Ushakova Mariya Sergeevna

Formal Verification of Data Driven Functional Parallel Programs

Based on PhD thesis

Scientific adviser: Doctor of Sciences in Technology, professor
Legalov Alexander Ivanovich

Trends and Challenges

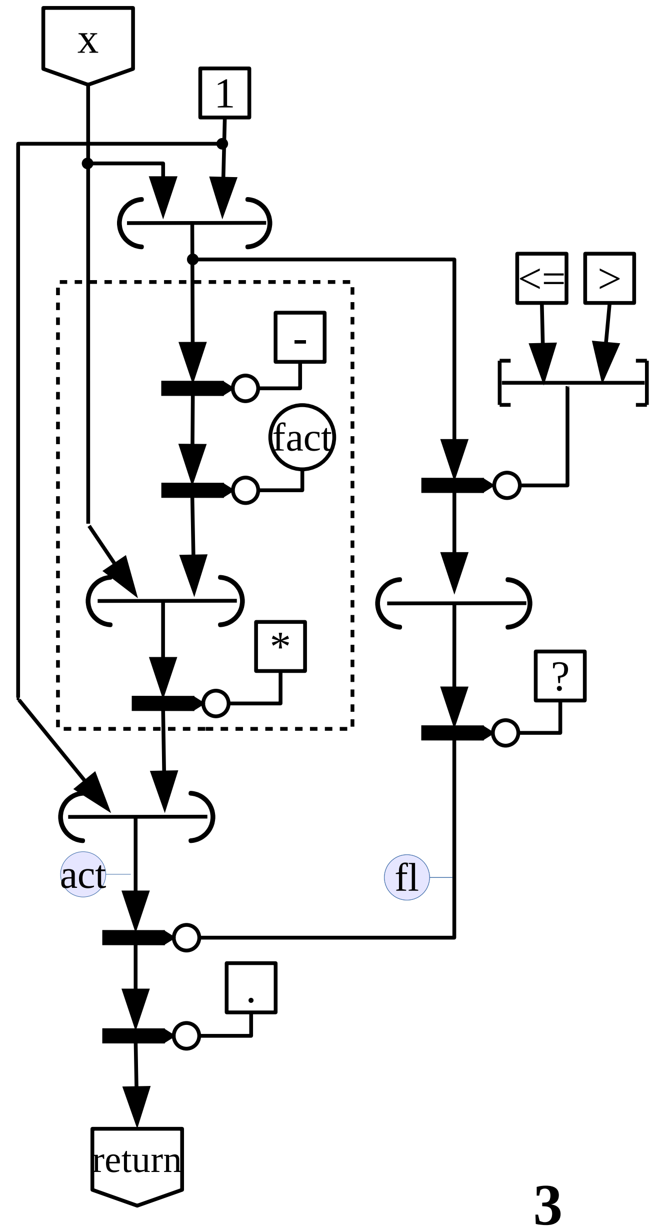
- **The increase of programs complexity** makes it essential to ensure the logical correctness of the program.
- **Concurrency** causes new types of errors:
  – deadlocks,
  – process races,
  – asynchrony,
  – resource conflicts.
- **Creating imperative style parallel programs** results in:
  – the need to simultaneously control the program logic, resources and interaction of processes,
  – the variety of approaches to parallel programming.

Data Driven Functional Parallel (DDFP) Programming

- resources are unlimited,
- the program is an acyclic data flow graph,
- calculations start on data readiness,
- parallelism is implemented at the level of operations,
- no loops, iterative calculations are implemented through recursion,
- architecture independence.

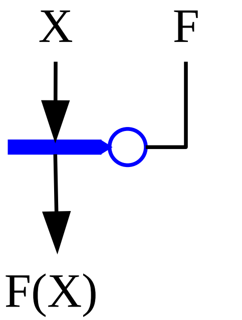
```
fact << funcdef x {
     fl << ((x,1):[<=,>]):?;
     act << (1,
             { (x, (x,1):-:fact ):*  } );
     return << act:fl:.;
}
```

Source code of the function **fact**, that calculates the factorial

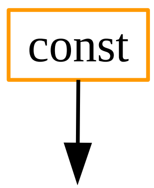


Data flow graph of **fact**

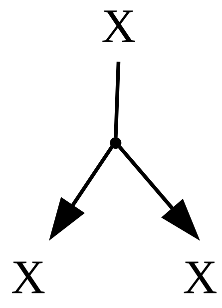
Program-Forming Operators



interpretation operator

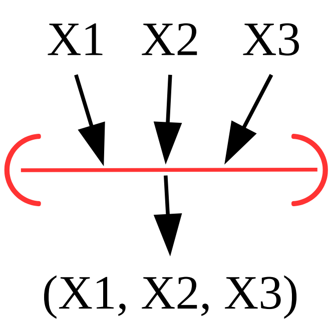


constant operator

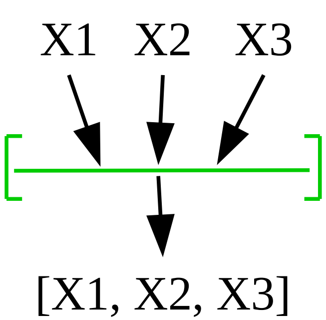


data copying operator

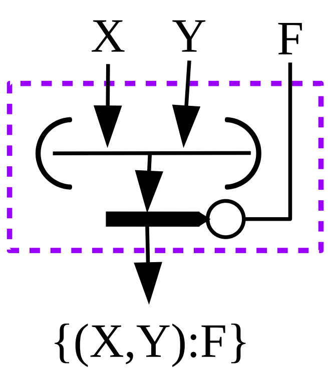
Data grouping operators:



