# Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

# Erzeuger-Verbraucher-Architektur mit Coroutinen

```
val buf = ref 0

fun produce (n,consumer:state) =
 (buf := n; produce (n+1,resume consumer))

fun consume (producer:state) =
 (print (Int.toString (!buf));consume (resume producer))
```

- `resume`-Aufrufe implementieren die Übergabe der Kontrolle zwischen dem Erzeuger und dem Verbraucher.
- Bei der Übergabe der Kontrolle muss der aktuelle Zustand des Aufrufenden mit übergeben werden.
- Der Zustand einer Coroutine ist eine Continuation.

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Implementierung der Kontrollübergabe, erster Versuch:

```
val buf = ref 0

fun resume k = callcc (fn k1 => throw k k1)

fun produce (n,consumer) =
 (buf := n; produce (n+1,resume consumer))

fun consume (producer) =
 (print (Int.toString (!buf));consume (resume producer))
```

- Die Typ-Inferenz für `produce` liefert (☞ Übung):

$$\{'a = 'a \text{ cont cont}$$

- Lösung hinsichtlich Typ-Überprüfbarkeit: definiere `state` rekursiv

```
datatype state = S of state cont
```

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Implementierung der Kontrollübergabe, zweiter Versuch:

```
val buf = ref 0

datatype state = S of state cont

fun resume (S k) = callcc (fn k1 => throw k (S k1))

fun produce (n,consumer:state) =
 (buf := n; produce (n+1,resume consumer))

fun consume (producer:state) =
 (print (Int.toString (!buf));consume (resume producer))
```

$\Longrightarrow$ produce and consume sind typbar (☞ Übung):

```
val produce = fn : int * state -> 'a
val consume = fn : state -> 'a
```

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Äquivalent dazu (durch Inlining von `resume`):

```
val buf = ref 0

datatype state = S of state cont

fun produce (n,S cons) =
  (buf := n;
   produce (n+1,callcc (fn k => throw cons (S k))))

fun consume (S prod) =
  (print (Int.toString (!buf));
   consume(callcc (fn k => throw prod (S k))))
```

# Erzeuger-Verbraucher-Architektur mit Coroutinen

Anfang:

```
val buf = ref 0

datatype state = S of state cont

fun produce (n,S cons) =
 (buf := n;
  produce (n+1,callcc (fn k => throw cons (S k))))

fun consume (S prod) =
 (print (Int.toString (!buf));
  consume(callcc (fn k => throw prod (S k))))

fun run () = consume (callcc (fn k => produce (0, S k)))
```

# Continuations-Anwendung: Threads

- Mit zunehmender Anzahl Coroutinen in einem Programm wird die explizite Übergabe der Kontrolle mühsam und unübersichtlich.
- Besser: **Threads**
  - flexibler und modularer Ansatz des Kontrollflusses
  - asynchrone Events können auch behandelt werden
- Eine Coroutine **Scheduler** koordiniert die verschiedenen Threads.
- Threads sind Coroutinen des Schedulers.

# Continuations-Anwendung: Threads

- Scheduler-Funktionalität:
  - verwaltet eine Schlange `ready` von Threads
  - wenn aufgerufen (via `dispatch`), wählt einen Thread und setzt diesen fort
  - Ein Thread ist eine `unit cont`
- Definition eines Typ-Synonyms

```
type thread = unit cont
```

# Continuations-Anwendung: Threads

## Scheduler-Funktionalität:

```
exception NoMoreThreads
type thread = unit cont

val ready : thread Queue.queue = Queue.mkQueue ()

fun dispatch () =
  throw (Queue.dequeue ready) ()
  handle Queue.Dequeue => raise NoMoreThreads
```

# Continuations-Anwendung: Threads

## Threads-Funktionalität:

- `fork : (unit -> unit) -> unit`
  `fork f` erzeugt einen neuen Thread, der die Funktion $f$ ausführt. Der aufrufende Thread wird suspendiert.

- `yield : unit -> unit`
  `yield ()` übergibt die Kontrolle an den Scheduler

- `exit : unit -> 'a`
  `exit ()` beendet den aufrufenden Thread

# Continuations-Anwendung: Threads

## Threads-Funktionalität:

```
fun enqueue t = Queue.enqueue (ready,t)

fun exit () = dispatch ()

fun fork f =
  callcc (fn parentCont =>
              (enqueue parentCont;f(); exit()))

fun yield () =
  callcc (fn cont => (enqueue cont; dispatch ()))
```

# Erzeuger-Verbraucher-Threads

```
val buffer = ref 0

fun producer () =
  (buffer := !buffer + 1;yield();producer ())

fun consumer () =
  (print (Int.toString (!buffer));yield();consumer())

fun run () =
  (init ();fork consumer;producer ())

fun run2 () =
  (init ();fork consumer; fork producer; producer ())
```

# Threads: Pre-emption

- Unser Scheduler ist nicht **pre-emptive**!

