

# Modelling Recursion in Event-B

Stefan Hallerstede

Universität Düsseldorf

Rodin User and Developer Workshop  
Düsseldorf  
21 September 2010

# Contents

Introduction

Modelling a Recursive Procedure

Recursive Program Development

Model and Proof

Abstract Program

Concrete Program

Program Termination

Mutual Recursion

Conclusion

# Contents

Introduction

Modelling a Recursive Procedure

Recursive Program Development

Model and Proof

Abstract Program

Concrete Program

Program Termination

Mutual Recursion

Conclusion

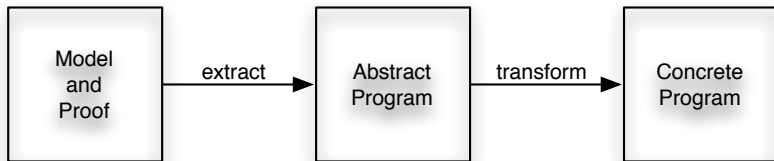
# Development and Verification of Recursive Programs

- ▶ We seek a **method**:
  - ▶ Easy to **apply** (also to larger programs)
  - ▶ **Tool** support (such as Rodin)
  - ▶ Data **refinement**
  - ▶ **Imperative** programs and procedures
- ▶ Some **related work**:<sup>1</sup>
  - ▶ **Program Verification** (Hoare)
  - ▶ **Proof Outlines** (Owicki/Gries)
  - ▶ **Refinement Calculus** (Morgan)
  - ▶ **Refinement Calculus** (Back)

---

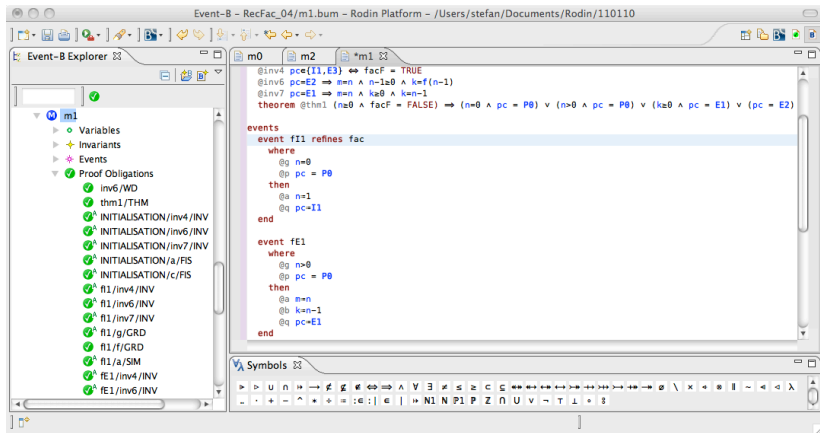
<sup>1</sup>Missing from the list: B-Method and VDM, for instance

# Method of Program Development



- ▶ A **model** describes an “*abstract program*”
- ▶ “*Concrete programs*” are an **indirect target**
  - ▶ Maybe **not** reached by refinement
- ▶ “*Abstract program*”:
  - ▶ only **call-by-reference** parameters
  - ▶ and no **global** variables

# Use of Event-B (and Rodin)



The screenshot shows the Rodin Platform interface. On the left, the 'Event-B Explorer' shows a tree view for project 'm1' with several proof obligations marked as successful (green checkmarks). The main editor displays Event-B code for a counter variable 'n'. The code includes initial conditions, a theorem, and two events: 'fi1' and 'fe1'. The 'fi1' event refines the 'fac' invariant, and the 'fe1' event updates 'n'.

```
@inv4 pc={I1,E3} ⇔ facF = TRUE
@inv6 pc=E2 ⇒ m=n ∧ n-1≥0 ∧ k=f(n-1)
@inv7 pc=E1 ⇒ m=n ∧ k≥0 ∧ k=n-1
theorem @thm1 (n≥0 ∧ facF = FALSE) ⇒ (n=0 ∧ pc = P0) ∨ (n>0 ∧ pc = P0) ∨ (k≥0 ∧ pc = E1) ∨ (pc = E2)

events
event fi1 refines fac
  where
    @g n=0
    @p pc = P0
  then
    @a n-1
    @q pc=I1
  end

event fe1
  where
    @g n>0
    @p pc = P0
  then
    @a m-n
    @b k=n-1
    @q pc=E1
  end
```