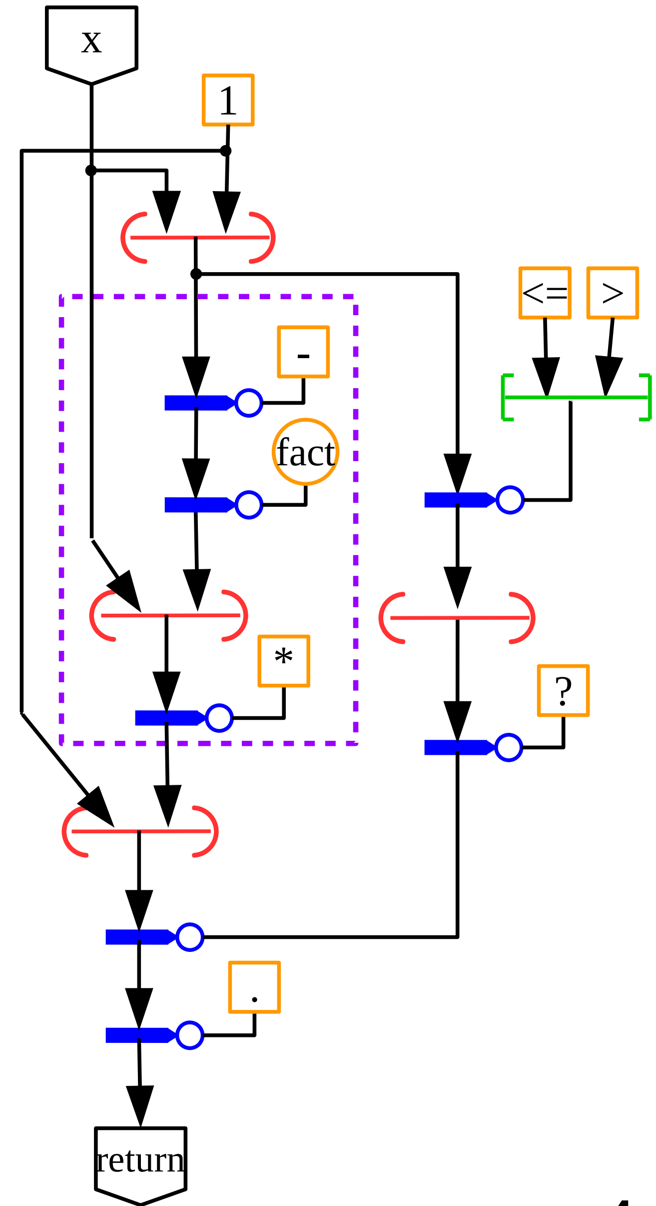
data list



parallel list

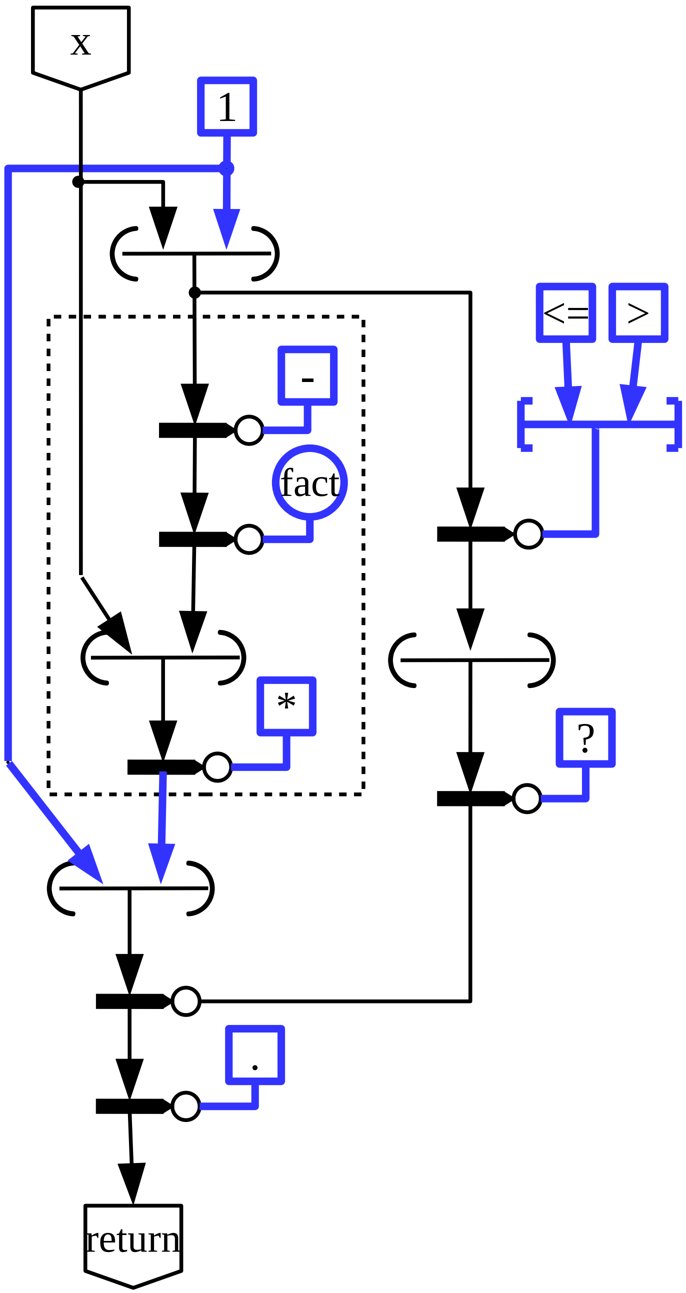


delay list

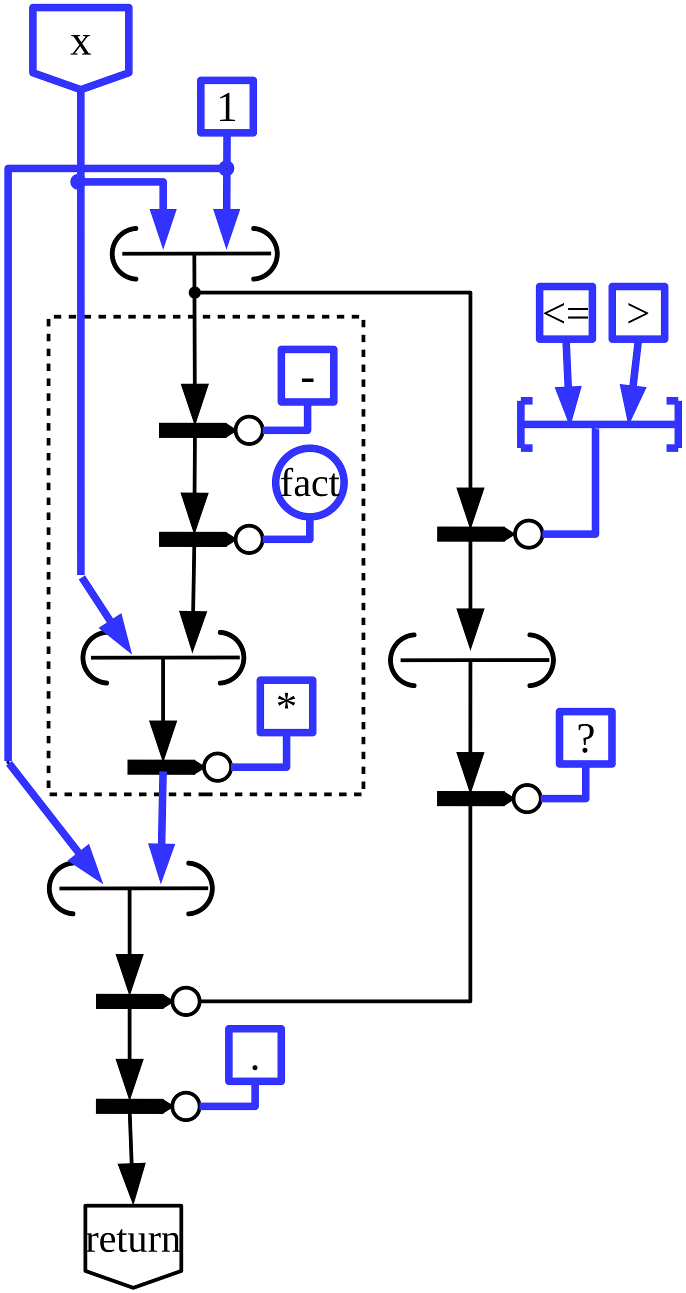


Data flow graph of **fact**

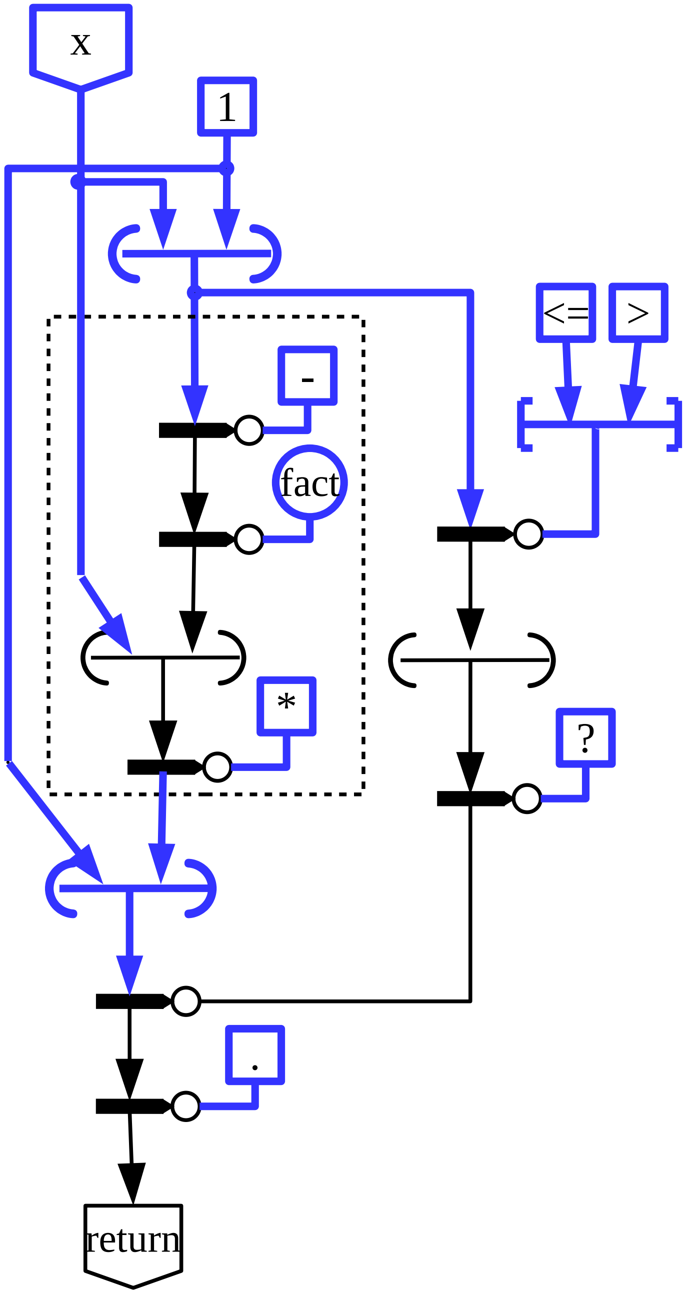
Program Execution



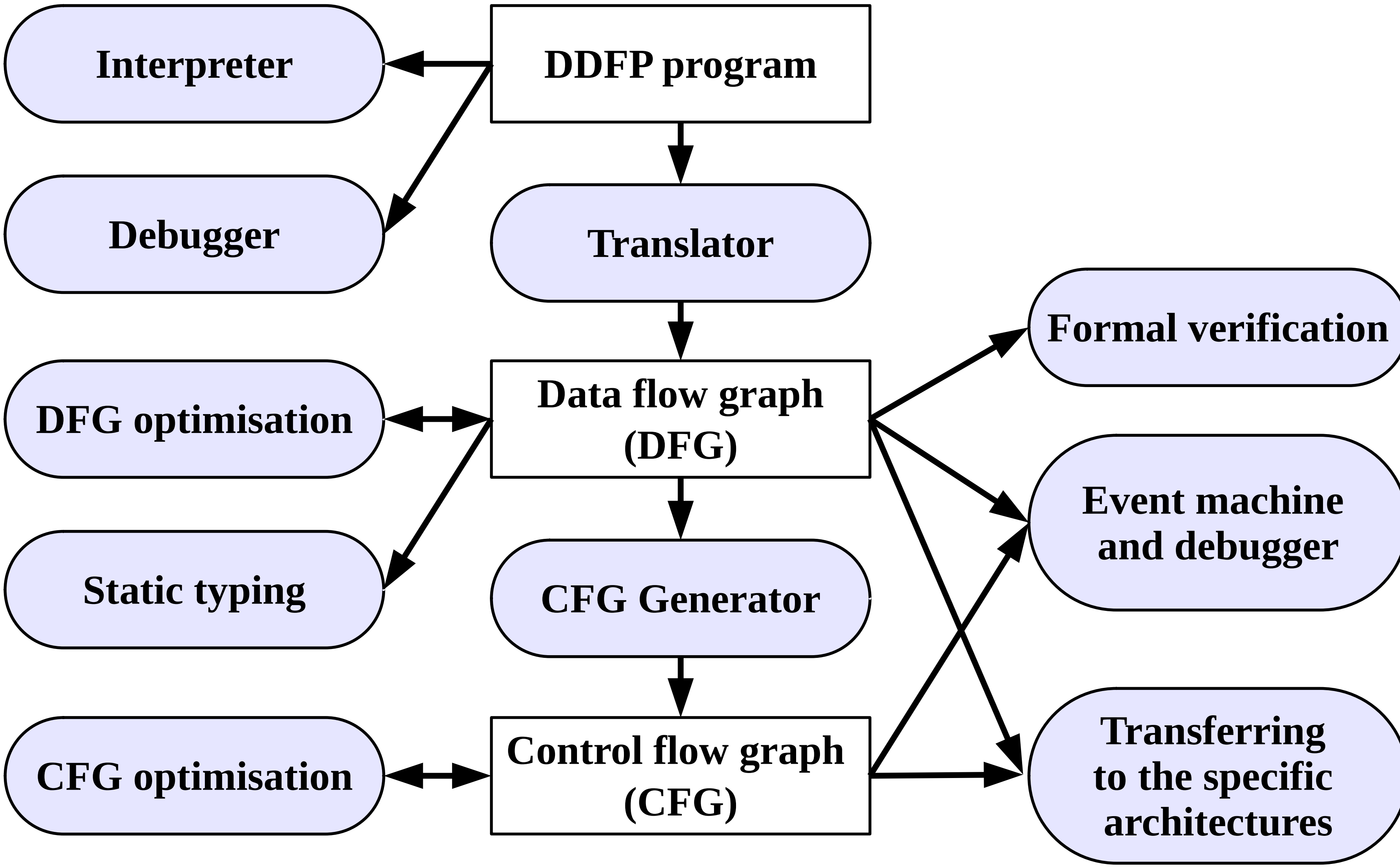
Program Execution



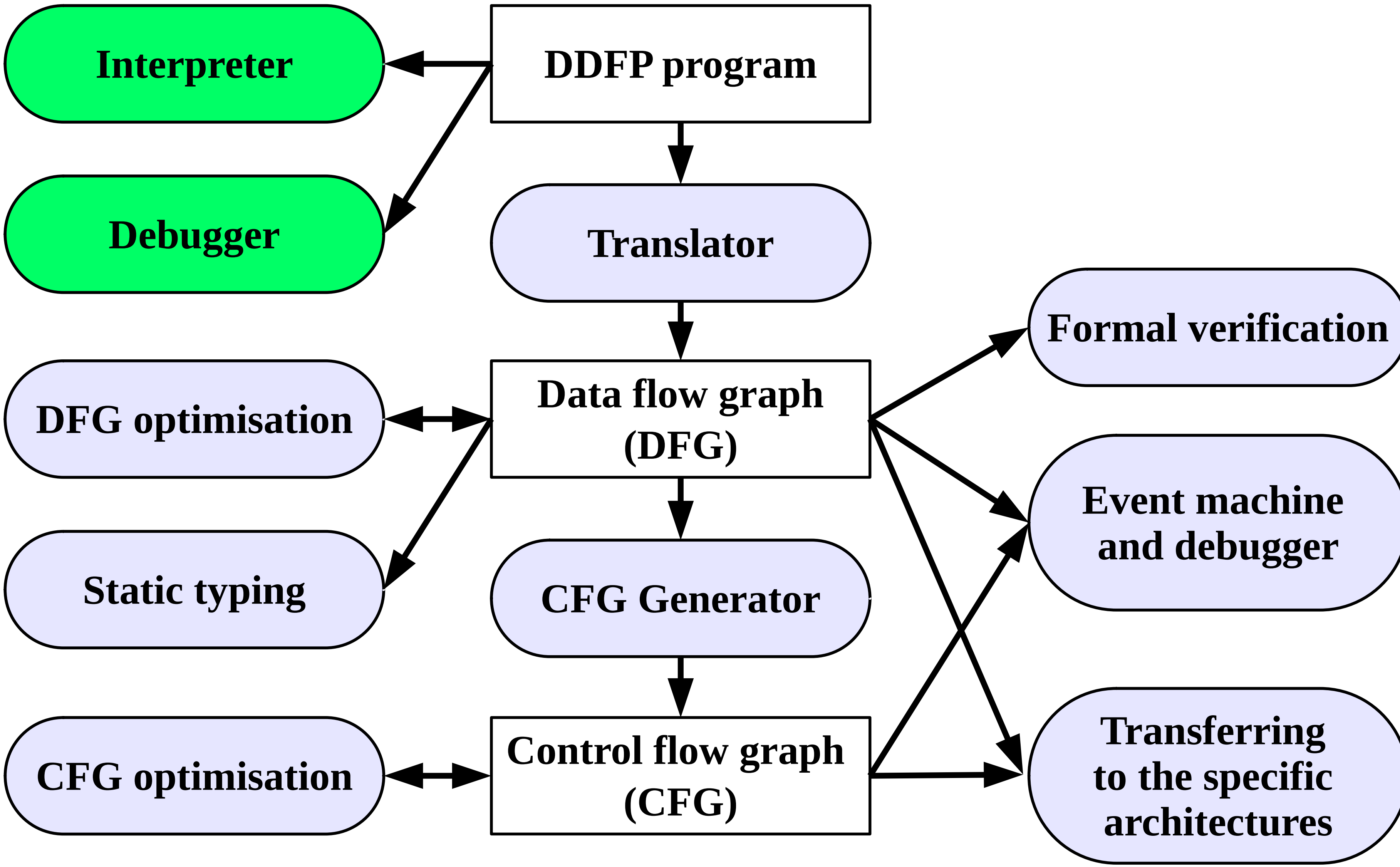
