Formal Methods – An Appetizer

Chapter 2: Guarded Commands

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

FormalMethods.dk

Programming Languages

The Programming Task

When programming we use a programming language such as C, Java or F# to construct the software components needed.

Syntax

The syntax of a programming language is usually specified by a BNF grammar.

Additional well-formedness conditions might be imposed.

Semantics

There are many approaches to how to define the semantics of a programming language.

Some are mathematical in nature, others are more operational.

Be Aware

Constructs that look similar in different programming languages may have subtle differences, and constructs that look rather different may turn out to be fairly similar after all.

The Role of Program Graphs

A Uniform Approach

We transform programs into program graphs and apply their semantics:

program

↓

construct program graph

↓

program graph

↓

apply semantics

↓

execution sequence

The approach is applicable to many programming languages.

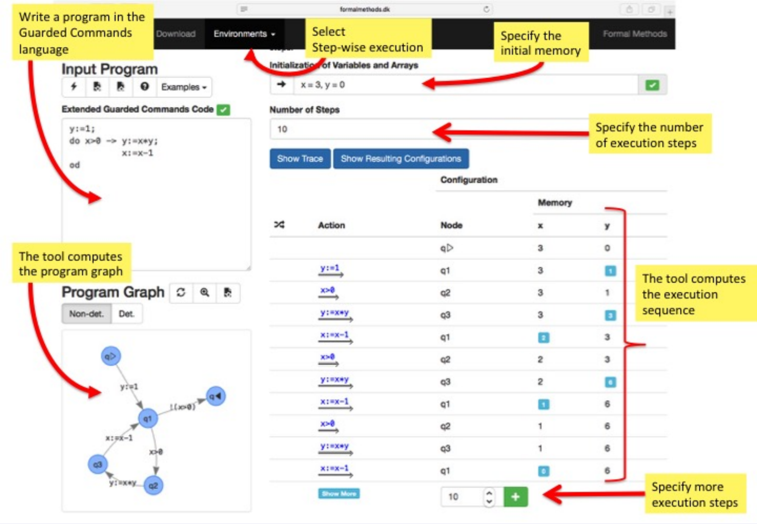
Guarded Commands (FM p 15)

We study programs in Dijkstra's language of Guarded Commands; this is a tiny, probably unfamiliar language.

MicroC (FM p 129)

In a sequence of tasks we study programs in a C-like language supporting a variety of control structures.

FormalMethods.dk/fm4fun



2.1 Syntax

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.

ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Guarded Commands

Factorial Function (FM p 15)

$\mathtt{y} := 1;$
$\mathtt{do}\ \mathtt{x} > 0 \rightarrow \mathtt{y} := \mathtt{x} * \mathtt{y};$
$\qquad\qquad\quad \mathtt{x} := \mathtt{x} - 1$
$\mathtt{od}$

Maximum Function (FM p 16)

$\mathtt{if}\ \mathtt{x} \geq \mathtt{y} \rightarrow \mathtt{z} := \mathtt{x}$
$\ [] \ \mathtt{y} > \mathtt{x} \rightarrow \mathtt{z} := \mathtt{y}$
$\mathtt{fi}$

Basic Commands

- assignment: $x := a$
- do nothing: `skip`

Composite Commands

- sequencing
  $C_1 ; \cdots ; C_k$
- conditional
  $\mathtt{if}\ b_1 \rightarrow C_1 \ [] \cdots [] \ b_k \rightarrow C_k\ \mathtt{fi}$
- iteration
  $\mathtt{do}\ b_1 \rightarrow C_1 \ [] \cdots [] \ b_k \rightarrow C_k\ \mathtt{od}$

BNF Syntax

Commands and Guarded Commands

$$C \quad ::= \quad x := a \mid \texttt{skip} \mid C_1 \,;\, C_2 \mid \texttt{if}\ GC\ \texttt{fi} \mid \texttt{do}\ GC\ \texttt{od}$$

$$GC \quad ::= \quad b \rightarrow C \mid GC_1 \,[]\, GC_2$$

Arithmetic and Boolean Expressions

$$a \quad ::= \quad n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \,/\, a_2 \mid a_1 \,\hat{}\, a_2$$

$$b \quad ::= \quad \texttt{true} \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 \geq a_2 \mid b_1 \wedge b_2 \mid b_1 \,\&\&\, b_2 \mid \neg\, b_0$$

The BNF syntax specify abstract syntax trees for commands, guarded commands and arithmetic and boolean expressions.

