Programmiersprachen

Contents

1 Implementation of Languages

1.1 Activation Record

Ein *Activation Record* oder *Frame* beinhaltet alle wichtigen Informationen (u.a. lokale Variablen), die notwendig sind, eine Routine auszuführen. Am

Offset 0 steht der *Return Pointer* (wohin nach Beendigung der Routine gesprungen werden soll) und am Offset 1 der *dynamic Link* (s.u.).

1.2 Rekursion

Durch rekursive Routinen können Activation Records nicht mehr an fixen Adressen stehen, welche durch den Compiler mittels *static Allocation* ausgesucht worden sind. Man benötigt einen Stack.

1.3 Dynamic Link

Der *dynamic Link* ist ein Pointer zur Basisadresse des Activation Records des Aufrufers.

GCC hat ein Flag `--fomit-frame-pointer`, durch welches der dynamic Link nicht mehr gespeichert wird (dieser kann berechnet werden).

1.4 Static Link

Für *geschachtelte Routinendefinitionen* werden *static Links* benötigt. Ein static Link ist ein Pointer zum Activation Record der statisch umschließenden Routine.

Jede Variable wird zur Übersetzungszeit an das Paar $\langle d, o \rangle$ gebunden, wobei d (*distance*) die Anzahl der statischen Links zum Block mit der Variablendeklaration und o den Offset im korrespondierenden Activation Record darstellt. Die Adresse einer Variable ist zur Laufzeit `D[`$fp(d) + o$`]`, wobei $fp(d)$ den Frame Pointer des Activation Records der Distanz d berechnet:

$$f(d) = \begin{cases} \texttt{D[0]} & \text{wenn } d = 0 \\ \texttt{D[}f(d-1) + \text{static link offset} & \text{wenn } d > 0 \end{cases}$$

Pascal und Modula-2 erlauben geschachtelte Routinendefinitionen, aber keine geschachtelten Compound-Blöcke. Ada erlaubt beides.

1.5 Heap

Wegen Nebenläufigkeit in modernen Sprachen wachsen Stack und Heap oft nicht mehr gegeneinander (1 Heap, viele Stacks).

1.6 Parameterübergabe

Call by Reference Übergabe der Addresse des l-Wertes des aktuellen Parameters.

Call by Copy Der Parameter ist eine lokale Variable des Aufgerufenen.

Call by Value l-Wert wird in die Variable des Aufgerufenen kopiert.

Call by Result Resultat wird in die Variable des Aufrufers zurückkopiert (in den Activation Record).

Call by Value-Result Call by Value beim Aufruf und Call by Result beim Retournieren.

Call by Name Der formale Parameter wird textuell durch den aktuellen Parameter ersetzt.

1.6.1 Call by Reference

- Wenn der aktuelle Parameter im Aufrufer eine Variable ist, dann übergib den l-Wert.
- Wenn der aktuelle Parameter im Aufrufer ein Call-by-Referenz-Parameter ist, dann übergib den r-Wert.

1.6.2 Call by Value-Result

Folgende Szenarien können bei Call by Value-Result unterschiedliche Resultate zu Call by Referenz liefern.

1. Zwei formale Parameter sind Aliase.

   Aktuelle Parameter: `a[i], [j]`, wobei `i` = `j`
   Formale Parameter: `x, y`
   Routinenkorpus: `x=0; y++;`

2. Ein formaler Parameter und eine für beide sichtbare nichtlokale Variable sind Aliase.

   Aktuelle Parameter: `a`
   Formale Parameter: `x`
   Routinenkorpus: `a=1; x=x+a;`

1.6.3 Call by Name

Call by Name ist ein mächtiges Instrument und spielt eine Rolle im lambda-Kalkül und ALGOL 68. Es ist nicht leicht anzuwenden, weil es in manchen Situationen zu unvorhergesehenen Effekten führen kann (siehe Beispiel zur Routine `swap`). C-Makros nicht das gleiche wie Call by Name, was sich zeigen kann, wenn es einen Konflikt zwischen einem Namen einer nichtlokalen Variablen im Routinenkorpus und einer lokalen Variable im Aufruf gibt. Hier stellt sich die Frage, auf welchen Speicherbereich sich der Name mit Konflikt im Routinenkörper beziehen soll. Bei Call by Value ist es der der nichtlokalen Variable, bei C-Makros der übergebene Parameter.