- Pre-emption braucht Laufzeitsystem-Unterstützung: `Signals` in SMLofNJ:

```
val setHandler : (signal * sig_action) -> sig_action

eqtype signal

datatype sig_action = IGNORE | DEFAULT
  | HANDLER of (signal * int * unit cont) -> unit cont
```

- Installierung eines Handler für ein Signal:
  `setHandler(signal,HANDLER (fn (signal,n,k) => k'))`
  Unterbrochener Thread und Handler sind Coroutinen. Tritt `signal` auf, dann:
  1. Laufzeitsystem übergibt aktuelle Cont. `k` dem Handler;
  2. Handler liefert eine neue Cont. `k'`;
  3. Das Laufzeitsystem aktiviert `k'`.

# Threads: Pre-emption

- Idee:

```
Signals.setHandler
  (Signals.sigALRM,
   Signals.HANDLER (fn (_,_,k) =>
                       (enqueue k; dispatch())

val granularity = Time.fromMilliseconds 10

SMLofNJ.IntervalTimer.setIntTimer (SOME granularity)
```

- Zugriff auf gemeinsamen Datenstrukturen (z.B. `ready`) muss atomisch sein
- Signal Behandlung ist eine atomische Operation
- Continuations + Signals Handling $\Longrightarrow$ **Concurrent ML** = SML Erweiterung um Nebenläufigkeit

# Andere Anwendungen von Continuations

- Ausnahmen
- Lösungs-Generatoren
- Multi-Agent-Programmierung
- Iteratoren
- Backtracking
- andere benutzer-definierte Kontrollstrukturen

# Überblick

## 2. Modularisierung

Strukturen

Signaturen

Funktoren

# Motivation

- Bis jetzt haben wir alle Deklarationen in der Top-Level-Umgebung eingeführt
  - Abhängigkeit der Deklarationen diktiert Reihenfolge ihrer Einführung/Implementierung

  $\Longrightarrow$ keine unabhängige Entwicklung der Programmteile

  $\Longrightarrow$ separate Kompilierung nicht möglich

- Einkapselung zusammenhängender Deklarationen in Programmeinheiten (**Module**), die relativ unabhängig voneinander behandelt werden können.

# Strukturen

- In SML heißen die Module **Strukturen**. Eine Struktur wird zwischen **struct** und **end** eingeklammert:

```
struct
   type 'a Pair = 'a * 'a
   fun Pair(a,b) = (a,b)
   fun first (a,b) = a
   fun second (a,b) = b
end
```

# Strukturen

- In SML heißen die Module **Strukturen**. Eine Struktur wird zwischen **struct** und **end** eingeklammert:

```
struct
  type 'a Pair = 'a * 'a
  fun Pair(a,b) = (a,b)
  fun first (a,b) = a
  fun second (a,b) = b
end
```

- Eine Struktur kann zu einem Struktur-Bezeichner gebunden werden:

```
structure Pairs =
struct
  type 'a Pair = 'a * 'a
  fun Pair(a,b) = (a,b)
  fun first (a,b) = a
  fun second (a,b) = b
end
```

# Signaturen

Ähnlich wie andere Werte...

- eine Struktur hat einen Typ, genannt **Signatur**
- Auf die Eingabe einer Struktur antwortet der Compiler mit ihrer Signatur:

```
structure Pairs :
    sig
        type 'a Pair = 'a * 'a
        val Pair : 'a * 'b -> 'a * 'b
        val first : 'a * 'b -> 'a
        val second : 'a * 'b -> 'b
    end
```

# Sichtbarkeit

- Die Definitionen innerhalb der Struktur sind außerhalb (insb. top-level) zunächst nicht sichtbar:

```
- first;
stdIn:41.1-41.6 Error: unbound variable or constructor: first
```

- Die Namen innerhalb einer Struktur sind ansprechbar über den Namen der Struktur:

```
- Pairs.first;
val it = fn : 'a * 'b -> 'a
```

# Strukturen als Namespaces

- Strukturen helfen u.a. zur Einführung von Namespaces zur Vermeidung von Namenskonflikten.

