Formal Methods – An Appetizer

Chapter 7: Procedures

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

An Abstraction Mechanism

The Role of Procedures

Procedures is a basic abstraction mechanism of programming helping us to structure our program by breaking it down in smaller modules reflecting its logical structure.

Procedure may be called from different contexts, they may be supplied with different parameters and they may call oneanother – and themselves!

Depending on the programming language procedures may also be called functions, methods, subroutines or subprograms.

Challenge

How does this change our notion of semantics?

The memory model is changed to be a stack of frames.

Guarded Commands With Procedures

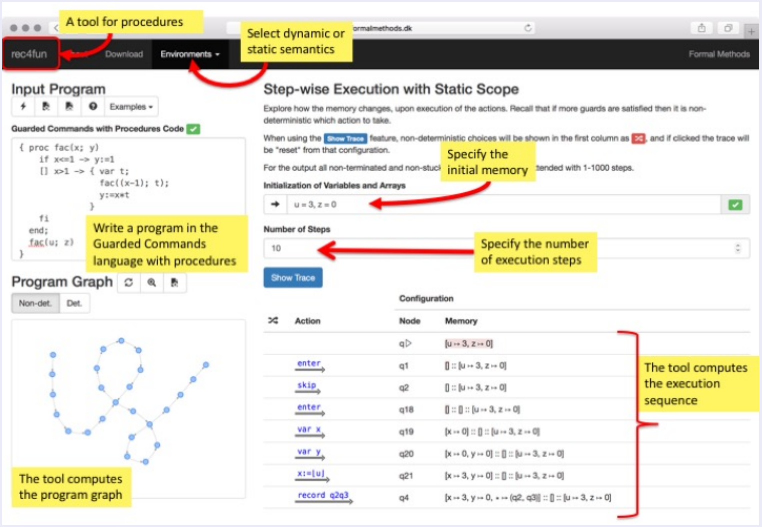
Factorial Function (FM p 98)

```
{ proc fac(x; y)
    if x ≤ 1 → y := 1
    [] x > 1 → { var t;
                 fac(x − 1; t);
                 y := x * t
               }
    fi
  end;
  fac(3; z)
}
```

New Constructs

- procedure declarations
  `proc fac(x; y) ... end`
- procedure calls
  `fac(3; z)`
  `fac(x − 1; t)`
- declarations
  `var t`
- blocks
  `{proc ... end; ...}`
  `{var t; ...}`

FormalMethods.dk/rec4fun


A tool for procedures
rec4fun
Download
Environments
Select dynamic or static semantics
Formal Methods
Examples
Non-det.
Det.
Show Trace
Specify the initial memory
Specify the number of execution steps
Write a program in the Guarded Commands language with procedures
The tool computes the program graph
The tool computes the execution sequence

Input Program

Guarded Commands with Procedures Code ✓

```
{ proc fac(x; y)
    if x<=1 -> y:=1
    [] x>1 -> { var t;
                fac((x-1); t);
                y:=x*t
              }
    fi
  end;
  fac(u; z)
}
```

Program Graph

Step-wise Execution with Static Scope

Explore how the memory changes, upon execution of the actions. Recall that if more guards are satisfied then it is non-deterministic which action to take.

When using the Show Trace feature, non-deterministic choices will be shown in the first column as 🟥, and if clicked the trace will be "reset" from that configuration.

For the output all non-terminated and non-stuck … tended with 1-1000 steps.

Initialization of Variables and Arrays

u = 3, z = 0

Number of Steps

10

		Configuration	
	Action	Node	Memory
		q▷	[u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{enter}}$	q1	[] :: [u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{skip}}$	q2	[] :: [u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{enter}}$	q18	[] :: [] :: [u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{var x}}$	q19	[x ↦ 0] :: [] :: [u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{var y}}$	q20	[x ↦ 0, y ↦ 0] :: [] :: [u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{x:=[u]}}$	q21	[x ↦ 3, y ↦ 0] :: [] :: [u ↦ 3, z ↦ 0]
	$\xrightarrow{\texttt{record q2q3}}$	q4	[x ↦ 3, y ↦ 0, ∗ ↦ (q2, q3)] :: [] :: [u ↦ 3, z ↦ 0]

7.1 Declarations

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Syntax and Program Graphs

Syntax (FM p 91)