Program Execution



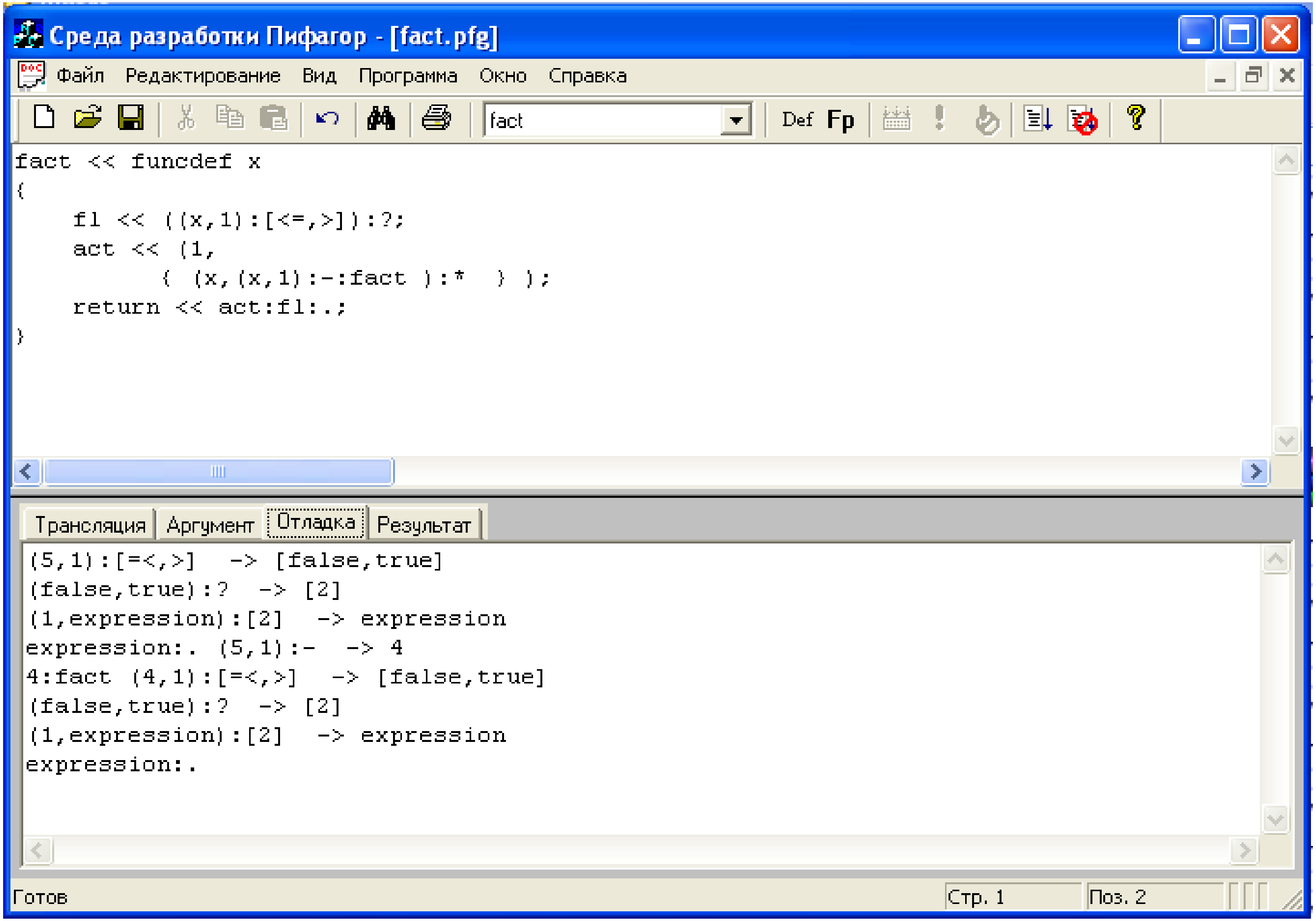
The Toolkit for Architecture-Independent Parallel Programming



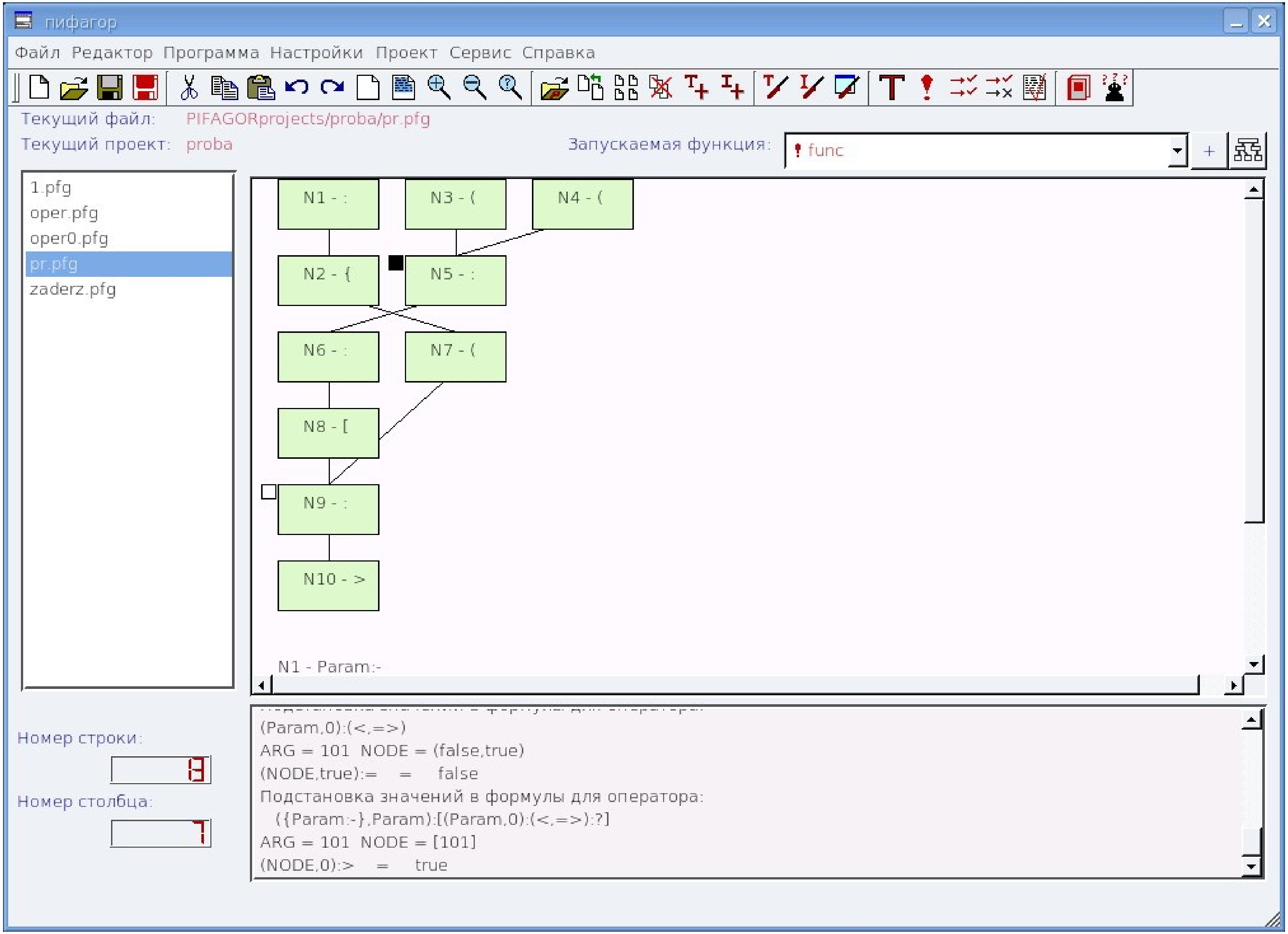
The Toolkit for Architecture-Independent Parallel Programming



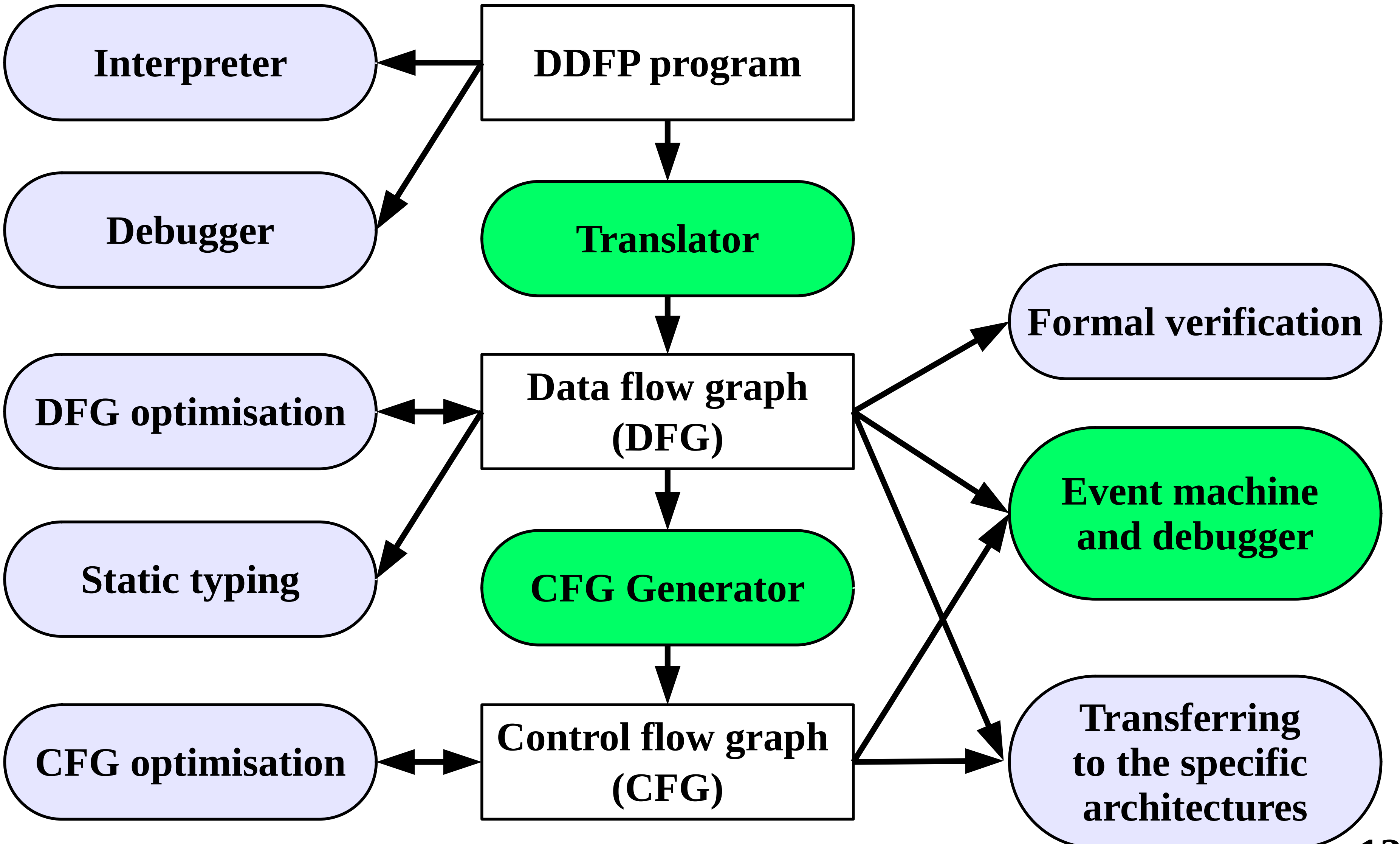
Interpreter of DDFP Programs (Privalikhin D.V.)