2 Structuring the Computation

2.1 Exceptions

2.1.1 Exceptions in Ada

Es gibt vier vordefinierte Exceptions:

`ConstraintError` Verletzung einer Laufzeitregel, bspw. Array Idx out of bounds oder Division durch 0.

`ProgramError` Verletzung einer Sprachregel, bspw. wenn ein kein return-Statement einen Wert zum Aufrufer zurückliefert.

`StorageError` Fehler bzgl. Speicherplatzverfügbarkeit, könnte bspw. beim Aufruf von `new` passieren.

`TaskingError` Fehler bzgl. Tasksystem.

Beispiel einer nutzerdefinierten Exception und deren Verwendung:

```
Help: exception;
raise Help;
```

Exception Handler können nach dem Keyword `exception` einem *Subprogram Body*, *Package Body* oder Block angehängt werden. Beispiel:

```
begin -- block with exception handling
-- ...statements...
exception when Help => -- ...statements...
  when Constraint_Error => -- ...statements...
  when others => -- ...statements...
end;
```

Nicht abgefangene Exceptions werden propagiert, es sei denn, eine solche wird in einem *task body* geworfen. Dann terminiert der Task abnormalerweise.

Jede deklarierte Exception kann zur Übersetzungszeit an einen eindeutigen internen Exceptionnamen gebunden werden.

Es wurden zwei Möglichkeiten für Exception Handling besprochen:

1. Jeder Activation Record beinhaltet einen Pointer zu einer statischen *Handler Table*. Jeder Eintrag dieser Tabelle verbindet einen internen Exceptionnamen mit dem korrespondieren Handlerkörper.
2. Es gibt eine globale Exceptiontabelle. Wenn eine Exception geworfen wird, muss über bspw. binäre Suche die richtige „kleine Tabelle" gefunden werden. D.h., wenn keine Exception aufkommt, verursacht diese Methode keine zusätzlichen Kosten, und wenn eine aufkommt, dann nur geringe zusätzliche Laufzeitkosten.

2.1.2 Exceptions in C++

Beliebige Daten können als Exception verwendet werden. Beispiel: `throw Help(MSG1);`. Durch mehrere catch-Klauseln können Exceptions nach Typ unterschieden werden.

Optional kann in einem Funkionskopf eine Liste an Exceptions angegeben werden, die propagiert werden drüfen, z.B. `void foo() throw(Help, Zerodevide);` (depricated). Wenn eine Exception, die nicht in der throw-Klausel angegeben ist, geworfen wird, wird `unexpected()` aufgerufen (depricated). Wird beim Weiterpropagieren nie ein passender Handler gefunden, wird `terminate()` (überschreibbar) aufgerufen.

2.1.3 Exceptions in Java

Java verwendet try-catch-finally-Blöcke. Propagierte Exceptions (außer `RunTimeException` und `Error`) müssen deklariert werden. Beispiel: `void foo() throws Help;`. Wenn im finally-Block eine Exception auftritt, unterdrückt diese die aus dem try-Block (sie wird nicht mehr weiter beachtet). Seit Java 7 gibt es automatisch schließende try-Blöcke für Objekte die `Closable` implementieren. Hier unterdrückt die Exception aus dem try-Block die eventuelle aus der schließenden Methode. Beispiel: `try (Reader r = new FileReader(path)) ...`

Exceptionhandling in Java ist sehr effizient. Das teuerste ist das erzeugen des Exception-Objekts.