A program consists of a sequence of declarations followed by a Guarded Command

- Variable declarations: `var` x
- Array declarations: `array` $A[n]$

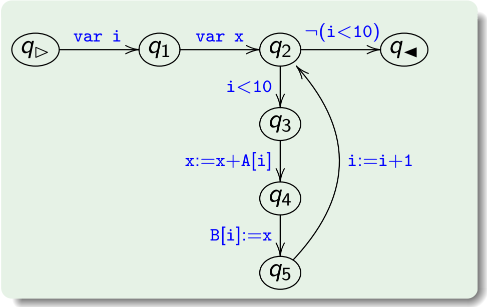
Program Graphs (FM p 92)

We introduce two new actions

- `var` x
- `array` $A[n]$

Traversing an array (FM p 91)

```
var i ;
var x ;
do i < 10 → x := x + A[i];
            B[i] := x;
            i := i + 1
od
```



The Semantics Of The New Actions

Variables and array entries are initialised to 0:

$$\mathcal{S}[\![\texttt{var } x]\!]\sigma = \sigma[x \mapsto 0]$$

$$\mathcal{S}[\![\texttt{array } A[n]]\!]\sigma = \sigma[A[0] \mapsto 0] \cdots [A[n-1] \mapsto 0]$$

Try It Out (FM p 92)

What happens if we make multiple declarations of the same variable as in `var u ; var u`?

Is it equivalent to just a single declaration as in `var u`?

Try It Out (FM p 92)

What happens if we make multiple declarations of the same array but using different lengths as in `array A[7] ; array A[3]`?

Note: $\mathcal{S}[\![\texttt{var } x]\!]\sigma$ extends the memory with a *new* entry for x

$\mathcal{S}[\![x := 0]\!]\sigma$ updates the memory of the *existing* entry for x

7.2 Blocks

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Local Declarations

Blocks $\{D\,;C\}$

Idea: the declarations of D are local to the command C

Example (FM p 93)

```
{  var y;
   y := 1;
   x := 1;
   { var x;
     x := 2;
     y := x + 1
   };
   x := x + y
}
```

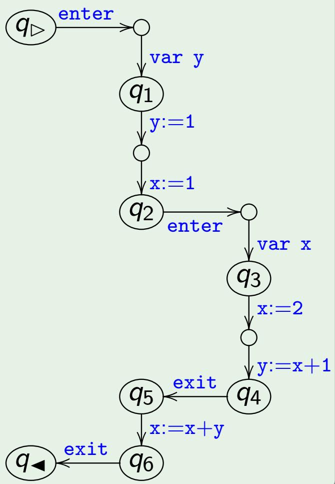
Program Graphs

(FM p 93)

We introduce two new actions for starting and ending a scope:

- enter
- exit

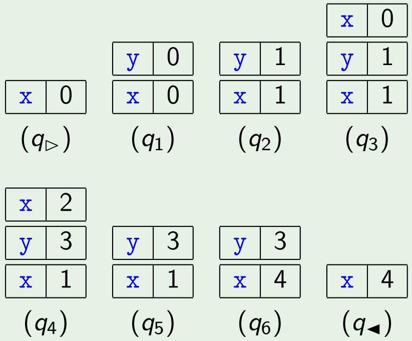
Program Graph (FM p 94)



A New Memory Model

A memory is now a stack of frames where a frame is a mapping from variables and array entries to values (as the memories before)

Example Memories (FM p 95)



Program Graph (FM p 94)

$$\begin{tikzcd}
q_\triangleright \arrow[r, "\texttt{enter}"] & \circ \arrow[d, "\texttt{var y}"] & \\
& q_1 \arrow[d, "\texttt{y:=1}"] & \\
& \circ \arrow[d, "\texttt{x:=1}"] & \\
& q_2 \arrow[r, "\texttt{enter}"'] & \circ \arrow[d, "\texttt{var x}"] \\
& & q_3 \arrow[d, "\texttt{x:=2}"] \\
& & \circ \arrow[d, "\texttt{y:=x+1}"] \\
& q_5 \arrow[d, "\texttt{x:=x+y}"] & q_4 \arrow[l, "\texttt{exit}"'] \\
q_\blacktriangleleft & q_6 \arrow[l, "\texttt{exit}"'] &
\end{tikzcd}$$

Using Stacks Of Frames

Revising The Semantics (FM p 96)

- the memory is a stack of frames $\mathbf{Mem_F} = \mathbf{Frame}^*$ where
  $$\mathbf{Frame} = \left(\ \mathbf{Var} \cup \{A[i] \mid A \in \mathbf{Arr}, \exists n : 0 \le i < n\}\ \right) \hookrightarrow \mathbf{Int}$$
- the semantic functions operate on stacks of frames
  $$\mathcal{S}_F[\![\cdot]\!] : \mathbf{Act} \to (\mathbf{Mem_F} \hookrightarrow \mathbf{Mem_F})$$