- ▶ Clear and **direct mapping** to Event-B machines
- ▶ Use **Rodin** to do the proofs
- ▶ Keep **Event-B** notion of refinement
- ▶ For now write the “mapped” model **directly in Event-B**

*Heinrich Heine*

HEINRICH HEINE  
UNIVERSITÄT DÜSSELDORF

# Contents

Introduction

**Modelling a Recursive Procedure**

Recursive Program Development

Model and Proof

Abstract Program

Concrete Program

Program Termination

Mutual Recursion

Conclusion

# Proof Outline of Factorial Procedure

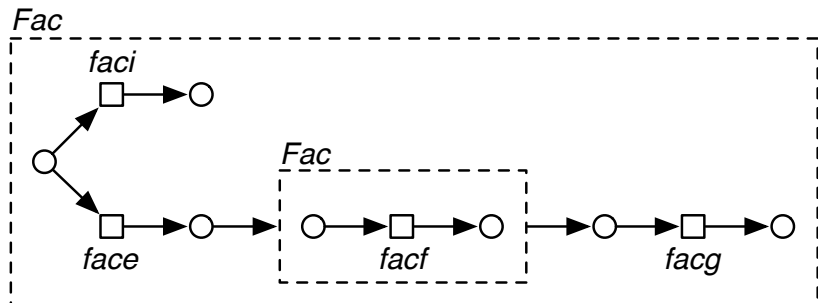
## Program Verification

```
{n ≥ 0 ∧ n = N} Fac(n) {n = N!} ::  
  {n ≥ 0 ∧ n = N}  
  if n = 0 then  
    {n = 0 ∧ n = N}  
    n := 1  
    {n = N!}  
  else  
    {n > 0 ∧ n = N}  
    var m := n;  
    {m = N ∧ n > 0 ∧ n = N-1}  
    n := n-1;  
    {m = N ∧ n ≥ 0 ∧ n = (N-1)}  
    Fac(n);  
    {m = N ∧ n = (N-1)!}  
    n := m * n  
    {n = N!}  
  end  
  {n = N!}
```



# Graphical Notation for a Proof Outline Using Events

Towards Event-B



- ▶ *faci* = when  $n = 0$  then  $n := 1$  end
- ▶ *face* = when  $n > 0$  then  $m, n := n, n - 1$  end
- ▶ *facf* = when  $n \geq 0$  then  $n := n!$  end
- ▶ *facg* =  $n := m * n$