Debugger and Verifier of DDFP Programs (Udalova J. V.)



The Toolkit for Architecture-Independent Parallel Programming



Translator into Data Flow Graph

Source code of function **abs**, calculating the absolute value

```
abs << funcdef arg{
   ({arg:-},arg):
   [((arg,0):
   [<,>=]):?]:. >> return
}
```

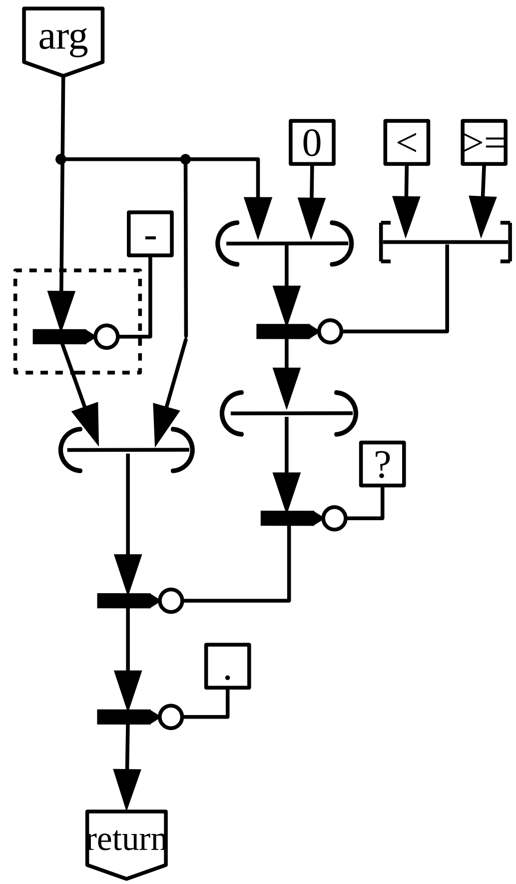
abs — function name;

arg — input argument;

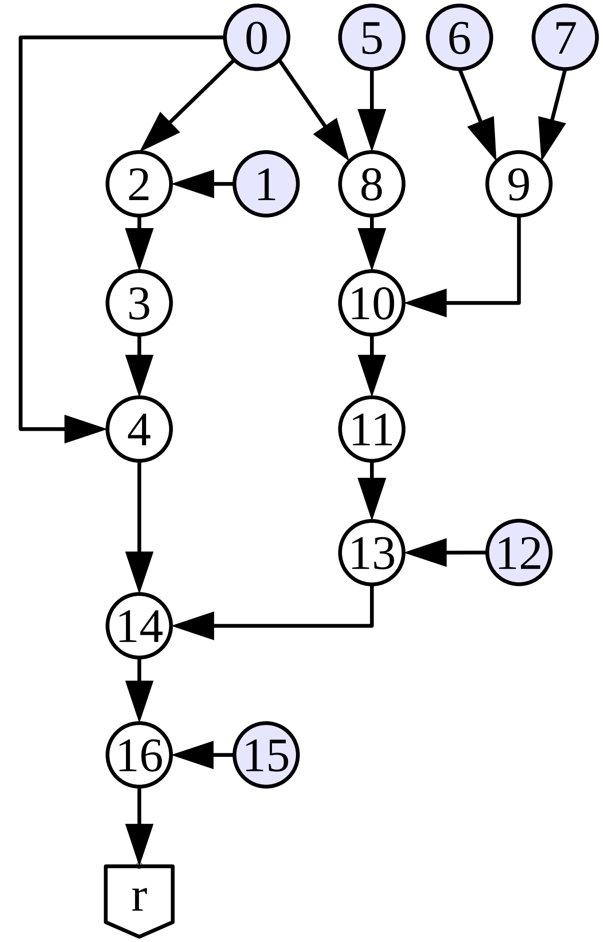
g : f — application of the function **f** to the argument **g**;

g << **f** — assigning the identifier **g** to the result of code **f** execution;

return — function return value.

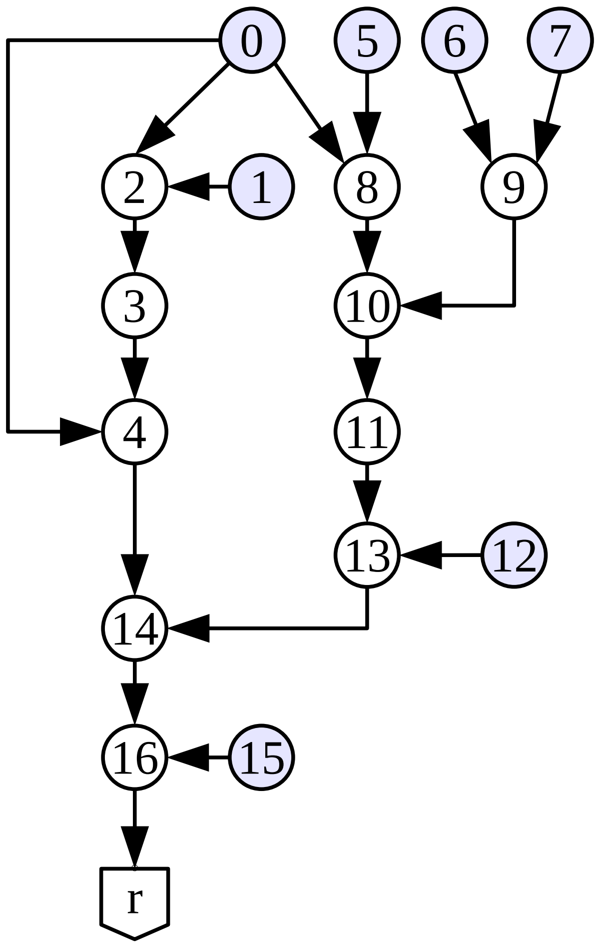


Data flow graph of **abs**

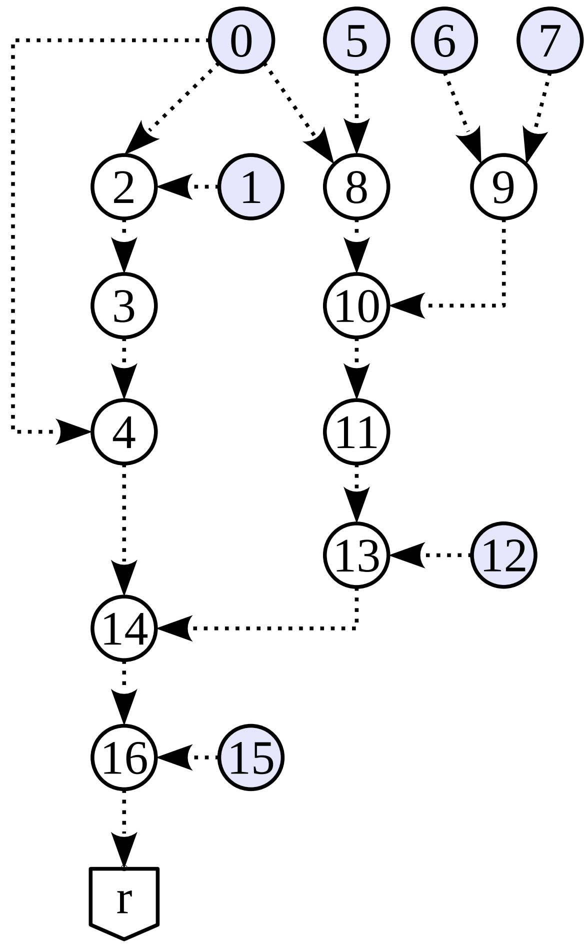


Scheme of data flow graph

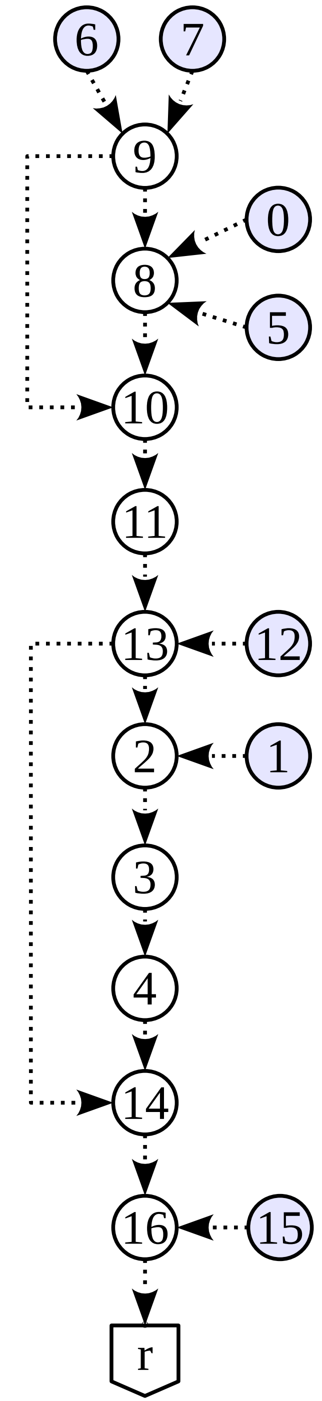
Generator of Control Flow Graph



Scheme of data flow graph (DFG)