Therefore we do not need to introduce explicit *brackets* in the syntax, although we shall feel free to use *parentheses* in textual presentations of programs in order to disambiguate the syntax.

Try It Out: Guarded Commands

Basic Commands

- assignment: $x := a$
- do nothing: `skip`

Composite Commands

- sequencing
  $C_1 ; \cdots ; C_k$
- conditional
  $\texttt{if}\ b_1 \rightarrow C_1 \ [] \cdots [] \ b_k \rightarrow C_k \ \texttt{fi}$
- iteration
  $\texttt{do}\ b_1 \rightarrow C_1 \ [] \cdots [] \ b_k \rightarrow C_k \ \texttt{od}$

Factorial Function (FM p 15)

```
y := 1;
do x > 0 → y := x * y;
           x := x − 1
od
```

Try It Out (FM p 17)

Construct a Guarded Commands program for the power function computing the power 2^n of a number n without using exponentiation (^).

2.2 Program Graphs

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.

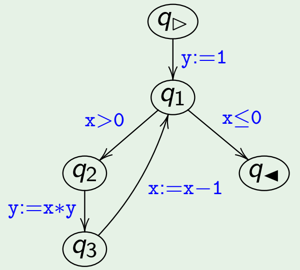
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Construction of Program Graphs (FM p 19)

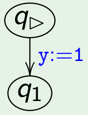
Factorial Function (FM p 15)

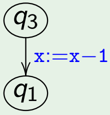
$$
\begin{array}{l}
\mathtt{y} := 1; \\
\mathtt{do}\ \mathtt{x} > 0 \rightarrow \mathtt{y} := \mathtt{x} * \mathtt{y}; \\
\qquad\qquad\quad\ \mathtt{x} := \mathtt{x} - 1 \\
\mathtt{od}
\end{array}
$$

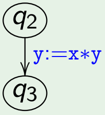


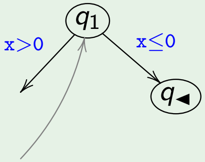
Idea

The program graph is constructed from smaller program graphs for its constituents.



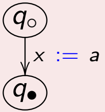




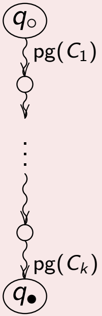


Construction of Program Graphs (FM p 18, 19)

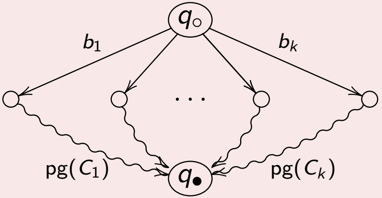
$x := a$



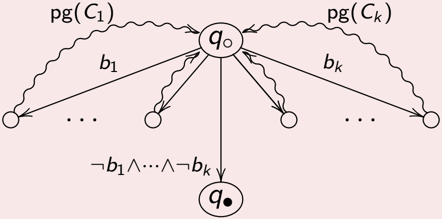
$C_1 \,;\, \cdots \,;\, C_k$



$\texttt{if}\ b_1 \to C_1\ [\,]\ \cdots\ [\,]\ b_k \to C_k\ \texttt{fi}$



$\texttt{do}\ b_1 \to C_1\ [\,]\ \cdots\ [\,]\ b_k \to C_k\ \texttt{od}$



Idea

The nodes $q_\circ$ and $q_\bullet$ are given, the others are generated on the fly.