# Contents

Introduction

Modelling a Recursive Procedure

Recursive Program Development

Model and Proof

Abstract Program

Concrete Program

Program Termination

Mutual Recursion

Conclusion

# Contents

Introduction

Modelling a Recursive Procedure

**Recursive Program Development**

**Model and Proof**

Abstract Program

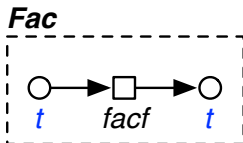
Concrete Program

Program Termination

Mutual Recursion

Conclusion

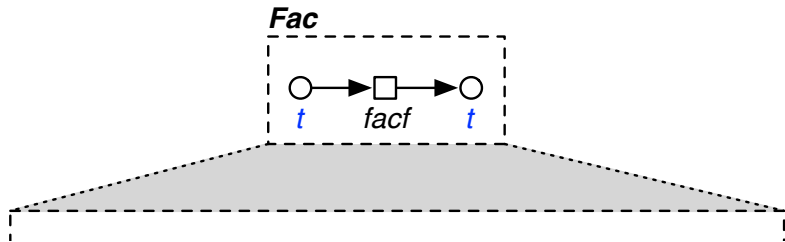
# Factorial Specification



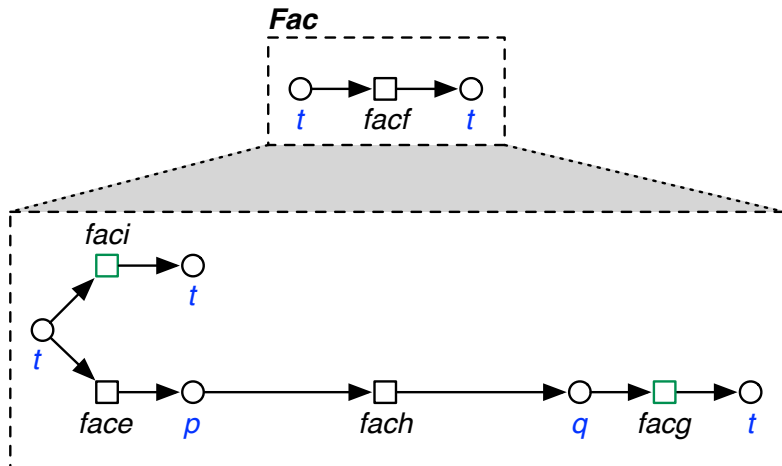
@t true

*fact* = when  $n \geq 0$  then  $n := n!$  end

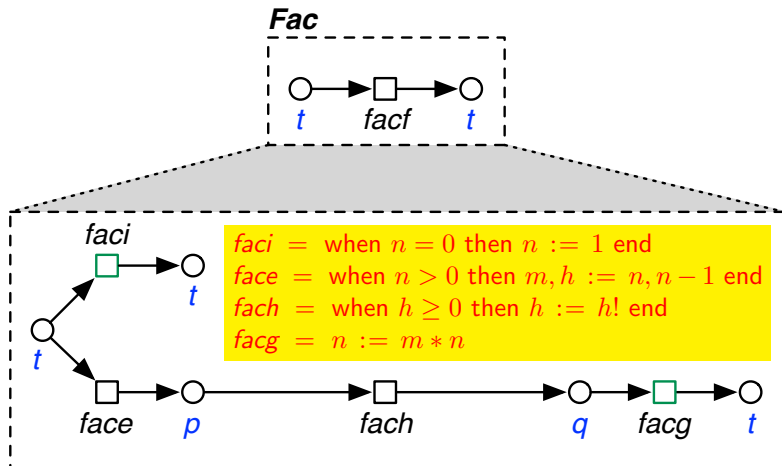