Values Of Variables

$$\vec{\sigma}(x) = \begin{cases} \sigma(x) & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', x \in \mathrm{dom}(\sigma) \\ \vec{\sigma}'(x) & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', x \notin \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\vec{\sigma}$	a stack of frames
$\sigma :: \vec{\sigma}$	a stack of frames with σ on the top
$\mathrm{dom}(\sigma)$	the domain of σ

Updating Variables

$$\vec{\sigma}[x \mapsto v] = \begin{cases} (\sigma[x \mapsto v]){::}\vec{\sigma}' & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', x \in \mathrm{dom}(\sigma) \\ \sigma{::}(\vec{\sigma}'[x \mapsto v]) & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', x \notin \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Semantics Of Selected Actions (FM p 96)

$$\mathcal{S}_{\mathsf{F}}[\![x := a]\!]\vec{\sigma} = \begin{cases} \vec{\sigma}[x \mapsto z] & \text{if } z = \mathcal{A}[\![a]\!]\vec{\sigma} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{F}}[\![b]\!]\vec{\sigma} = \begin{cases} \vec{\sigma} & \text{if } \mathcal{B}[\![b]\!]\vec{\sigma} = \text{true} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{F}}[\![\mathtt{var}\ x]\!]\vec{\sigma} = \begin{cases} (\sigma[x \mapsto 0])::\vec{\sigma}' & \text{if } \vec{\sigma} = \sigma::\vec{\sigma}' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{F}}[\![\mathtt{enter}]\!]\vec{\sigma} = [\,]::\vec{\sigma}$$

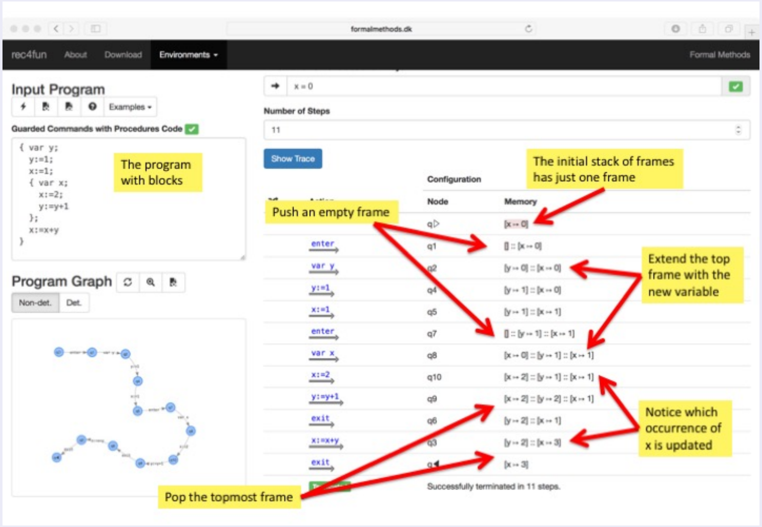
$$\mathcal{S}_{\mathsf{F}}[\![\mathtt{exit}]\!]\vec{\sigma} = \begin{cases} \vec{\sigma}' & \text{if } \vec{\sigma} = \sigma::\vec{\sigma}' \\ \text{undefined} & \text{otherwise} \end{cases}$$

We Need To Define

- $\mathcal{A}[\![a]\!]\vec{\sigma}$: the value of a in the frame stack $\vec{\sigma}$
- $\mathcal{B}[\![b]\!]\vec{\sigma}$: the value of b in the frame stack $\vec{\sigma}$

- `enter` pushes an empty frame $[\,]$ on the stack
- `exit` pops the topmost frame on the stack

FormalMethods.dk/rec4fun



Guarded Commands with Procedures Code ✓

```
{ var y;
  y:=1;
  x:=1;
  { var x;
    x:=2;
    y:=y+1
  };
  x:=x+y
}
```

x = 0

Number of Steps

11

Configuration

Action	Node	Memory
	q▷	[x ↦ 0]
enter	q1	[] :: [x ↦ 0]
var y	q2	[y ↦ 0] :: [x ↦ 0]
y:=1	q4	[y ↦ 1] :: [x ↦ 0]
x:=1	q5	[y ↦ 1] :: [x ↦ 1]
enter	q7	[] :: [y ↦ 1] :: [x ↦ 1]
var x	q8	[x ↦ 0] :: [y ↦ 1] :: [x ↦ 1]
x:=2	q10	[x ↦ 2] :: [y ↦ 1] :: [x ↦ 1]
y:=y+1	q9	[x ↦ 2] :: [y ↦ 2] :: [x ↦ 1]
exit	q6	[y ↦ 2] :: [x ↦ 1]
x:=x+y	q3	[y ↦ 2] :: [x ↦ 3]
exit	q◀	[x ↦ 3]

Successfully terminated in 11 steps.