- Bsp.: Funktionen für verschiedene Typen mit dem gleichen Namen definieren:

```
structure Triples =
  struct
    datatype 'a Triple = Triple of 'a * 'a * 'a
    fun first (Triple abc) = #1 abc
    fun second (Triple abc) = #2 abc
    fun third (Triple abc) = #3 abc
  end

- Triples.first;
val it = fn : 'a Triples.Triple -> 'a
```

# Öffnen von Strukturen

- Um nicht immer den Strukturnamen verwenden zu müssen, kann man auch alle Definitionen einer Struktur auf einmal sichtbar machen:

```
- open Pairs;
opening Pairs
   type 'a Pair = 'a * 'a
   val Pair : 'a * 'b -> 'a * 'b
   val first : 'a * 'b -> 'a
   val second : 'a * 'b -> 'b
- Pair;
val it = fn : 'a * 'b -> 'a * 'b
- Pair (4,3);
val it = (4,3) : int * int
```

# Öffnen von Strukturen

- Strukturen öffnen sollte man aber möglichst nur lokal tun, um Namenskonflikte mit anderen offenen Modulen zu vermeiden:

- Einschränkung der Sichtbarkeit der Deklarationen:
  - Für Ausdrücke:

```
let open struct
in expr end
```

```
let open Real in (toString 2.2)^"f" end;
val it = "2.2f" : string
```

  - Für Definitionen:

```
local open struct
in expr end
```

```
local open Real
in fun Real2String r = (toString r)^"f" end;
val Real2String = fn : real -> string
```

# Geschachtelte Strukturen

- Strukturen können selbst auch wieder Strukturen enthalten:

```
structure Quads =
  struct
    structure Pairs =
      struct
        type 'a Pair = 'a * 'a
        fun Pair(a,b) = (a,b)
        fun first(a,_) = a
        fun second(_,b) = b
      end
    type 'a Quad = 'a Pairs.Pair Pairs.Pair
    fun Quad (a,b,c,d) =
        Pairs.Pair (Pairs.Pair(a,b),Pairs.Pair(c,d))
    fun first q = Pairs.first(Pairs.first q)
    fun second q = Pairs.second(Pairs.first q)
    fun fourth q = Pairs.second(Pairs.second q)
  end
```

# Geschachtelte Strukturen

```
- Quads.Quad(1,2,3,4);
val it = ((1,2),(3,4)) : (int * int) * (int * int)
- Quads.Pairs.first;
val it = fn : 'a * 'b -> 'a
```

# Signaturen...

- erscheinen nicht nur als Compiler-Antworten;
- können eine erwünschte Funktionalität spezifizieren;
- können zu einem Signatur-Bezeichner gebunden werden.
- Bsp.: Counter-Funktionalität:

```
signature CountSig =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end
```

# Signatur-Einschränkungen

- Mit Hilfe von Signaturen kann man einschränken, was ein Modul nach außen exportiert.

- Bsp.:

```
structure Count =
  struct
    val cnt = ref 0
    fun setCounter n  = cnt := n
    fun getCounter () = !cnt
    fun incCounter () = cnt := !cnt+1
  end

- !Count.cnt;
val it = 0 : int
```

Der Zähler ist nach außen sichtbar, zugreifbar und sogar veränderbar.

# Signatur-Einschränkungen

- Mit einer Signatur kann man eine eingeschränkte Schnittstelle (*view*) einer Struktur ereugen:

```
- structure SafeCount = Count:CountSig;
structure SafeCount : CountSig
- open SafeCount;
opening SafeCount
   val setCounter : int -> unit
   val incCounter : unit -> unit
   val getCounter : unit -> int
- SafeCount.cnt;
stdIn:1.1-1.14 Error: unbound variable or constructor:cnt in path SafeCount.cnt
```

- Die Signatur bestimmt also, welche Definitionen exportiert werden (`CountSig` enthält den Counter selbst nicht).

# Signatur-Einschränkungen

- Eine eingeschränkte Schnittstelle zuweisen geht auch schon direkt bei der Definition:

```
structure Count1 : Count =
  struct
    val cnt = ref 0
    fun setCounter n  = cnt := n
    fun getCounter () = !cnt
    fun incCounter () = cnt := !cnt+1
  end

- !Count1.cnt;
stdIn:196.2-196.12 Error: unbound variable or constructor: cnt in path Count1.cnt
```

# Signatur-Einschränkungen

- Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein: Dadurch werden deren Typen spezialisiert:

```
signature A1 = sig val f : 'a -> 'b -> 'b end
signature A2 = sig val f : int -> char -> int end
structure A = struct fun f x y = x end
- structure A1 = A:A1;
stdin: Error: value type in structure doesn't match signature spec name: f
   spec: 'a -> 'b -> 'b
   actual: 'a -> 'b -> 'a
- structure A2 = A:A2;
structure A2 : A2
- A2.f;
val it = fn : int -> char -> int
```

