und dann noch der zweite Wert vom anderen Channel gelesen. Danach wird die Summe dieser beiden Zahlen an den Ausgangs-Channel gesendet.

Die Implementation des `copy`-Threads sollte man genauso sorgfältig programmieren. Sie könnte so aussehen:

```
fun copy (inCh, outCh1, outCh2) = forever () (fn () => let
     val x = recv inCh
     in
       select [
         wrap (sendEvt (outCh1, x), fn () => send (outCh2, x)),
         wrap (sendEvt (outCh2, x), fn () => send (outCh1, x))
       ]
     end)
```

Auch hier wird derjenige Ausgangs-Channel zuerst beliefert, der zuerst dazu bereit ist. Den `delay`-Thread kann man allerdings ganz einfach realisieren, weil er nur einen Eingang und eine Ausgang hat:

```
fun delay init (inCh, outCh) =
     forever init (fn NONE => SOME(recv inCh)
                    | (SOME x) => (send(outCh, x); NONE))
```

Hier wurde der Datentyp `'a option` verwendet. Er hat zwei Konstruktoren - `NONE` und `SOME`. Eine option die man mit `NONE` erzeugt ist leer, und eine mit `SOME` erzeugte, kann einen bestimmten Wert enthalten. Er dient hier dazu zwischen den zwei verschiedenen Zuständen *Den nächsten Wert lesen* und *Wert x weitergeben* zu unterscheiden. Daher ist die Überführungsfunktion auch für diese beiden Möglichkeiten definiert. Wenn der Zustand `NONE` vorliegt, wird in den Zustand `SOME` mit dem vom Eingangs-Channel gelesenen Wert übergegangen. Und wenn man im Zustand `SOME` mit Wert x ist, dann wird dieser an den Ausgangs-Channel gesendet, und in den Zustand `NONE` übergegangen.

Jetzt fehlt nur noch der Channel der uns die Fibonacci-Folge liefert und der Code der alle Threads miteinander verbindet. Dazu benutzen wir die Funktion `mkFibNetwork`:

```
fun mkFibNetwork () = let
     val outCh = channel()
     val c1 = channel() and c2 = channel() and c3 = channel()
     val c4 = channel() and c5 = channel()
     in
       delay (SOME 0) (c4, c5);
       copy (c2, c3, c4);
       add (c3, c5, c1);
       copy (c1, c2, outCh);
       send (c1, 1);
       outCh;
     end
```

6 Quellen

Concurrent Programming in ML von John H. Reppy,
Cambridge University Press, ISBN 0-521-48089-2.