At the top-level we start with $q_\triangleright$ and $q_\blacktriangleleft$.

Try It Out and Hands On: Program Graphs

Try It Out (FM p 20)

Returning to the Guarded Command program for the power function, use these techniques to construct the corresponding program graph.

FormalMethods.dk/fm4fun

Once you have typed in a syntactically correct program, the tool will construct the corresponding program graph.

Use the tool to check your understanding of how program graphs are constructed.

FormalMethods.dk/fm4fun

In its simplest form the Guarded Command program for the power function consists of a number of assignments and a single loop. Explain why the program graph is as constructed by the tool.

2.3 Semantics

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

FormalMethods.dk

Semantics

The Semantics consists of (FM p 4)

- A non-empty set **Mem** called the memory
- A semantic function $\mathcal{S}[\![\cdot]\!]$ specifying the meaning of the actions

For Guarded Commands

- $\mathbf{Mem} = \mathbf{Var} \to \mathbf{Int}$
- Three kinds of actions
  - $x := a$
  - `skip`
  - b

$\mathcal{S}[\![\cdot]\!] : \mathbf{Act} \to (\mathbf{Mem} \hookrightarrow \mathbf{Mem})$ (FM p 23)

$$\mathcal{S}[\![\texttt{skip}]\!]\sigma = \sigma$$

$$\mathcal{S}[\![x := a]\!]\sigma = \begin{cases} \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] & \text{if } \mathcal{A}[\![a]\!]\sigma \text{ is defined} \\ & \text{and } x \in \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

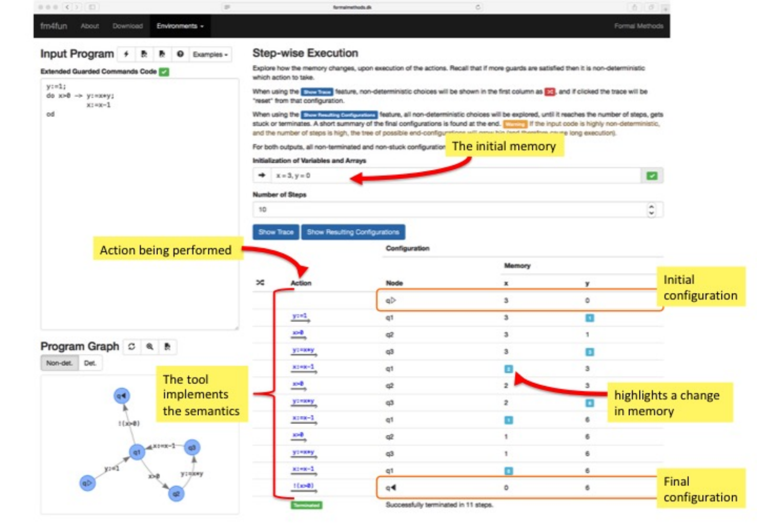
Evaluating Expressions

(FM p 21, 22)

$\mathcal{A}[\![a]\!]\sigma$: the value of a in memory σ

$\mathcal{B}[\![b]\!]\sigma$: the truth value of b in σ

FormalMethods.dk/fm4fun



Guarded Commands with Arrays

Example (FM p 24)

```
i := 0;
x := 0;
do i < 10 → x := x + A[i];
            B[i] := x;
            i := i + 1
od
```

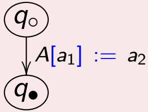
Extended Syntax

$$C ::= \cdots \mid A[a_1] := a_2$$

$$a ::= \cdots \mid A[a]$$

Program Graph

(FM p 24)



Memory

$$\mathbf{Mem} = \big(\ \mathbf{Var} \cup \{A[i] \mid A \in \mathbf{Arr}, 0 \leq i < \mathrm{size}(A)\}\ \big) \rightarrow \mathbf{Int}$$

