Program Realisation 2

http://www.win.tue.nl/~hemerik/2IP20/

Lecture 8

Kees Hemerik
Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Feedback to T.Verhoeff@TUE.NL

Today's Topics

- Drag and Drop in Delphi
- Recursion in routine implementations

Drag and Drop: Terminology

- Drag: Move mouse while holding mouse button down
- Source: TControl where drag begins
- Drop: Release mouse button after drag
- Target: TControl where drag ends

  N.B. Possibly, Source = Target
- Program can control at which locations a drop is acceptable
- User can cancel drag by releasing mouse button in a place where the drop is not accepted
- Program can associate a DragImage with the cursor

Drag and Drop in Delphi

The following Events control drag and drop:

- OnMouseDown, calls BeginDrag(False) to initiate drag when property DragMode = dmManual (default)
- OnStartDrag, can create a DragObject
- OnDragOver, sets var parameter Accept
- OnDragDrop, only called when drop is successful
- OnEndDrag, always called, also when drag canceled (then Target = nil); can destroy the DragObject

Source Event Target Event

Steen der wijzen

Specificeer procedure $Miracle$ door:

procedure $Miracle$;
{ **pre** $true$
post $false$ }

Dan geldt voor *alle* Q en R

$\{\ Q\ \}\ Miracle\ \{\ R\ \}$

o.g.v.

- exportregel,
- preconditie versterken ($Q \Rightarrow true$),
- postconditie verzwakken ($false \Rightarrow R$)

Steen der wijzen: implementatie

Een (partieel!) correcte implementatie van $Miracle$ is:

procedure $Miracle$;
{ **pre** $true$
post $false$ }

begin
{ $true$ } $Miracle$ { $false$ }
end

Holy Grail (1)

Hier bevatten de routine implementatie's geen aanroep van zichzelf:

procedure *Holy*;
{ **pre** *true*
  post *false* }

  procedure *Grail*;
  { **pre** *true*
    post *false* }
    begin
      Holy
    end; { *Grail* }

  begin { *Holy* }
    Grail
  end; { *Holy* }

Er geldt: { Q } *Holy* { R } voor alle Q, R

Holy Grail (2)

Geneste procedure definities zijn niet nodig:

procedure *Grail*; **forward**;

procedure *Holy*;
{ **pre** *true*
  post *false* }
  begin
    Grail
  end; { *Holy* }

procedure *Grail*;
{ **pre** *true*
  post *false* }
  begin
    Holy
  end; { *Grail* }

Recursie

Statische aanroep-graaf van programma P:

- Knopen: routines gedefinieerd in P.
- Pijlen: $q \rightarrow r$ waarbij body van routine q een aanroep van routine r bevat.

Recursie: Cykel in aanroep-graaf (*syntactisch* fenomeen).

Directe recursie: Cykel met lengte 1, d.w.z.

> de body van een routine bevat een aanroep van zichzelf.

Indirecte recursie: Cykel met lengte > 1.

Wederzijdse recursie: Cykel met lengte 2.

Bewijsregel voor direct-recursieve body

Extra annotatie: **variante functie** VF:

> **procedure** $p(e, f\colon T$; **var** $x, y\colon T$ { ; **glob** g }); { **specvar** m ; **pre** U ; **post** V ; **vf** VF }

VF is partiële functie van toestandsruimte opgespannen door parameters e, f, x, y, g naar een **well-founded domein** $\mathcal{D}$ (bijv. $\mathbb{N}$).

Body S past bij specificatie van p, indien voor alle m en $C \in \mathcal{D}$ geldt

- $U \Rightarrow VF \in \mathcal{D}$
- |[**var** $e, f, x, y\colon T$; { $U \wedge VF = C$ }
  var $\ell\colon T$;
  S
  { V }
  ]|

waarbij de preconditie van iedere recursieve aanroep $p(E, F, v, w)$ in S impliceert

$$VF(e, f, x, y := E, F, v, w) < C$$

While-loop en recursie

Stel je hebt bewezen

$$\{\ Q\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\ R\ \}$$

met invariant P en variante functie VF.