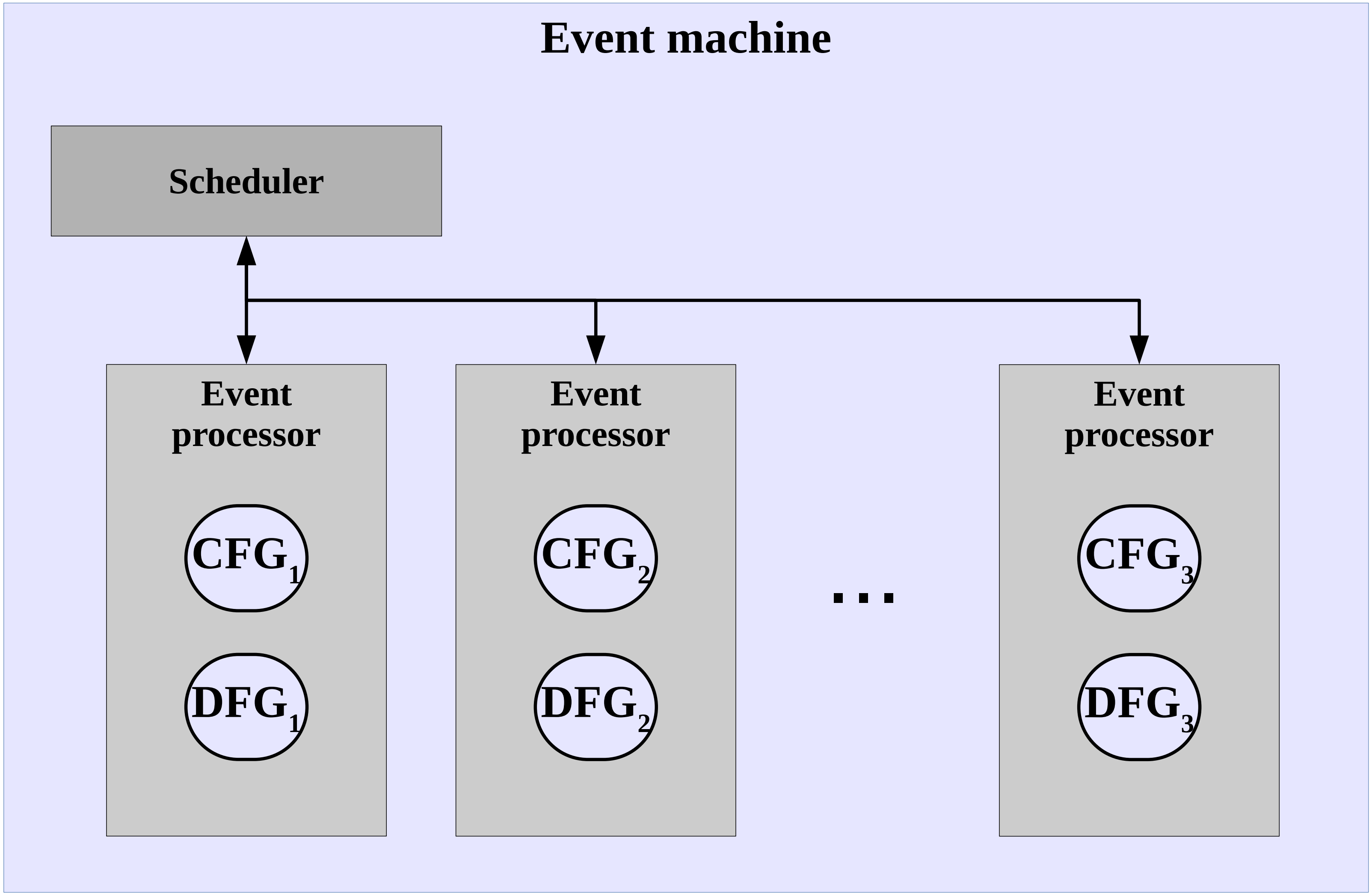


Scheme of control flow graph (CFG)

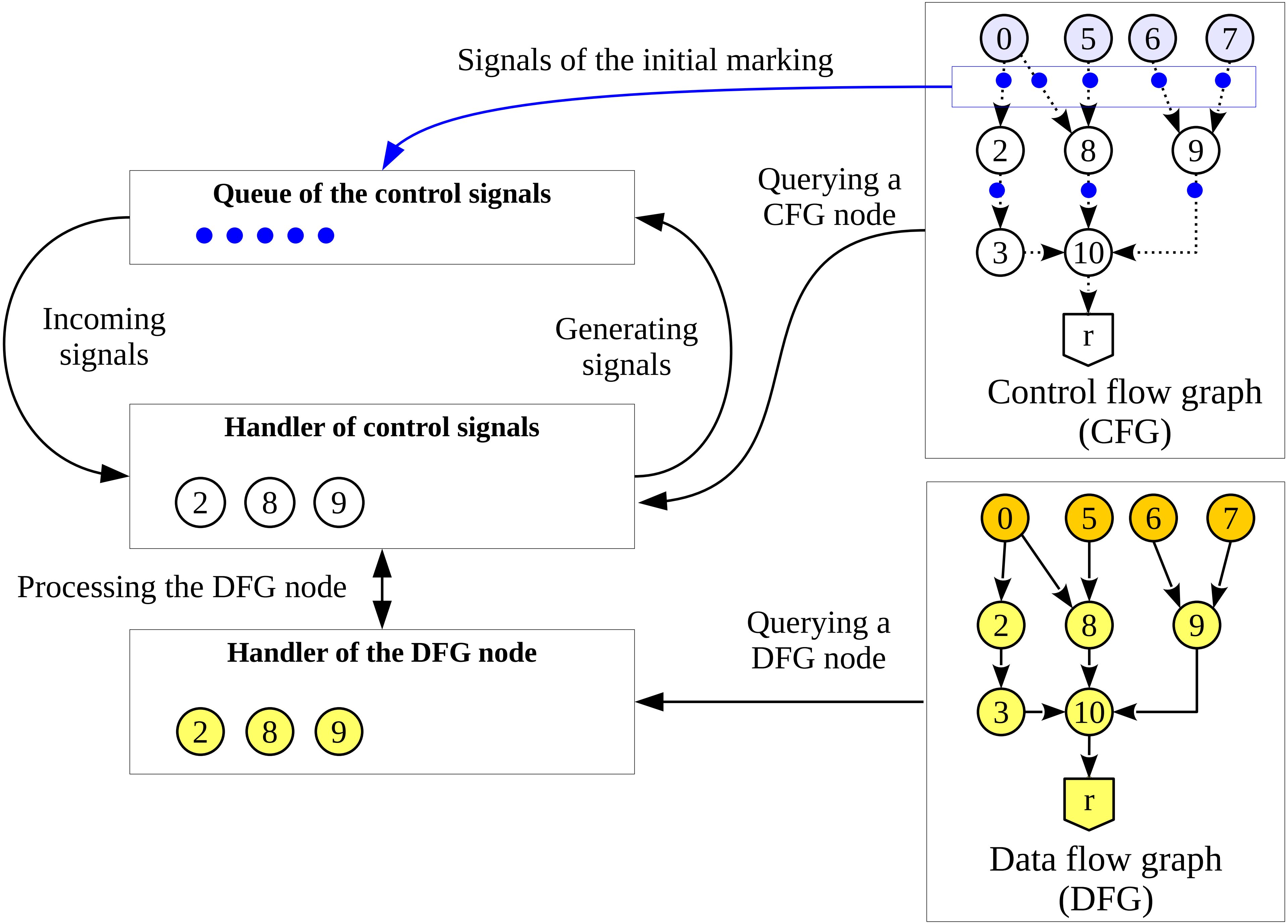


Modified CFG

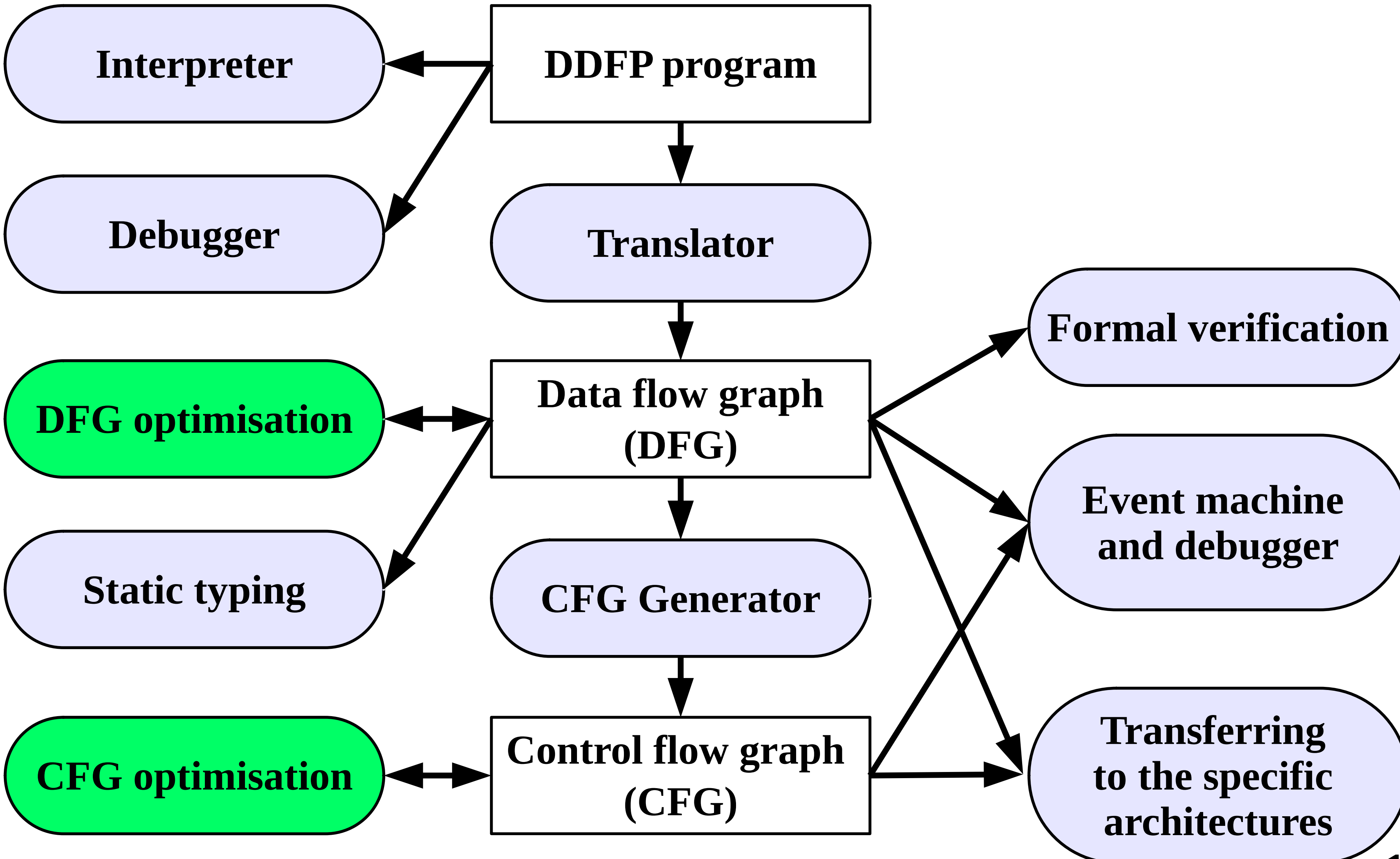
Event Machine (Matkovskii I.V.)



Event Processor (Matkovskii I.V.)



The Toolkit for Architecture-Independent Parallel Programming



Optimisation of Data Flow Graph (Vasilyev V.S.)

1. **Invariant optimisation** is the motion of code from recursions or massive operations of a parallel list, that do not depend on the recursion argument or the number of the element in the list;
2. **Dead-code elimination** is the removal of code that does not affect the program result;
3. **Duplicate-code elimination (elimination of mutual subexpressions)** is the search of same subgraphs and their replacement by one subgraph with the help of the data copying operator;
4. **Optimisations based on equivalent transformations**, that are determined by algebra of data driven functional parallel computing model.