Semantics (FM p 24)

$$\mathcal{S}[\![A[a_1] := a_2]\!]\sigma = \begin{cases} \sigma[A[z_1] \mapsto z_2] & \text{if } z_1 = \mathcal{A}[\![a_1]\!]\sigma, 0 \leq z_1 < \mathrm{size}(A) \\ & \text{ and } z_2 = \mathcal{A}[\![a_2]\!]\sigma \\ \text{undefined} & \text{otherwise} \end{cases}$$

Evaluating Arithmetic Expressions

$\mathcal{A}[\![a]\!]\sigma$: Value of a in Memory σ

$$\mathcal{A}[\![n]\!]\sigma = n$$

$$\mathcal{A}[\![x]\!]\sigma = \sigma(x)$$

$$\mathcal{A}[\![A[a]]\!]\sigma = \begin{cases} \sigma(A[z]) & \text{if } z = \mathcal{A}[\![a]\!]\sigma, \\ & \quad 0 \leq z < \text{size}(A) \\ \text{undefined otherwise} \end{cases}$$

$$\mathcal{A}[\![a_1 \ op_a \ a_2]\!]\sigma = \begin{cases} z \text{ if } z_1 = \mathcal{A}[\![a_1]\!]\sigma, z_2 = \mathcal{A}[\![a_2]\!]\sigma, \\ \quad \text{and } z = z_1 \ op_a \ z_2 \text{ is defined} \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

op_a	arithmetic operators
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	integer division; undefined if second argument is zero
^	exponentiation; undefined if second argument is negative
$\cdot$	$\cdots$

Example Evaluation

$$\begin{aligned} \mathcal{A}[\![\mathtt{x} + \mathtt{A[i]}]\!]\sigma &= \mathcal{A}[\![\mathtt{x}]\!]\sigma + \mathcal{A}[\![\mathtt{A[i]}]\!]\sigma \\ &= \sigma(\mathtt{x}) + \sigma(\mathtt{A}[\sigma(\mathtt{i})]) \\ &= 12 + \sigma(\mathtt{A[2]}) = 12 + 7 = 19 \end{aligned}$$

x	12
i	2

A[0]	2
A[1]	3
A[2]	7
A[3]	5

Evaluating Boolean Expressions

$\mathcal{B}[\![b]\!]\sigma$: Truth Value of b in Memory σ

$$\mathcal{B}[\![\texttt{true}]\!]\sigma = \text{true}$$

$$\mathcal{B}[\![a_1 \; op_r \; a_2]\!]\sigma = \begin{cases} t \text{ if } z_1 = \mathcal{A}[\![a_1]\!]\sigma, z_2 = \mathcal{A}[\![a_2]\!]\sigma, \\ \quad \text{and } t = z_1 \; op_r \; z_2 \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![b_1 \; op_b \; B_2]\!]\sigma = \begin{cases} t \text{ if } t_1 = \mathcal{B}[\![b_1]\!]\sigma, t_2 = \mathcal{B}[\![b_2]\!]\sigma, \\ \quad \text{and } t = t_1 \; op_b \; t_2 \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

op_r	relational operators
$=$	equality
$<$	less than
...	...

op_b	boolean operators
$\wedge$	conjunction; needs both arguments
$\&\&$	short-circuit conjunction; needs second argument only if the first is true
...	...

Example Evaluation

$\mathcal{B}[\![(\texttt{j} > \texttt{0}) \wedge (\texttt{A[j-1]} > \texttt{A[j]})]\!]\sigma$ is undefined

$\mathcal{B}[\![(\texttt{j} > \texttt{0}) \;\&\&\; (\texttt{A[j-1]} > \texttt{A[j]})]\!]\sigma$ is false

j	0

A[0]	2
A[1]	3
A[2]	7

Hands On: Execution Sequences

FormalMethods.dk/fm4fun

Use the tool to construct execution sequences for the power function as written in the Guarded Command language. What happens when the exponent is zero and when it is negative? Explain why the tool gives rise to the observed result.