# Factorial Refinement Sketch



# Factorial Refinement Sketch



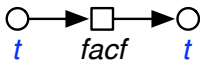
# Factorial Refinement Sketch



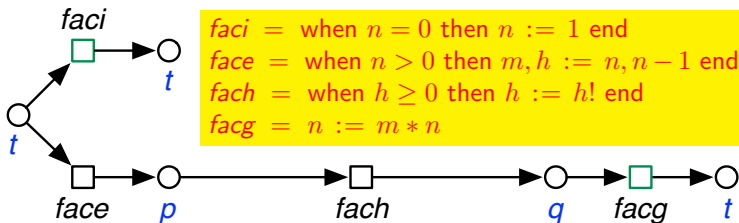
# Factorial Refinement Sketch

@*p*  $m = n$   
 $h \geq 0$   
 $h = n - 1$

**Fac**

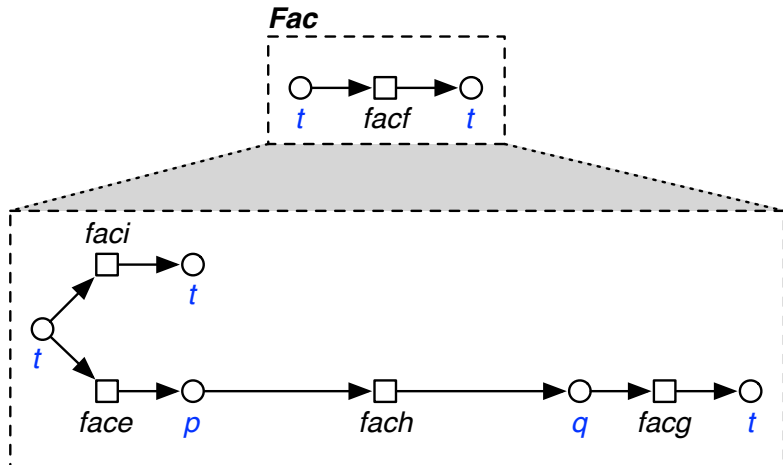


@*q*  $m = n$   
 $n - 1 \geq 0$   
 $h = (n - 1)!$

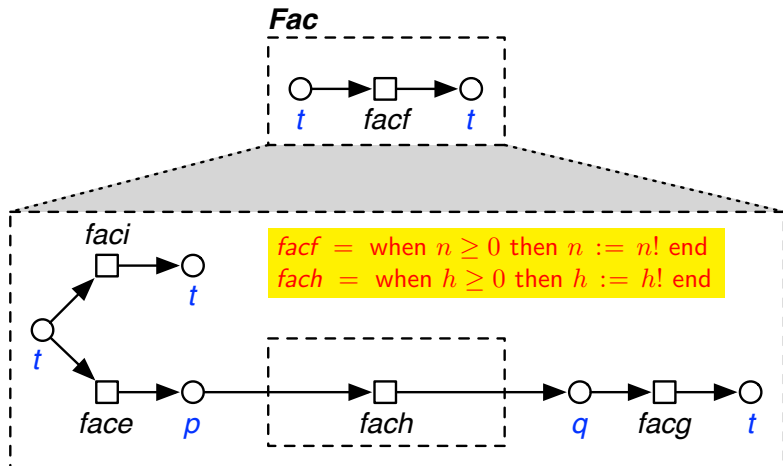




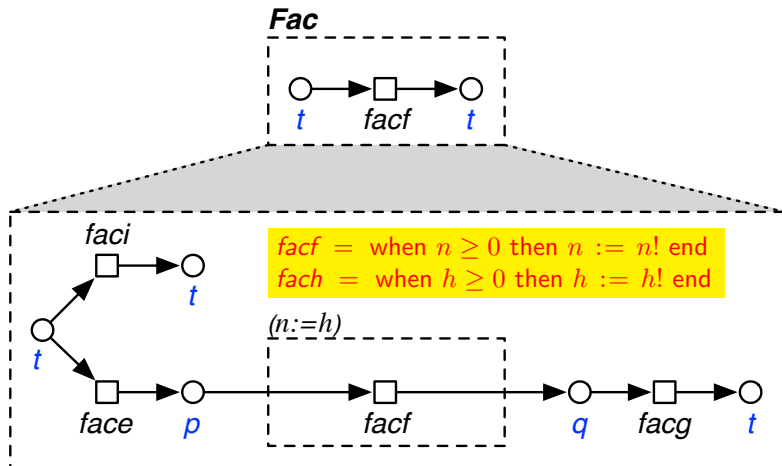
# Factorial Refinement Sketch



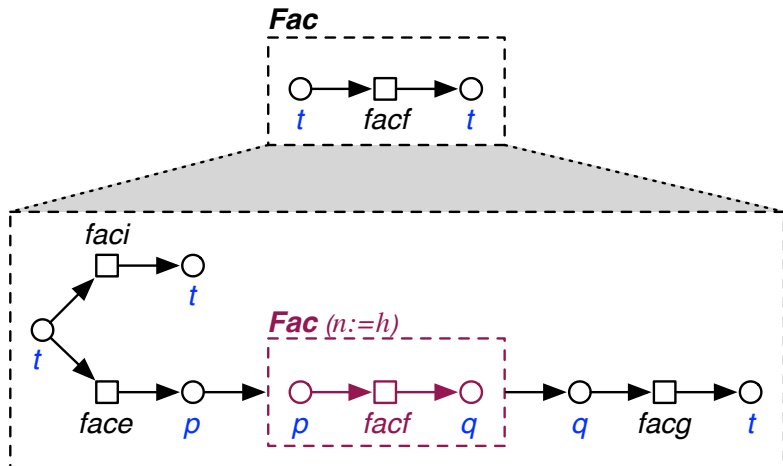
# A Recursive Reference?



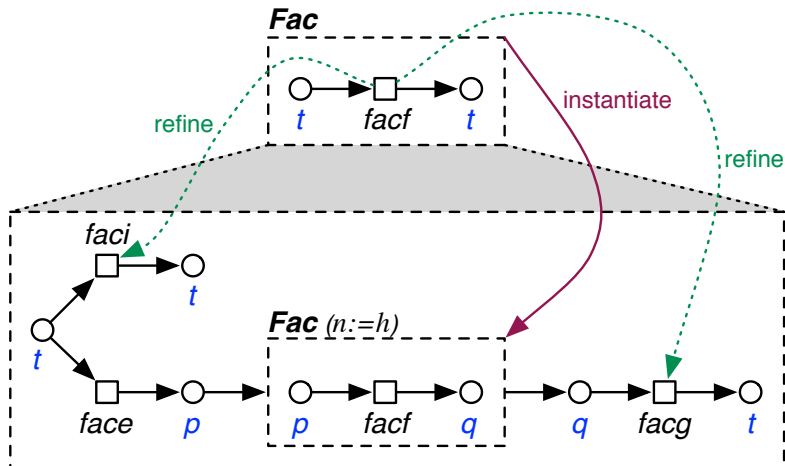
# A Recursive Reference?