Handling Arrays

Values of Array Entries

$$\vec{\sigma}(A[i]) = \begin{cases} \sigma(A[i]) & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', A[i] \in \mathrm{dom}(\sigma) \\ \vec{\sigma}'(A[i]) & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', A[i] \notin \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Updating Array Entries

$$\vec{\sigma}[A[i] \mapsto v] = \begin{cases} (\sigma[A[i] \mapsto v]){::}\vec{\sigma}' & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', A[i] \in \mathrm{dom}(\sigma) \\ \sigma{::}(\vec{\sigma}'[A[i] \mapsto v]) & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}', A[i] \notin \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{F}}[\![A[a_1] := a_2]\!]\vec{\sigma} = \begin{cases} \vec{\sigma}[A[z_1] \mapsto z_2] & \text{if } z_1 = \mathcal{A}[\![a_1]\!]\vec{\sigma}, z_2 = \mathcal{A}[\![a_2]\!]\vec{\sigma} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{F}}[\![\texttt{array } A[n]]\!]\vec{\sigma} = \begin{cases} \sigma[A[0] \mapsto 0]\cdots[A[n-1] \mapsto 0]{::}\vec{\sigma}' & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Try It Out and Hands On: Local Declarations

FormalMethods.dk/rec4fun (FM p 94, 97)

Construct a complete execution sequence for the Fibonnacci program from a memory where `x` is 4. Explain why the stack of frames is evolving as displayed.

Fibonnacci Program (FM p 94)

```
{ var z;
  var i;
  z := 1;
  y := 0;
  i := 0;
  do i < x → { var t;
               t := z;
               z := y;
               y := t + z;
               i := i + 1
             }
}
```

Array Program (FM p 97)

```
A[3] := 1;
{   array A[2];
    A[3] := 7
}
```

FormalMethods.dk/rec4fun (FM p 97)

Specify an initial memory for the array program where the global array `A` has length 5 and construct a complete execution sequence. Do you get the expected result? Discuss how to modify the semantics.

7.3 Procedures with Dynamic Scope

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Guarded Commands With Procedures

Factorial Function (FM p 98)

```
{ proc fac(x; y)
    if x ≤ 1 → y := 1
    [] x > 1 → { var t;
                 fac(x − 1; t);
                 y := x ∗ t
               }
    fi
  end;
  fac(3; z)
}
```

proc $p(x\,;y)$ C end (FM p 98)

Declares a procedure with name p, a formal input parameter x, a formal output parameter y and body C.

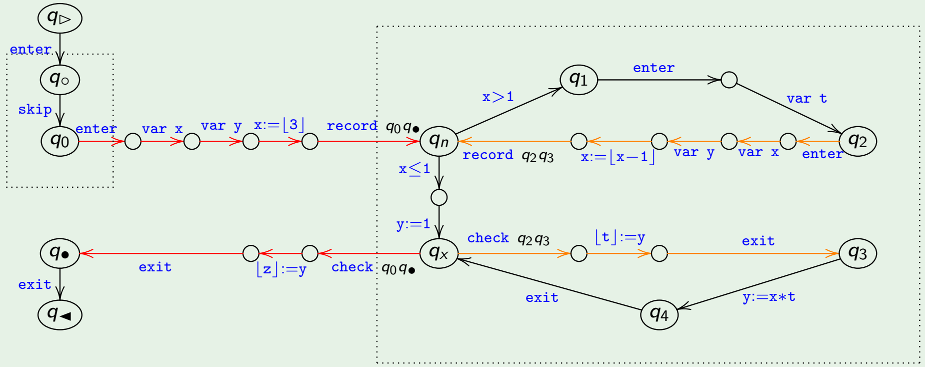
$p(a\,;z)$ (FM p 98)

Calls a procedure with name p, actual input parameter a and actual output parameter z.

Idea: At the call, the value of the arithmetic expression a is assigned to the variable x and the variable y is initialised to 0. The procedure body C is then executed. When returning from the call the value of y is assigned to the variable z.