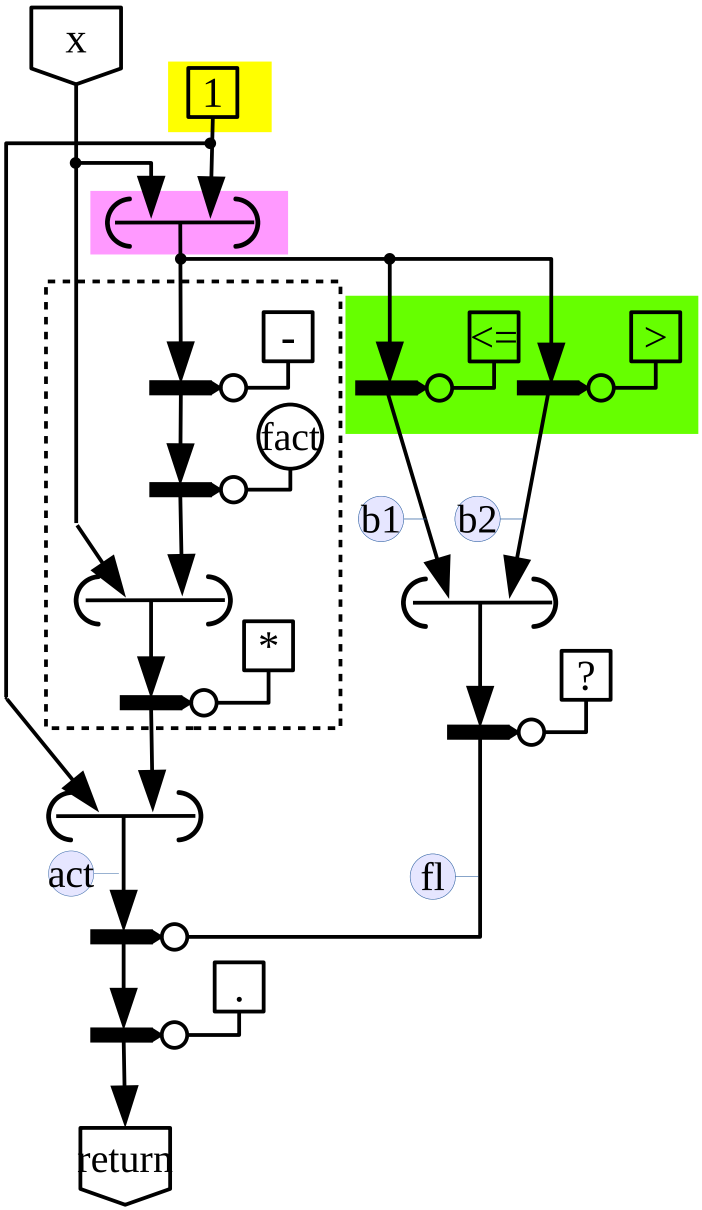
Optimisation of Data Flow Graph (Vasilyev V.S.)

```
fact << funcdef x {
     fl << ((x,1):[<=,>]):?;
     act << (1,
            { (x, (x,1):-:fact ):*  } );
     return << act:fl:.;
}
```

Source code of the function **fact**, that calculates the factorial

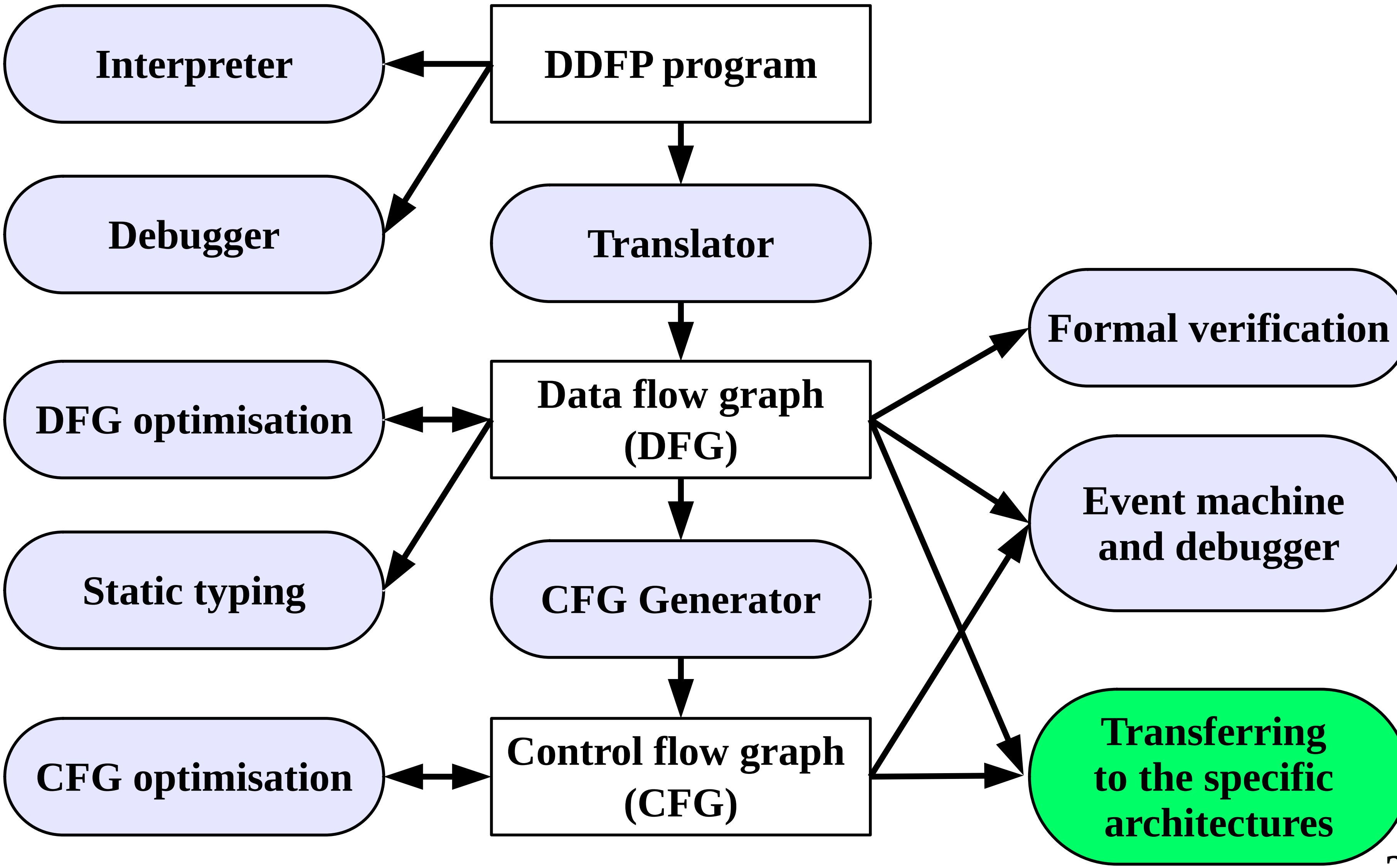
```
fact1 << funcdef x {
  one << 1;
  x_1 << (x,one);
  b1 << x_1:<=;
  b2 << x_1:>;
  fl << (b1,b2):?;
  act << (one,
          { (x,x_1:-:fact1 ):*  } );
  return << act:fl:.
}
```

Optimised source code of the function **fact**



Data flow graph of **fact** after optimisation

The Toolkit for Architecture-Independent Parallel Programming

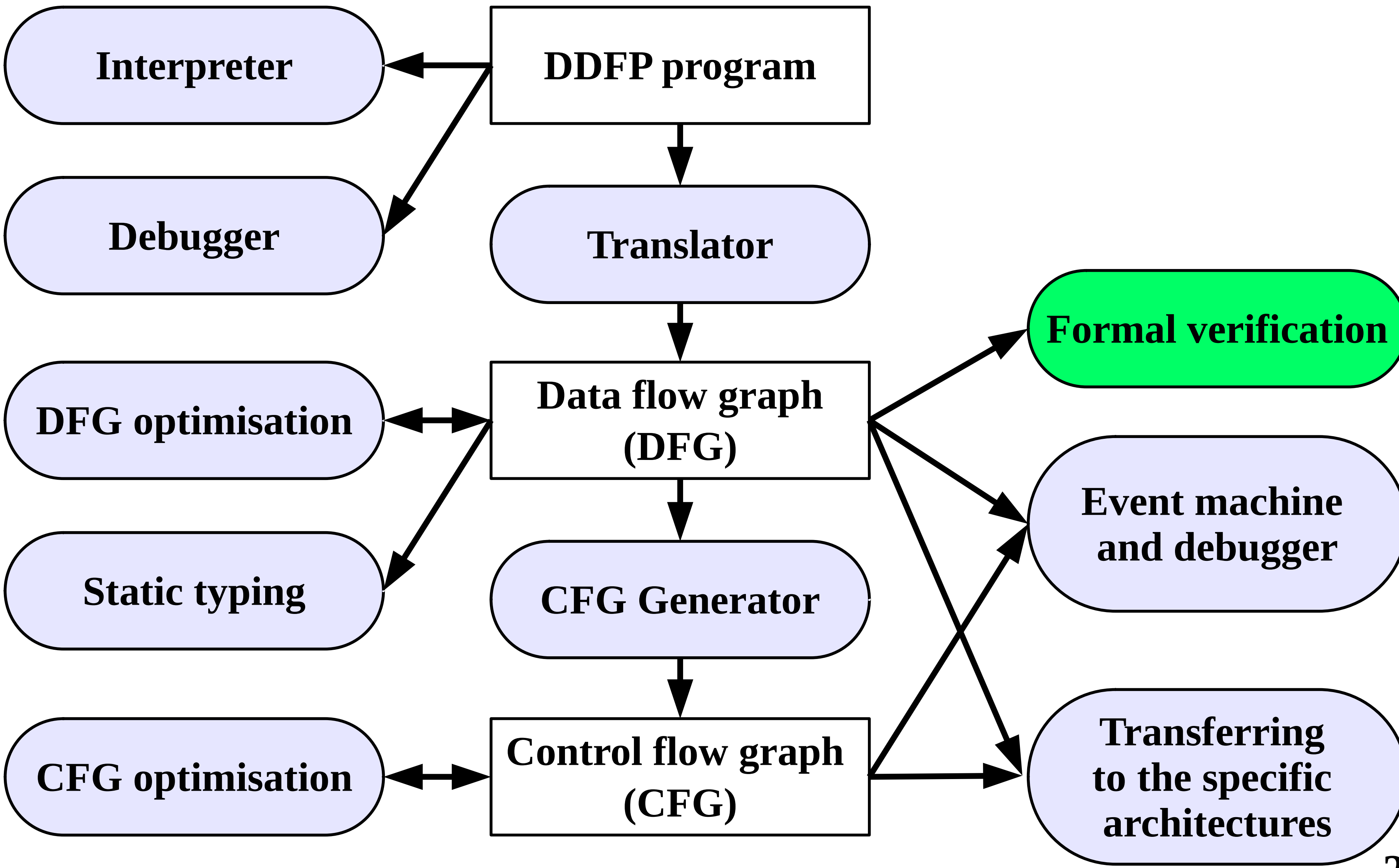


Transferring to the Specific Architectures

1. Transformation of subset of programs in Pifagor language to **C++ language** (Vasilyev V.S.)

2. Transformation of programs in Pifagor language to **Verilog** and **VHDL** language for creating very large integrated circuits:
   2.1. transformation of ordinary DDFP programs with some restrictions to Verilog and VHDL (Ryzhenko I.N.);
   2.2. design of combinational circuit directly in the Pifagor language (Romanova D.S.)

The Toolkit for Architecture-Independent Parallel Programming



Known Results

- Basic approaches to formal verification:
  – model checking,
  – theorem proving,
  – different variants of program refinement.

- Toolkits for formal verification:
  – Boogie (C, Dafny, Java bytecode, Eiffel),
  – C-lightVer (formerly called SPECTRUM),
  – LIQUID HASKELL (Haskell),
  – Predicate programs verifier.

- Automated theorem provers:
  HOL, Coq, Isabelle, PVS.

- Works of Udalova J.V.:
  – debugging of DDFP programs,
  – using of verification methods for data correctness analysis.

Errors in DDFP Programs

- Errors in program semantics.
- Program nontermination.
  - Occur only as a result of infinite recursion.
  - There are no program crashes because there are no partially defined functions in the language.
- There are no errors caused by limited resources, that are typical of parallel programs.

Analysis of the correctness of DDFP programs is reduced to the analysis of errors similar to errors in sequential programs.

Application of Formal Verification Methods to DDFP Programs

Method selection criteria:

- The proof of correctness is carried out for the already written programs.
- The specification language should fully describe the logic of the program.
- Applicability to a wide class of problems.
- Capability to automate the proof process.
- Simplicity of the method.