# Recursion by Instantiation



# Completed Factorial Model



# Contents

Introduction

Modelling a Recursive Procedure

**Recursive Program Development**

Model and Proof

**Abstract Program**

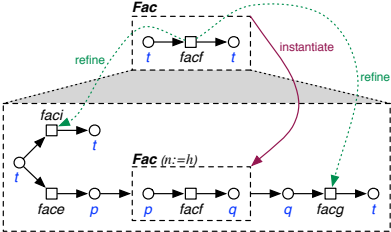
Concrete Program

Program Termination

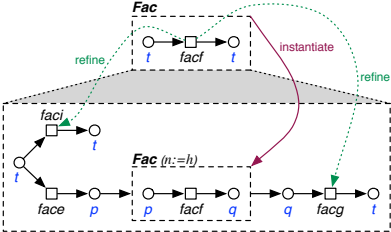
Mutual Recursion

Conclusion

# Extracted Factorial Program

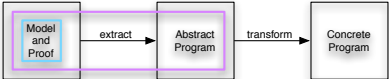
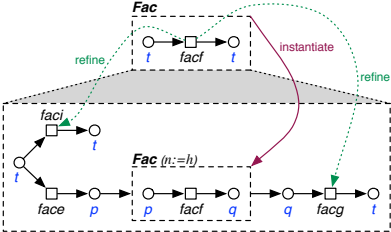


# Extracted Factorial Program

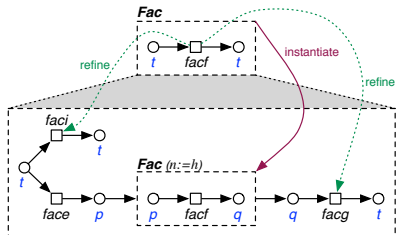




# Extracted Factorial Program

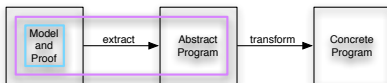


# Extracted Factorial Program

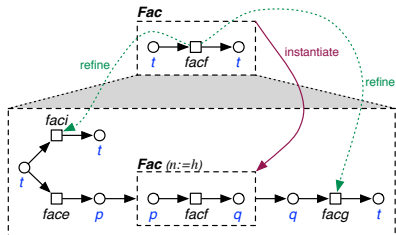


```

Fac(n) ::
  if n = 0 then
    n := 1
  else
    var m, h := n, n-1;
    Fac(h);
    n := m * h
  end
  
```

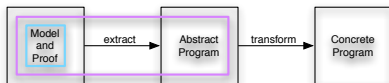


# Extracted Factorial Program

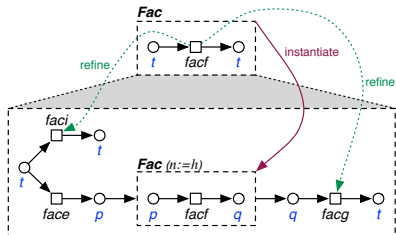


```

Fac(n) ::
  if n = 0 then
    n := 1
  else
    var m, h := n, n-1;
    Fac(h);
    n := m * h
  end
  
```



# Extracted Factorial Program



```

Fac(n) ::
  if n = 0 then
    n := 1
  else
    var m, h := n, n-1;
    Fac(h);
    n := m * h
  end
  