Program Graphs: What We Are Aiming For

Program Graph For Factorial Function (FM p 101)



Leftmost:
the main program

Red and orange parts:
procedure call and return

Rightmost:
procedure body

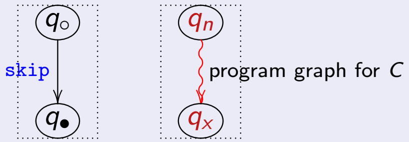
Program Graph For Procedure Declarations

Idea: We construct the program graph for the procedure at the point of its declaration

The Environment ρ (FM p 99)

We build an environment holding information about the procedure declarations.

`proc` $p(x\,;y)$ C `end` (FM p 99)



For `proc` $p(x\,;y)$ C `end` we will record

$$\rho(p) = (x, y, q_n, q_x)$$

For the declaration itself we only generate a `skip` action from $q_\circ$ to $q_\bullet$; for the procedure body we create fresh nodes q_n and q_x.

Idea: When a procedure is called we will consult the environment ρ to obtain its formal parameters and the initial and final nodes of its program graph.

Program Graph For Procedure Call

$p(a\,;z)$ (FM p 100)

$$\begin{tikzcd}
q_\circ \arrow[d, "\texttt{enter}"'] & q_\bullet \\
\circ \arrow[d, "\texttt{var}\ x"'] & \\
\circ \arrow[d, "\texttt{var}\ y"'] & \\
\circ \arrow[d, "x := \lfloor a \rfloor"'] & \circ \arrow[uuu, "\texttt{exit}"'] \\
\circ \arrow[d, "\texttt{record}\ q_\circ q_\bullet"'] & \circ \arrow[u, "\lfloor z \rfloor := y"'] \\
q_n \arrow[r, squiggly] & q_x \arrow[u, "\texttt{check}\ q_\circ q_\bullet"']
\end{tikzcd}$$

Assume $\rho(p) = (x, y, q_n, q_x)$

At The Call From $q_\circ$ (FM p 100)

- create a new frame and declare x and y
- assign the value of a (as computed before the call) to x; written $x := \lfloor a \rfloor$
- record that the call is between $q_\circ$ and $q_\bullet$; continue from q_n

At The Return From q_x (FM p 100)

- check that we are returning to where we came from (between $q_\circ$ and $q_\bullet$)
- assign the value of y to z (as known before the call); written $\lfloor z \rfloor := y$
- pop the top frame; continue from $q_\bullet$

Hands On: Program Graphs For Procedures

FormalMethods.dk/rec4fun (FM p 101)

Use the tool to construct the program graph for the Fibonnacci function.

Explain how the various occurrences of the new actions are used in the program graph; in particular

- the occurrences of record $q_\circ q_\bullet$ and check $q_\circ q_\bullet$
- the occurrences of variable declarations
- the occurrences of the special assignments $x := \lfloor a \rfloor$ and $\lfloor z \rfloor := y$

Fibonnacci Function (FM p 101)

```
{ proc fib(x; y)
    if x ≤ 1 → y := 1
    [] x > 1 → { var t;
                 var u;
                 fib(x − 1; t);
                 fib(x − 2; u);
                 y := t + u
               }
    fi
  end;
  fib(5; z)
}
```

Semantics Of The Special Assignments

At The Procedure Call: $x := \lfloor a \rfloor$ (FM p 102)

We bypass the topmost frame when determining the value of a; we do not do so when determining x:

$$\mathcal{S}_{\mathsf{F}}[\![x := \lfloor a \rfloor]\!]\vec{\sigma} \quad = \quad \begin{cases} \vec{\sigma}[x \mapsto \mathcal{A}[\![a]\!]\vec{\sigma}'] & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}' \text{ and} \\ & \mathcal{A}[\![a]\!]\vec{\sigma}' \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

At The Procedure Return: $\lfloor z \rfloor := y$ (FM p 102)

We bypass the topmost frame when assigning to the variable z; we do not do so when determining the value of y:

$$\mathcal{S}_{\mathsf{F}}[\![\lfloor z \rfloor := y]\!]\vec{\sigma} \quad = \quad \begin{cases} \sigma{::}\vec{\sigma}'[z \mapsto \mathcal{A}[\![y]\!]\vec{\sigma}] & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}' \text{ and} \\ & \mathcal{A}[\![y]\!]\vec{\sigma} \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Procedure Calls And Returns

We introduce a special token $*$ in the frames:

$\sigma(*) = (q_\circ, q_\bullet)$: the call happened at $q_\circ$ and should return to $q_\bullet$