# Information Hiding

- Aus Gründen der Modularität möchte man oft nicht, dass die Struktur der Typen, die ein Modul zur Verfügung stellt, nach außen bekannt ist. Beispiel:

```
structure ListQueue =
  struct
    exception EmptyQueue
    type 'a Queue = 'a list
    val empty = nil
    fun isempty nil = true
      | isempty _   = false
    fun enqueue nil y = [y]
      | enqueue (x::xs) y = x :: enqueue xs y
    fun dequeue nil = raise EmptyQueue
      | dequeue (x::xs) = (x,xs)
  end
```

# Information Hiding

- Will man verstecken, dass eine Queue eine Liste ist, kann man das mit einer Signatur:

```
signature QueueSig =
  sig
    exception EmptyQueue
    type 'a Queue
    val empty : 'a Queue
    val isempty : 'a Queue -> bool
    val enqueue : 'a Queue -> 'a -> 'a Queue
    val dequeue : 'a Queue -> 'a * 'a Queue
  end
```

# Information Hiding

- Das Einschränken per Signatur genügt nicht, um die wahre Natur des Typs `Queue` zu verschleiern.

```
- structure Queue = ListQueue : QueueSig;
structure Queue : QueueSig

- open Queue;
opening Queue
  exception EmptyQueue
  type 'a Queue = 'a list
  val empty : 'a Queue
  val isempty : 'a Queue -> bool
  val enqueue : 'a Queue -> 'a -> 'a Queue
  val dequeue : 'a Queue -> 'a * 'a Queue
- isempty nil;
val it = true : bool
```

# Information Hiding

- Um die Implementierung der Datentypen zu verstecken, muss man das so genannte *opaque signature matching* (`:>` anstatt `:`) verwenden:

```
- structure HiddenQueue = ListQueue :> Queue;
structure HiddenQueue : Queue
```

- Durch die Verwendung von `:>` werden alle exportierten Typen, deren Definition nicht explizit in der Signatur steht, abstrahiert.

# Information Hiding

```
- open HiddenQueue;
opening HiddenQueue
  exception EmptyQueue
  type 'a Queue
  val empty : 'a Queue
  val isempty : 'a Queue -> bool
  val enqueue : 'a Queue -> 'a -> 'a Queue
  val dequeue : 'a Queue -> 'a * 'a Queue
```

# Information Hiding

- Die Struktur `HiddenQueue` ist ein **abstrakter Datentyp**:

```
- isempty empty;
val it = true : bool

- isempty nil;
stdIn:301.1-301.12 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z Queue
  operand: 'Y list
  in expression:
    isempty nil
```

# Funktoren: Motivation

- Möglichst unabhängige Entwicklung: Ein Modul hängt nur von den Signaturen anderer Module und nicht von ihren Implementierungen.

- Bsp.: ein Parser kann einen beliebigen Lexer benutzen, dessen Signatur eine Funktion `next` implementiert, die das nächste Token liefert.
  - Problem: Code Duplizierung
    - `struct MyParser = ... MyLexer.next() ... end`
    - `struct YourParser = ... YourLexer.next() ... end`
  - Lösung: parametrisierte Module = **Funktoren**

# Funktoren: Motivation

- Ein Funktor bekommt als Parameter eine Reihe von Werten, Typen oder ganzen Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Parameter des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.
- Bsp:

```
functor Parser(Lexer :
                  sig val next: unit -> int end) =
  struct ... Lexer.next() ... end
```

Instantiiere Parser mit einem bestimmten Lexer = **Funktor-Anwendung**:

- `structure MyParser = Parser(MyLexer)`
- `structure YourParser = Parser(YourLexer)`

# Funktoren: Motivation

- Funktoren unterstützen **generische** Datenstrukturen/Algorithmen:
  - Signatur-Constraints für Argumente erfassen das **Generische**
  - Das Spezifische/Unwesentliche steckt in den übergebenen Strukturen, die mehr als von der Signatur verlangt implementieren können

```
functor SortedList
          (Element:sig val le : 'a*'a -> bool end )=
  struct ... if Element.le (x,y) ... end
```

# Funktoren: Vorgehensweise

- Man legt zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
signature Enum =
  sig
    type Enum
    val null : Enum
    val incr : Enum -> Enum
  end

signature Counter =
  sig
    type Counter
    val setCounter : Counter -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> Counter
  end
```

# Funktoren

```
functor GenCounter (structure Enum : Enum) : Counter =
  struct
    open Enum
    type Counter = Enum
    val cnt = ref null
    fun setCounter x = cnt := x
    fun incCounter () = cnt := incr(!cnt)
    fun getCounter () = !cnt
  end
```

- Die Anwendung des Funktors `GenCounter` auf eine Struktur mit der Signatur `Enum` erzeugt eine neue Struktur mit der Signatur `Counter`.