```



# Contents

Introduction

Modelling a Recursive Procedure

**Recursive Program Development**

Model and Proof

Abstract Program

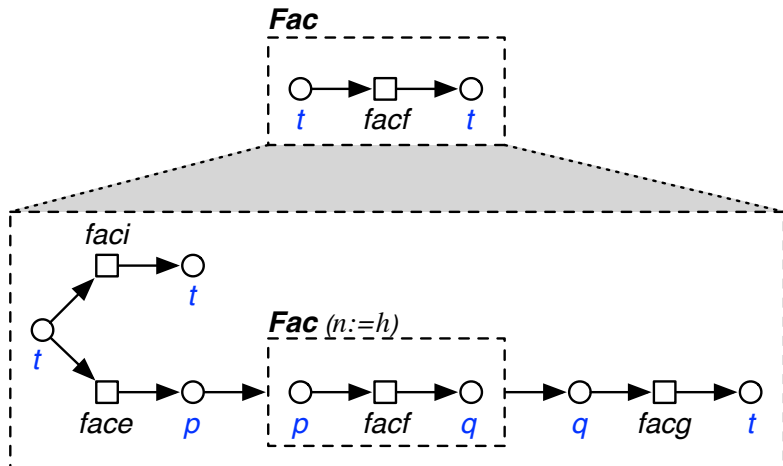
**Concrete Program**

Program Termination

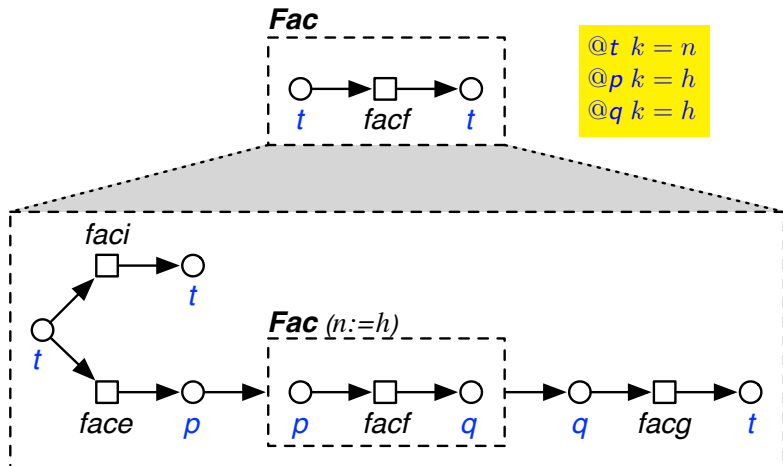
Mutual Recursion

Conclusion

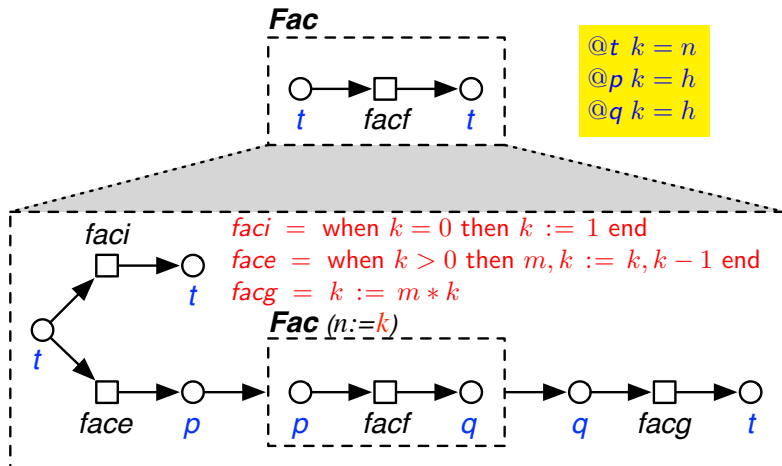
# Towards a Concrete Program



# Towards a Concrete Program



# Towards a Concrete Program





# Concrete Factorial Program

```
Fac(k) ::  
  if  $k = 0$  then  
     $k := 1$   
  else  
    var  $m := k$ ;  
     $k := k - 1$ ;  
    Fac(k);  
     $k := m * k$   
  end
```

# Concrete Factorial Program

```
Fac(k) ::  
  if k = 0 then  
    k := 1  
  else  
    var m := k;  
    k := k-1;  
    Fac(k);  
    k := m * k  
  end
```

transform  
→

```
global var k  
Fac ::  
  if k = 0 then  
    k := 1  
  else  
    var m := k;  
    k := k-1;  
    Fac;  
    k := m * k  
  end
```

# Concrete Factorial Program

$Fac(k) ::=$

if  $k = 0$  then

$k := 1$

else

var  $m := k$ ;

$k := k - 1$ ;

$Fac(k)$ ;

$k := m * k$

end

transform



global var  $k$

$Fac ::=$

if  $k = 0$  then

$k := 1$

else

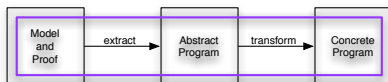
var  $m := k$ ;

$k := k - 1$ ;

$Fac$ ;