Basic group of methods of formal verification:

- model checking,
- theorem proving,
- program refinement.

Methods for Formal Verification Applicable to DDFP Programs

Method based on the Hoare logic

The Hoare triple:

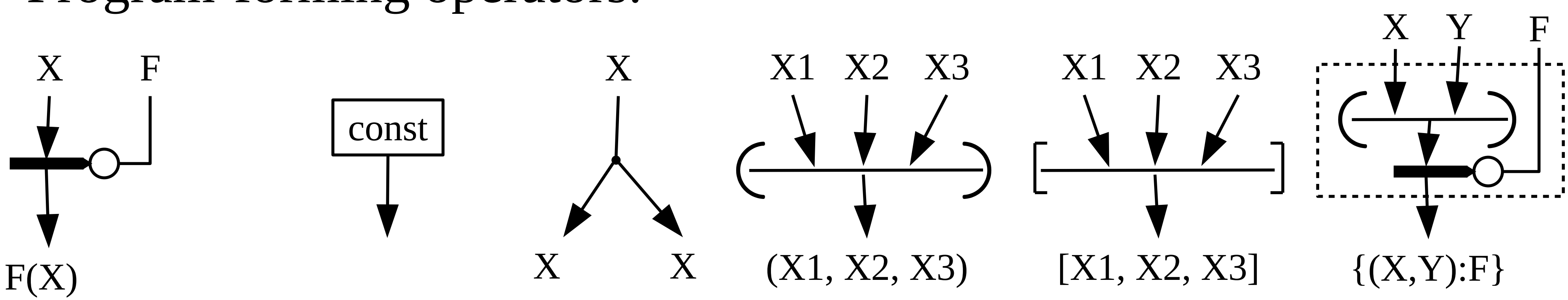
Precondition	Program code	Postcondition

Method for proving program termination using the bound function

- the basis is in the decreasing the value of the bound function at each iteration;
- allows to extend the method based on the Hoare logic

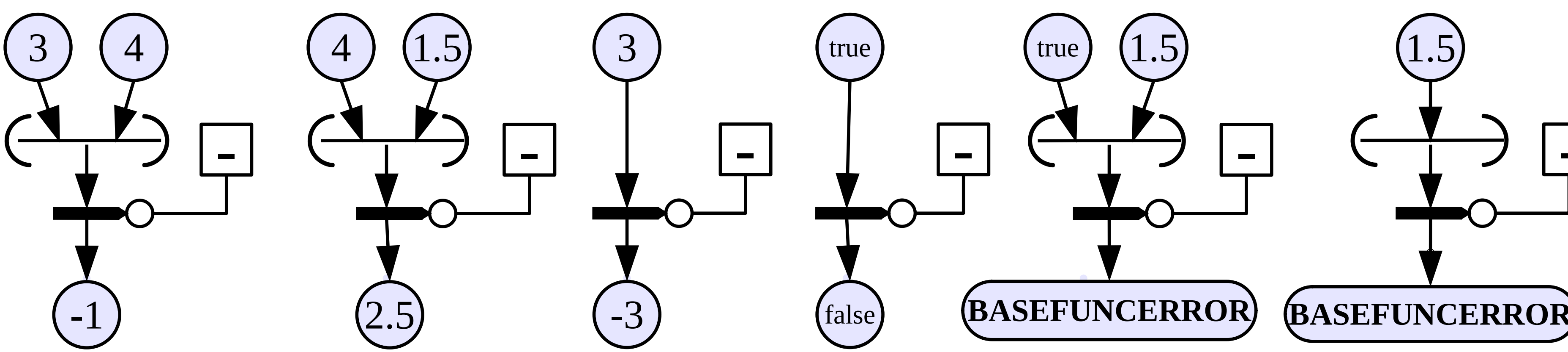
Formal Semantics for DDFP Programming Language

Program-forming operators:



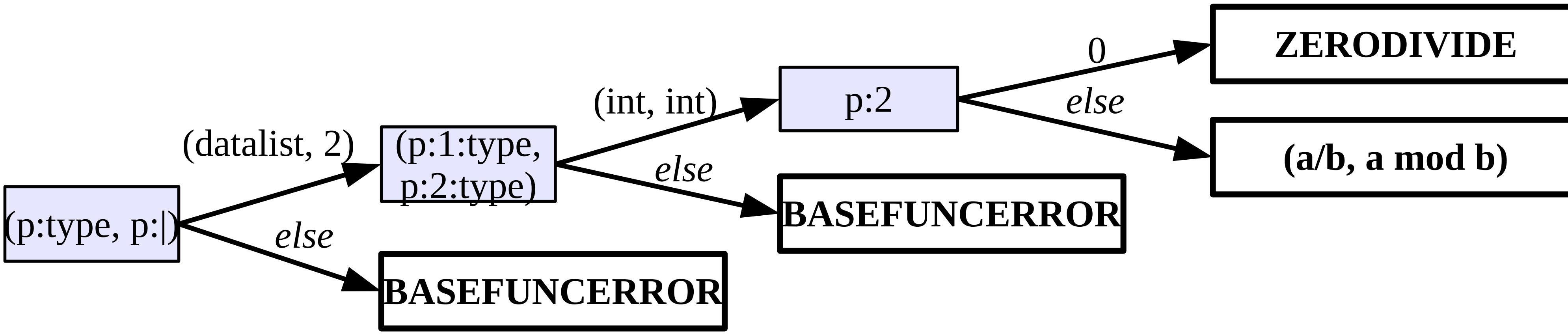
interpretation operator | constant operator | data copying operator | operators of grouping in data list, parallel list, delay list

Built-in functions of Pifagor language are completely defined. Examples of the function “minus” execution:



Formal Semantics for DDFP Programming Language

Semantic rule for built-in function of integer division with remainder (p is the argument) :



The Hoare triples for function of integer division with remainder:

$$\boxed{p=(a:\mathrm{int}, b:\mathrm{int})\wedge(p[2]\neq 0)}\ \ \mathrm{p:\%}\rightarrow \mathrm{r}\ \ \boxed{r=((a/b):\mathrm{int}, (a \bmod b):\mathrm{int})}$$

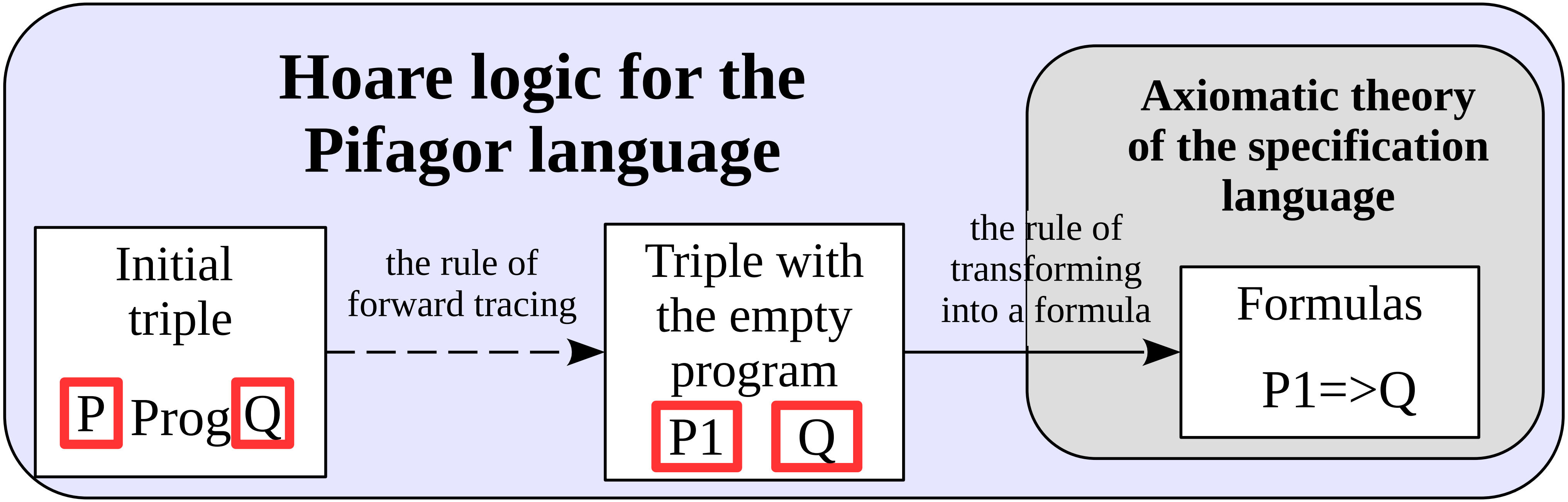
$$\boxed{p=(a:\mathrm{int}, b:\mathrm{int})\wedge(p[2]=0)}\ \ \mathrm{p:\%}\rightarrow \mathrm{r}\ \ \boxed{r=\mathrm{ZERODIVIDE}}$$

$$\boxed{p\neq(a:\mathrm{int}, b:\mathrm{int})}\ \ \mathrm{p:\%}\rightarrow \mathrm{r}\ \ \boxed{r=\ \mathrm{BASEFUNCERROR}}$$

Method for Proving the Correctness of DDFP Programs Based on the Hoare Logic

To prove the correctness of DDFP programs, an axiomatic theory for the Pifagor language is constructed:

– objects are Hoare triple,
– axioms are Hoare triples for built-in functions,
– rules of inference are introduced.

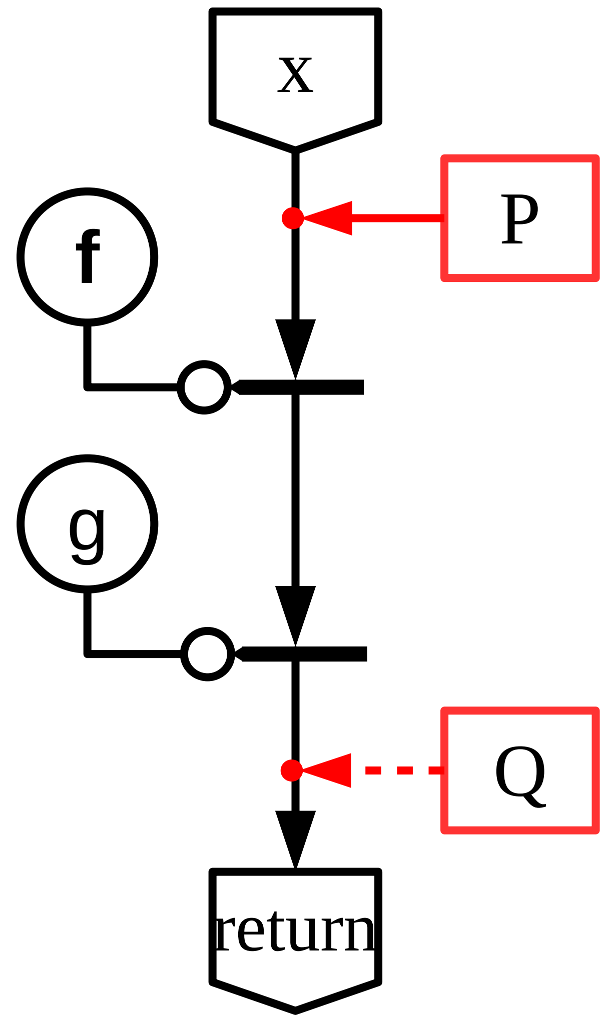


Graphical Representation of a Hoare Triple

An example of a function in Pifagor that calculates the composition of two functions