Dan geldt

$$\{\ Q\ \}\ \mathit{WhileDo}\ \{\ R\ \}$$

met

procedure $\mathit{WhileDo}$;
{ **pre** P ; **post** $P \wedge \neg B$; **vf** VF }
begin
{ $P \wedge VF = C$ }
if B **then begin** { $P \wedge B \wedge VF = C$ }
S { $P \wedge VF < C$}
; $\mathit{WhileDo}$ { $P \wedge \neg B$ }
end { if }
{ $P \wedge \neg B$ }
end; { $\mathit{WhileDo}$ }

Ontaarde recursie (1)

procedure Nep;
{ **pre** true ; **post** true ; **vf** 0 }

begin
{ $VF = C$ }
if false **then** Nep
end; { Nep }

Syntactisch gezien is er sprake van recursie.

Maar bij executie doet het zich niet voor.

Ontaarde recursie (2)

procedure $Abs(e\colon\ real;\ \textbf{var}\ x\colon\ real)$;
{ **pre** $e = E$; **post** $x = |E|$; **vf** VF }

begin
  if $e < 0$ **then** $Abs(-e, x)$
  else $x := e$
end; { Abs }

waarbij

$$VF \;=\; \begin{cases} 0 & \text{als } e \geq 0 \\ 1 & \text{als } e < 0 \end{cases}$$

Bij executie is de recursiediepte maximaal 1.

Is '**if** $e \leq 0$ **then** $Abs(-e, x)$' ook goed?

Cijfers tellen

function $NDigits(n, r\colon\ Integer)\colon\ Integer$;
{ **pre** $0 \leq n \ \wedge\ 2 \leq r$
  ret # cijfers in r-tallige representatie van n
  vf n }

begin
  if $n < r$ **then**
    $Result := 1$
  else
    { $2 \leq r \leq n$, dus n **div** $r < n$ }
    $Result := 1 +$ $NDigits(n$ **div** $r, r)$
end; { $NDigits$ }

Variante functie is hier een zeer ruime bovengrens op recursiediepte.

Kan ook met **while**-lus i.p.v. recursie.

Bitpatronen

Alle bitpatronen met lengte 3:

0 0 0

0 0 1

0 1 0

0 1 1

1 0 0

1 0 1

1 1 0

1 1 1

Hoe genereren met programma?

Bitpatronen van lengte 3

```
procedure Generate3BitPatterns;
var
  b0, b1, b2: 0..1; { de drie bits }

begin

  for b2 := 0 to 1 do begin

    for b1 := 0 to 1 do begin

      for b0 := 0 to 1 do begin
        writeln ( b2, b1, b0 )
      end { for b0 }

    end { for b1 }

  end { for b2 }

end;
```

Hoe de bitpatronen met lengte N genereren?

Variabel aantal geneste for-lussen?

Bitpatronen

Stappenplan (vergelijk met ontwerpen van een lus)

Specificeer: genereer alle N-bit rijtjes

Generaliseer: genereer alle N-bit rijtjes *met gegeven beginstuk* s ($\#s \leq N$)

Geef naam: $Generate(s)$

Druk oorspronkelijke probleem uit in termen van generalisatie (initialisatie): $Generate(\text{''}\text{''})$

Los $Generate(s)$ op, eventueel in termen van 'eenvoudigere' $Generate(t)$, bijv. $t = s + \text{'0'}$ en $t = s + \text{'1'}$ (stap, VF neemt af)

Variante functie: $N - \#s$ (aantal *rest*-bits)

Triviaal als $\#s = N$ (finalisatie)

Bitpatronen: oplossing

{ $P.s$: '$ProcessBitPattern$ is $1\times$ aangeroepen
voor elk N-bit patroon met beginstuk s' }

procedure $Generate(s\text{: }String)$;
{ **pre** $0 \leq \#s \leq N$
post $P.s$
vf $N - \#s$ }

begin
if $Length(s) = N$ **then**
$ProcessBitPattern(s)$ { $P.s$ }
else begin { $0 \leq \#s < N$ }
$Generate(s + \text{'0'})$ { $P.(s + \text{'0'})$ }
; $Generate(s + \text{'1'})$ { $P.(s + \text{'0'}) \wedge P.(s + \text{'1'})$ }
{ $P.s$ }
end { else }
end; { $Generate$ }

...