2.1.4 Exceptions in ML

Definition:

```
exception Neg
```

Anwendung:

```
fun fact(n) =
  if n<0 then raise Neg
  else if n=0 then 1
    else n*fact(n-1)

fun fact_0(n) =
  fact(n) handle Neg => 0;
```

2.1.5 Exceptions in Eiffel

Exceptionhandling folgt in Eiffel der *Retry-Semantik*. Man handhabt Exceptions indem man den Programmcode korrigiert. Eine Routine kann eine rescue-Klausel besitzen, die abgearbeitet wird, wenn eine Exception im Routinenkorpus geworfen wird. Wenn in der rescue-Klausel die Instruktion `retry` erreicht wird, beginnt die Ausführung der ursprünglichen Routine erneut (idealerweise mit geändertem Programmzustand), ansonsten wird die Exception propagiert.

2.2 Nebenläufigkeit

2.2.1 Prozesse in Ada

Prozesse in Ada heißen *Tasks*. In der Deklaration eines Task Typs werden Entries angegeben, mit denen mit dem Task interagiert werden kann. Die Deklaration eines Task Typs ist eine Deklaration eines ADTs.

2.2.2 Synchronisation und Kommunikation

1. Semaphore

   ```
   P(s) := if s > 0
             then s = s - 1
   ```

```
          else suspend current process
V(s) := if there is process supended in sempahore
          then wake up process
          else s = s + 1
```

`P` wird vor dem Eintritt in einen kritischen Abschnitt aufgerufen, `V` vor dem Austritt.

2. Monitore und Signale

   (a) Monitore in Pascal

   Concurrent Pascal hat Monitore und Signale für Nebenläufigkeit eingeführt. Signale sind Synchronisationsprimitive; Monitore beschreiben ADTen in nebenläufigen Umgebungen. Die darunterliegende Implementierung garantiert, dass die Operationen, die die Datenstruktur verändern, gegenseitig ausschließend durcheführt werden. Zusammenarbeiten bzgl. Zugriff auf die geteilte Datenstruktur muss durch `delay` und `continue` gewährleistet werden. Es folgt ein Beispiel für Producer-Consumer:

```
type fifostorage =
monitor
    var contents: array [1. .n] of integer; {buffer contents}
        tot: 0. .n; {number of items in buffer}
        in, {position of item to be added next}
        out: 1. .n; {position of item to be removed next}
        sender, receiver: queue;
    procedure entry append (item: integer);
    begin if tot = n then delay (sender);
        contents [in] := item;
        in := (in mod n)+1;
        tot := tot + 1;
        continue (receiver)
    end;
    procedure entry remove (var item: integer);
    begin if tot = 0 then delay (receiver);
        item := contents[out];
        out := (out mod n) + 1;
        tot := tot - 1;
        continue (sender)
```

```
    end;
begin {initialization part}
    tot := 0; in := 1; out := 1
end
```

Monitor-Instanzen sind abstrakte Objekte, durch welche IPC und Synchronisierung koordiniert wird. Ein Programm mit zwei Prozessen (Consumer und Producer), das vom obigen Monitor gebrauch macht, könnte folgendermaßen aussehen:

```
const n = 20;
type fifostorage =   . . . as above . . .
type producer = process (storage: fifostorage);
var element: integer;
begin cycle
    . . .
    storage.append (element);
    . . .
    end
end;
type consumer = process (storage: fifostorage);
var datum: integer;
begin cycle
    . . .
    storage.remove (datum);
    . . .
    end
end;
var meproducer: producer;
    youconsumer: consumer;
    buffer: fifostorage;
begin
    init buffer, meproducer (buffer), youconsumer (buffer)
end
```