Recording The Call (FM p 102)

We record in the topmost frame that the call happens on the edge $(q_\circ, q_\bullet)$:

$$\mathcal{S}_{\mathsf{F}}[\![\texttt{record}\ q_1 q_2]\!]\vec{\sigma} \;=\; \begin{cases} \sigma[* \mapsto (q_1, q_2)]{::}\vec{\sigma}' & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}' \\ \text{undefined} & \text{otherwise} \end{cases}$$

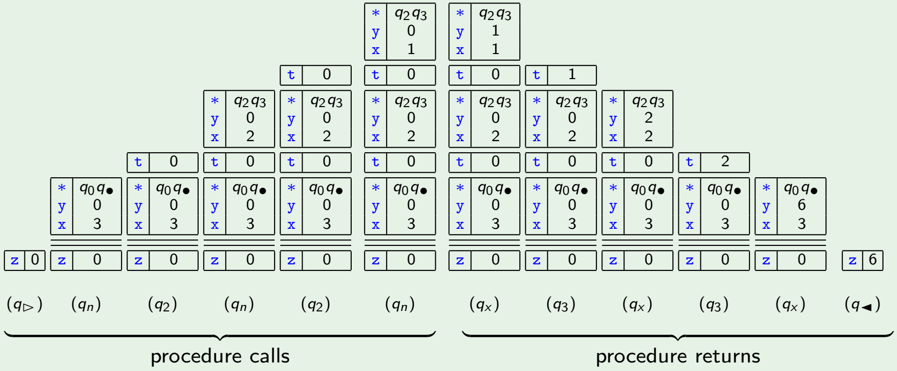
Checking The Call (FM p 102)

We check that we return from the call recorded in the topmost frame:

$$\mathcal{S}_{\mathsf{F}}[\![\texttt{check}\ q_1 q_2]\!]\vec{\sigma} \;=\; \begin{cases} \vec{\sigma} & \text{if } \vec{\sigma} = \sigma{::}\vec{\sigma}' \text{ and } \sigma(*) = (q_1, q_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Execution Sequence: What We Are Aiming For

Selected Memories For The Factorial Function (FM p 103)



- $(q_0, q_\bullet)$: top-most procedure call
- (q_2, q_3): recursive procedure call

- q_n: entry node for procedure body
- q_x: exit node for procedure body

Hands On: Program Graphs For Procedures

FormalMethods.dk/rec4fun (FM p 101)

Use the tool to construct an execution sequence for the Fibonnacci function.

Explain how the frames evolve during the execution; in particular

- how the memories are effected by occurrences of `record` $q_\circ q_\bullet$ and `check` $q_\circ q_\bullet$
- how the memories are updated by the actual parameters (using the special assignments $x := \lfloor a \rfloor$ and $\lfloor z \rfloor := y$)

Fibonnacci Function (FM p 101)

```
{ proc fib(x; y)
    if x ≤ 1 → y := 1
    [] x > 1 → { var t;
                 var u;
                 fib(x − 1; t);
                 fib(x − 2; u);
                 y := t + u
               }
    fi
  end;
  fib(5; z)
}
```

7.4 Procedures with Static Scope

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Dynamic Versus Static Scope

FormalMethods.dk/rec4fun

(FM p 105)

Use the tool to construct a program graph and an execution sequence for the program.

Example Program (FM p 104)

```
u := 1;
{   proc p(x; y)
        y := x + 1;
        u := 2
    end;
    var u;
    var v;
    p(u; v)
}
```

Which Variables Should Be Updated?

- dynamic scope: the procedure p updates the occurrence of u that is in the scope when p is called
- static scope: the procedure p updates the occurrence of u that is in the scope when p is declared

FormalMethods.dk/rec4fun

The tool has execution environments for dynamic as well as static scope. Try both of them and explain that the difference is as expected.

Renaming Variables And Arrays

How Does The Difference Between Dynamic And Static Scope Arise?

It happens when variable names are reused.

How Can We Enforce Static Scope?

We rename all variable and array names are distinct – this includes global variables and arrays, local variables and arrays, as well as formal parameters of procedures

Construction Of Program Graphs

(FM p 105, 106, 107)

We rename variables and arrays on the fly so that they all have different names – to do so we extend the construction with an environment keeping track of the new names.

Semantics Of Actions

No changes – it just uses the new names

FormalMethods.dk/rec4fun

Note that the tool displays the original names; the renaming is internal in the system and hence hidden from the user.