$Generate(\text{''}\text{''})$ { $P.\text{''}\text{''}$ }

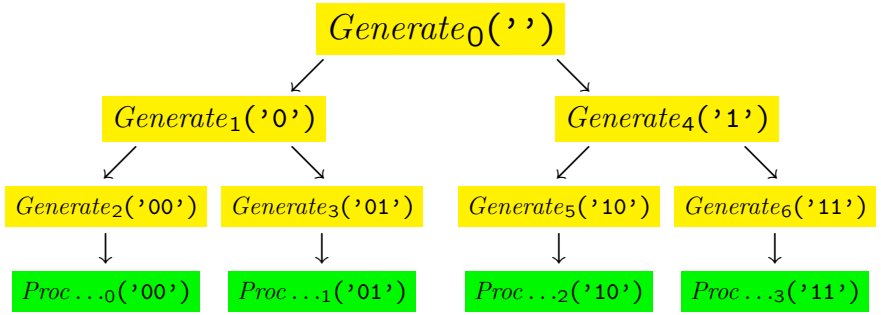
Eventueel **const** parameter s, maar toch veel gekopieerd

Dynamische aanroep-boom

Dynamische aanroep-boom
bij een *executie* van programma P:

- Knopen: routine-*aanroepen* bij executie van P.
- Pijlen: $q \rightarrow r$ waarbij executie van aanroep q resulteert in executie van aanroep r.

Voorbeeld bij bitpatronen met $N = 2$:



Bitpatronen: oplossing (2)

type $BitPattern$ = **array** [0..$N-1$] **of** 0..1;

var a: $BitPattern$;

{ $P.s$: '$ProcessBitPattern$ is 1× aangeroepen
voor elk N-bit patroon met staart s' }

procedure $Generate(i$: $Integer$ { ; **glob** a }); { **pre** $0 \leq i \leq N$
post $P.\tilde{a}[\tilde{i}..N) \ \wedge \ a[\tilde{i}..N) = \tilde{a}[\tilde{i}..N)$
vf i }

begin
if $i = 0$ **then**
$ProcessBitPattern(a)$ { $P.\tilde{a}[\tilde{i}..N)$ }
else begin { $0 < i \leq N$ }
$i := pred(i)$ { $i < \tilde{i}$ }
; $a[i] := 0$; $Generate(i)$ { $P.(0 \vdash \tilde{a}[\tilde{i}..N))$ }
; $a[i] := 1$; $Generate(i)$ { $P.(1 \vdash \tilde{a}[\tilde{i}..N))$ }
end { else }
end; { $Generate$ }

...

$Generate(N)$ { $P.\langle\rangle$ }

Bitpatronen: oplossing (3)

```
procedure Generate(var a: BitPattern);
{ pre true ; post P.⟨⟩ }

  var i: Integer;

  procedure GenRec { (glob i, a) } ;
  { pre 0 ≤ i ≤ N
    post P.ã[ĩ..N) ∧ i = ĩ ∧ a[ĩ..N) = ã[ĩ..N)
    vf i }

    begin
      if i = 0 then
        ProcessBitPattern(a) { P.ã[ĩ..N) }
      else begin { 0 < i ≤ N }
        i := pred(i) { i < ĩ }
       ; a[i] := 0 ; GenRec { P.(0 ⊢ ã[ĩ..N)) }
       ; a[i] := 1 ; GenRec { P.(1 ⊢ ã[ĩ..N)) }
       ; i := succ(i)
      end { else }
    end; { GenRec }

  begin i := N ; GenRec end; { Generate }

var b: BitPattern;
...
Generate(b) { P.⟨⟩ }
```


Foutieve array-sommeerder

```
const N = ... ; { 0 ≤ N }

type AI = array [0..N−1] of Integer;

function Sum(const a: AI; i, j: Integer): Integer;
{ pre 0 ≤ i ≤ j ≤ N
  ret (Σ k : i ≤ k < j : a[k])
  vf j − i }

  var h: Integer;

  begin
    if i = j then
      Result := 0
    else begin { i < j }
      h := (i + j) div 2
     ; Result := Sum(a, i, h) + Sum(a, h, j)
    end { else }
  end; { Sum }
```