Insertion Sort

```
i := 1;
do i < n →
  j := i;
  do (j > 0)&&(A[j-1] > A[j]) →
       t := A[j]; A[j] := A[j-1];
       A[j-1] := t; j := j-1
  od;
  i := i + 1
od
```

Construct execution sequences for the sorting program for different initial memories and for modifications of the program.

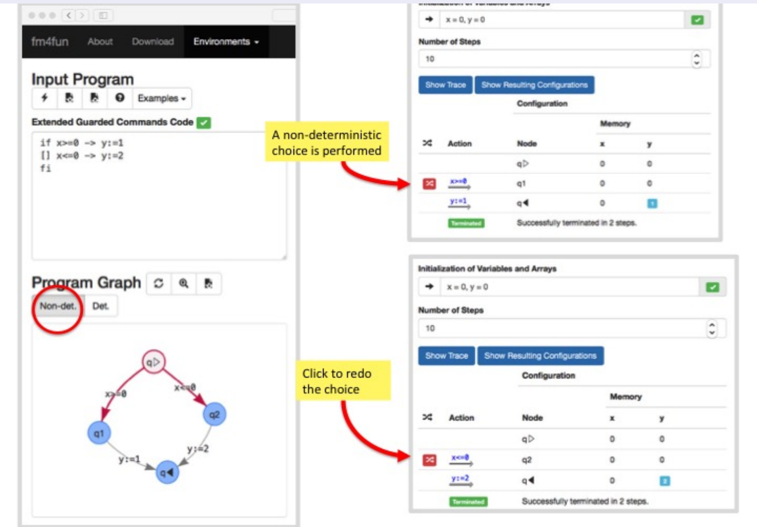
- What happens if `A` has size less than `n`? Or greater than `n`?
- What happens if `&&` is replaced by $\wedge$? And `i < n` by `i ≤ n`?

2.4 Alternative Approaches

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

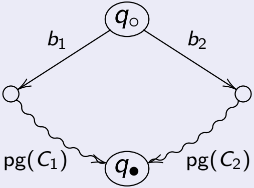
`FormalMethods.dk`

Non-deterministic programs



Alternatives: Deterministic and Evolving Versions

if $b_1 \rightarrow C_1 \; [] \; b_2 \rightarrow C_2$ **fi**

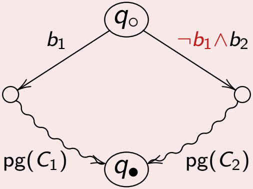


What happens if

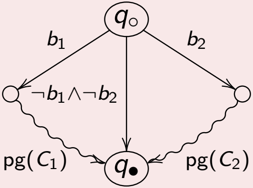
- both b_1 and b_2 hold?
- neither b_1 nor b_2 holds?

We can modify the construction of program graphs to be explicit on this.

Deterministic Version (FM p 25)



Evolving Version (FM p 26)



Hands On: Deterministic and Evolving Versions

Non-deterministic and Deterministic Program Graphs

In the tool you can select whether you want to construct program graphs that are non-deterministic or deterministic.

Example

```
if   x ≥ 10 → y := 10
[]   x ≥ 5 → y := 5
[]   x ≥ 0 → y := 0
fi
```

FormalMethods.dk/fm4fun

Use the tool to determine how many execution sequences we have for various choices of values for `x` and depending on whether we use the non-deterministic or the deterministic mode.

Modify the program to be an evolving system and use the tool to check that it is indeed evolving.

Would you like to make a similar modification if the program had used a `do` ··· `od` construct rather than an `if` ... `fi` construct?