`init` allokiert den Speicher für die Variablen im Monitor und führt den Initialisierungscode aus. `append` und `remove` sind mit dem Keyword `entry` deklariert, was bedeutet, dass sie die einzigen exportierten Prozeduren sind, die Monitor-Instanzen manipulieren können. Die Operation `delay(sender)` suspendiert den ausführenden Prozess in der Queue `sender`. Dieser Prozess ver-

liert seinen exklusiven Zugriff auf die Datenstruktur des Monitors und die Ausführung wird aufgeschoben, bis ein anderer Prozess `continue(sender)` ausführt. Gleiches gilt für `receiver`. Im oberen Beispiel sind die Prozesse als nichtterminierende zyklische Aktivitäten beschrieben (`cylce...end`). `meproducer` und `youconsumer` sind als gebunden an `buffer` (vom Typ `fifostorage`) deklariert und werden als nebenläufige Prozesse vom init-Statement aktiviert.

(b) Monitore in Java

Gegenseitiger Ausschluss wird durch `synchronized` Methoden gewährleistet. Zu einem Zeitpunkt darf nur ein Thread auf eine synchronisierte Methode des Monitors zugreifen. Durch den Aufruf von `wait` suspendiert ein Thread, gibt den Monitor frei, und kann durch `notify` oder `notifyAll` geweckt werden. Beispiel:

```
public class IntBuffer100 {
    private int[] cont = new int[100];
    private int in = 0, out = 0, total = 0;
    public synchronized void append(int item) {
        while (total >= 100) try { wait(); }
                             catch(InterruptedException e){}
        cont[in] = item;
        total++;
        if (++in >= 100) in = 0;
        notifyAll();
    }
    public synchronized int remove() {
        while (total <= 0) try { wait(); }
                           catch(InterruptedException e){}
        int temp = cont[out];
        total--;
        if (++out >= 100) out = 0;
        notifyAll();
        return temp;
   }
}
```

3. Rendezvous in Ada

Rendezvous kann als high-level Mechanismus, der Synchronisierung und Kommunikation kombiniert, angesehen werden, wobei die Kommunikation auf dem konzeptuellen *Message Passing* Paradigma beruht. Beispiel:

```
task Buffer_Handler is   --task declaration
    entry Append (Item: in Integer);
    entry Remove (Item: out Integer);
end;
task body Buffer_Handler is --task implementation
    N: constant Integer := 20;
    Contents: array (1. .N) of Integer;
    In_Index, Out_Index: Integer range 1. .N := 1;
    Tot: Integer range 0. .N := 0;
begin loop
    select
        when Tot < N =>
            accept Append (Item: in Integer) do
                Contents (In_Index) := Item;
            end;
            In_index := (In_Index mod N)+1;
            Tot := Tot+ 1
    or
        when Tot > 0 =>
            accept Remove (Item: out Integer) do
                Item := Contents (Out_Index);
            end;
            Out_Index := (Out_Index mod N) + 1;
            Tot := Tot - 1;
    end select;
end loop;
end Buffer_Handler;
```

Entries können als *Ports* (Anschluss?) gesehen werden, durch welche ein Task eine Nachricht an einen anderen schicken, der diese Nachricht dann das accept-Statement akzeptieren kann. An dieser Stelle „treffen" sich dann die beiden Tasks. Der Aufrufer suspendiert beim Aufrufen eines Entries so lange, bis der Empfänger die Nachricht akzeptiert, bzw. der Empfänger suspendiert beim akzeptieren einer Nachricht so lange, bis das entsprechende Entry aufgerufen wird. Ein Task kann Nachrichten von mehreren Tasks akzeptieren, weshalb jeder

Entry potentiell eine Queue von Tasks mit gesendeten Nachrichten.

Bei einem Match wird/bleibt der Sender suspendiert, bis der *Accept Body* abgearbeitet ist. Mögliche Out-Parameter werden am Ende des Accept Bodys zum Sender zurückgegeben und beide arbeiten parallel weiter.

4. Protected Types in Ada

   TODO