$k := m * k$

end



# Contents

Introduction

Modelling a Recursive Procedure

Recursive Program Development

Model and Proof

Abstract Program

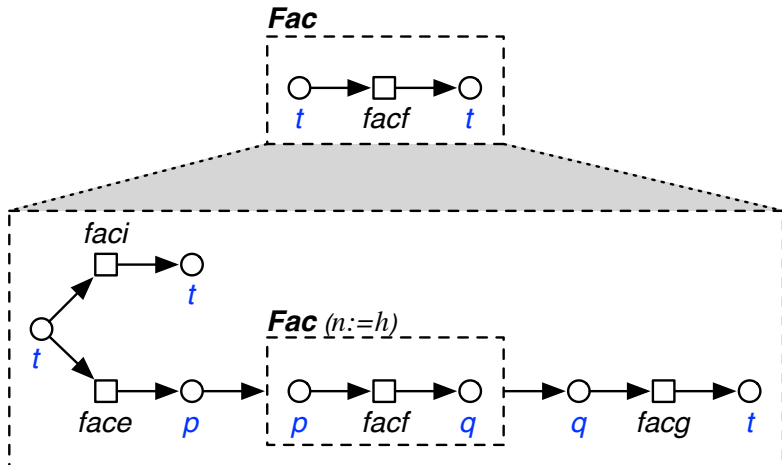
Concrete Program

Program Termination

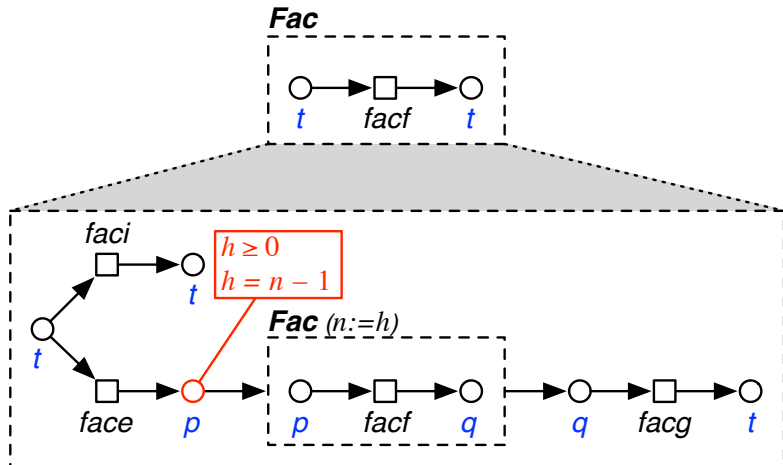
Mutual Recursion

Conclusion

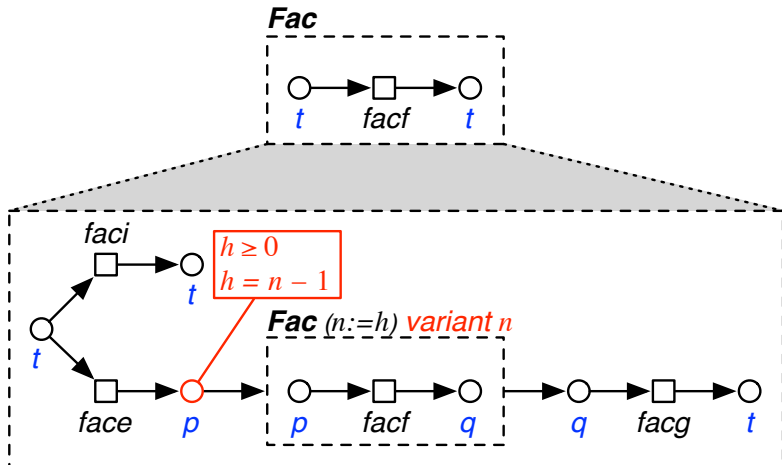
# Termination of the Factorial Procedure