2.5 More Control Structures

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

FormalMethods.dk

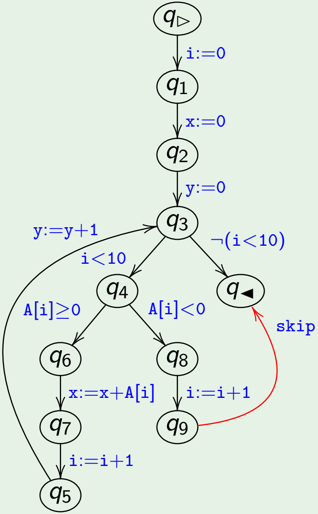
The break and continue Commands

A break inside a do-loop transfers control to its end. A continue transfers control to its beginning.

Example (FM p 27)

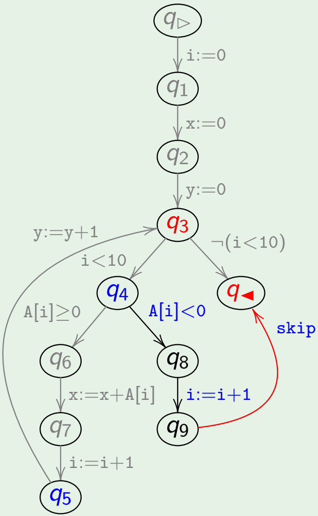
```
i := 0;
x := 0;
y := 0;
do i < 10 →
   if  A[i] ≥ 0 → x := x + A[i];
                  i := i + 1
   []  A[i] < 0 → i := i + 1;
                  break
   fi;
   y := y + 1
od
```

Program Graph (FM p 28)



Construction of Program Graphs

Program Graph (FM p 28)



Idea

The following nodes are given:

- $q_\circ$ and $q_\bullet$
- q_b to be used for `break`
- q_c to be used for `continue`

Whenever a `do` construct is considered we redefine q_b and q_c for its body.

Example: $\texttt{A[i]} < 0 \to \texttt{i} := \texttt{i} + 1; \texttt{break}$

The following nodes are given:

$q_\circ$	$q_\bullet$	q_b	q_c
q_4	q_5	$q_\blacktriangleleft$	q_3

The nodes q_8 and q_9 are created on the fly.

Hands On: break, continue and skip

Example (FM p 27)

```
i := 0;
x := 0;
y := 0;
do i < 10 →
   if  A[i] ≥ 0 → x := x + A[i];
                  i := i + 1
   []  A[i] < 0 → i := i + 1;
                  continue
   fi;
   y := y + 1
od
```

FormalMethods.dk/fm4fun

Use the tool to construct program graphs for variations of the programs using break, continue or skip in the second branch of the conditional.

Explain why the program graphs are as constructed.

Use the tool to construct a couple of execution sequences illustrating the differences in the semantics of the three programs.

Defining, Throwing and Handing Exceptions

Searching for a value (FM p 29)

$$
\begin{array}{l}
\texttt{i} := 0;\\
\texttt{try}\\
\quad \texttt{do } \texttt{A[i]} = \texttt{x} \to\\
\qquad\quad \texttt{throw yes}\\
\quad [] \ \ \neg(\texttt{A[i]} = \texttt{x}) \to\\
\qquad\quad \texttt{if i} < 9 \to\\
\qquad\qquad\quad \texttt{i} := \texttt{i} + 1\\
\qquad\quad [] \ \ \texttt{i} \geq 9 \to\\
\qquad\qquad\quad \texttt{throw no}\\
\qquad\quad \texttt{fi}\\
\quad \texttt{od}\\
\texttt{catch yes}: \cdots\\
\quad [] \quad \texttt{no}: \cdots\\
\texttt{yrt}
\end{array}
$$

Extended Syntax

$$
\begin{array}{rcl}
C & ::= & \texttt{try } C \texttt{ catch } HC \texttt{ yrt}\\
 & | & \texttt{throw } e \mid \cdots\\
HC & ::= & e\text{: } C\\
 & | & HC_1 \; [] \; HC_2
\end{array}
$$

Construction of Program Graphs

We need a handler environment that for each exception e tells which node q_e control should be transfered to.

- it is build by the handler commands
- it is used by the `throw` construct

Hands On: FormalMethods.dk/fm4fun