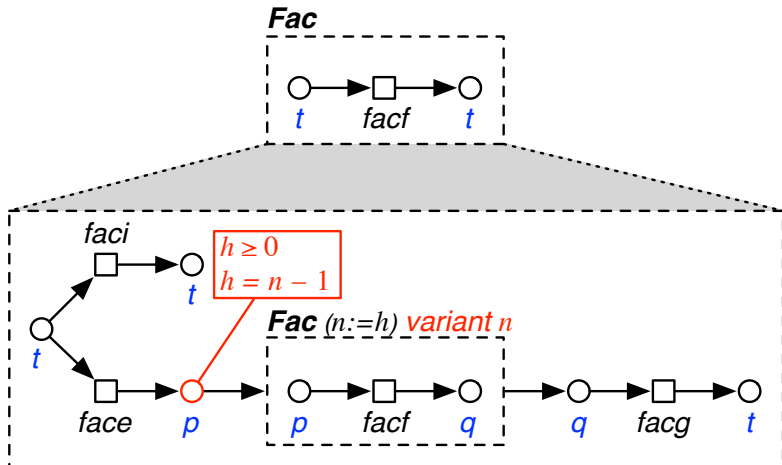
# Termination of the Factorial Procedure



# Termination of the Factorial Procedure



# Termination of the Factorial Procedure



To be proved:  $n \geq 0 \wedge n - 1 < n$



# Contents

Introduction

Modelling a Recursive Procedure

Recursive Program Development

Model and Proof

Abstract Program

Concrete Program

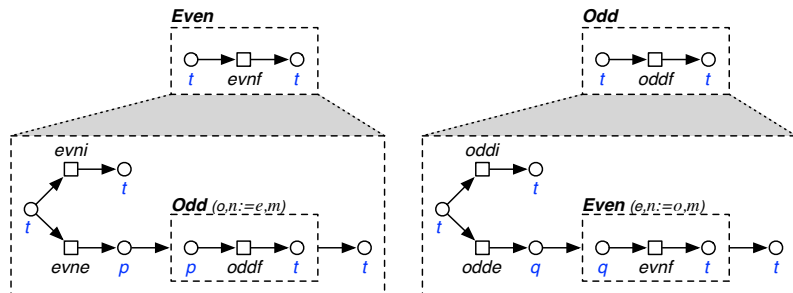
Program Termination

**Mutual Recursion**

Conclusion

# Specification of EvenOdd

“Defunctionalisation” (J. C. Reynolds)



$evnf =$  when  $n \geq 0$  then  $e := "n \bmod 2 = 0"$  end

$evni =$  when  $n = 0$  then  $e := \text{TRUE}$  end

$evne =$  when  $n > 0$  then  $m := n - 1$  end

@ $p$   $m = n - 1$

$oddf =$  when  $n \geq 0$  then  $o := "n \bmod 2 \neq 0"$  end

$oddi =$  when  $n = 0$  then  $o := \text{FALSE}$  end

$odde =$  when  $n > 0$  then  $m := n - 1$  end

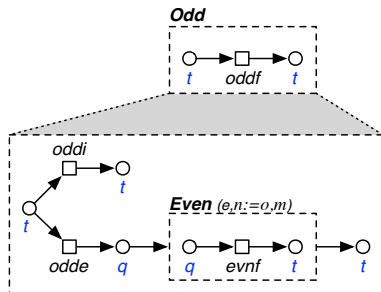
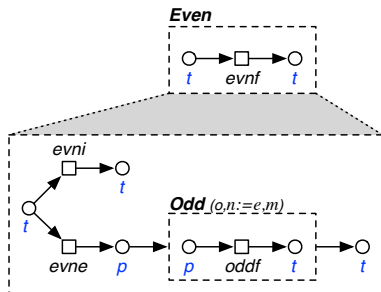
@ $q$   $m = n - 1$

*Heinrich Heine*

HEINRICH HEINE  
UNIVERSITÄT DÜSSELDORF

# Specification of EvenOdd

“Defunctionalisation” (J. C. Reynolds)



$Even(e, n) ::$   
 if  $n = 0$  then  
      $e := \text{TRUE}$   
 else  
     var  $m := n - 1$ ;  $Odd(e, m)$   
 end

$Odd(o, n) ::$   
 if  $n = 0$  then  
      $o := \text{FALSE}$   
 else  
     var  $m := n - 1$ ;  $Even(o, m)$   
 end

# Contents

Introduction

Modelling a Recursive Procedure

Recursive Program Development

Model and Proof

Abstract Program

Concrete Program

Program Termination

Mutual Recursion

Conclusion

# Conclusion

- ▶ Method for sequential program development
  - ▶ Recursion
  - ▶ Mutual Recursion
  - ▶ Termination
- ▶ To be investigated:
  - ▶ Modularity
  - ▶ Concurrency
  - ▶ Soundness