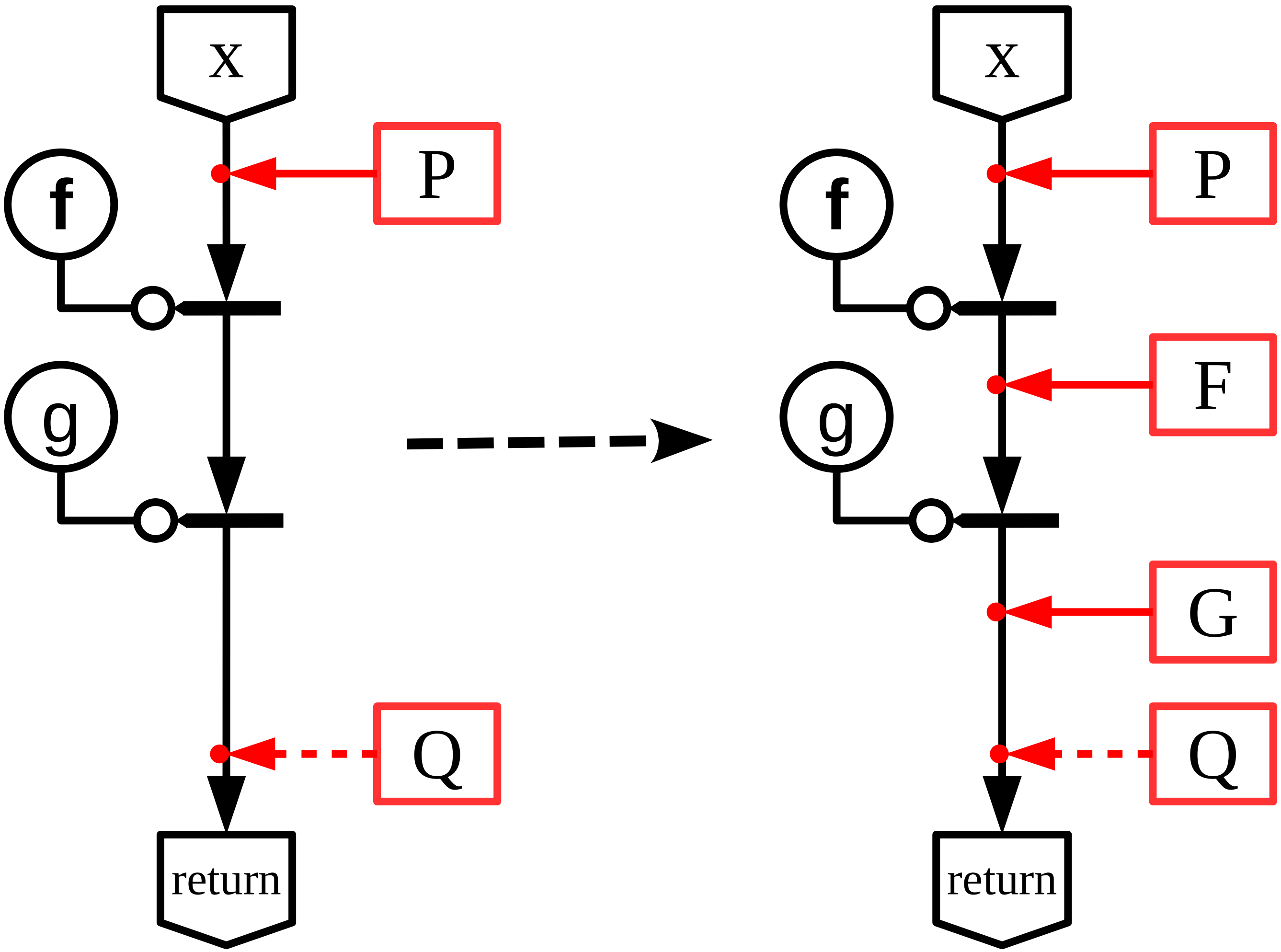
```
Func << funcdef x
{
      x:f:g >> return
}
```

Func — function name;
x — input argument;
f, **g** — functions applied to the argument.



Labeled data flow graph (LDFG) is a data flow graph, whose edges are marked with formulas in the specification language

Transformations of Data Flow Graph



$$(P \wedge F \wedge G) \Rightarrow Q$$

The initial triple
(initial LDFG)

Fully marked
LDFG

Formula in the
specification
language

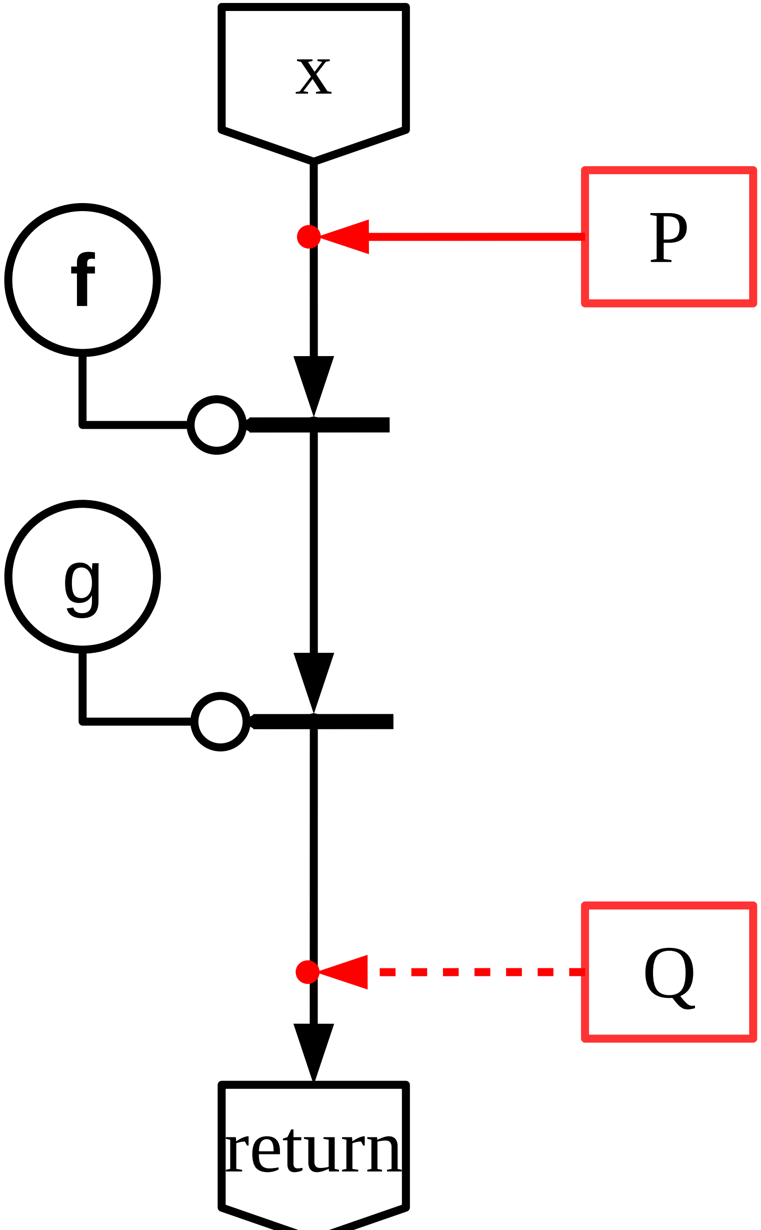
Types of Data Flow Graph Transformations

- **Edge marking** is the marking of graph edges with formulas in the specification language.
- **Folding of the program** is the reduction of the program code.
- **Modification of a data flow graph**:

  1) **equivalent transformation** is a transformation according to the rules of equivalent transformations for operators of the Pifagor language;

  2) **splitting** is LDFG splitting resulting in two or more LDFGs with modified graphs.

Marking Edges with Formulas



Theorems for f :

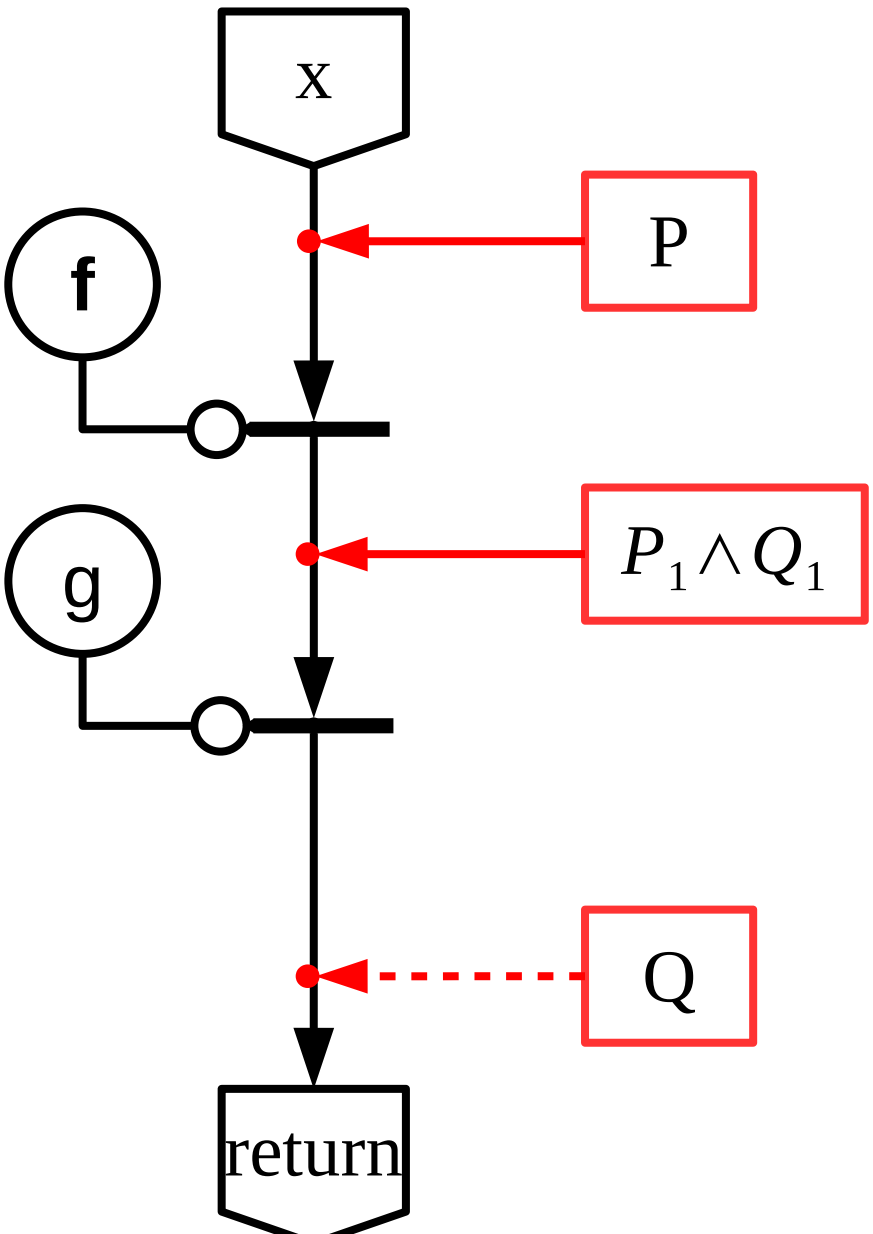
$\boxed{P_1}$ `x:f` $\rightarrow$ `r` $\boxed{Q_1}$

$\boxed{P_2}$ `x:f` $\rightarrow$ `r` $\boxed{Q_2}$

$\boxed{P_3}$ `x:f` $\rightarrow$ `r` $\boxed{Q_3}$

Each theorem corresponds to one way of the function **f** execution

Marking Edges with Formulas



Theorems for f :

$\boxed{P_1}$ x:f → r $\boxed{Q_1}$

~~$\boxed{P_2}$ x:f → r $\boxed{Q_2}$~~

~~$\boxed{P_3}$ x:f → r $\boxed{Q_3}$~~

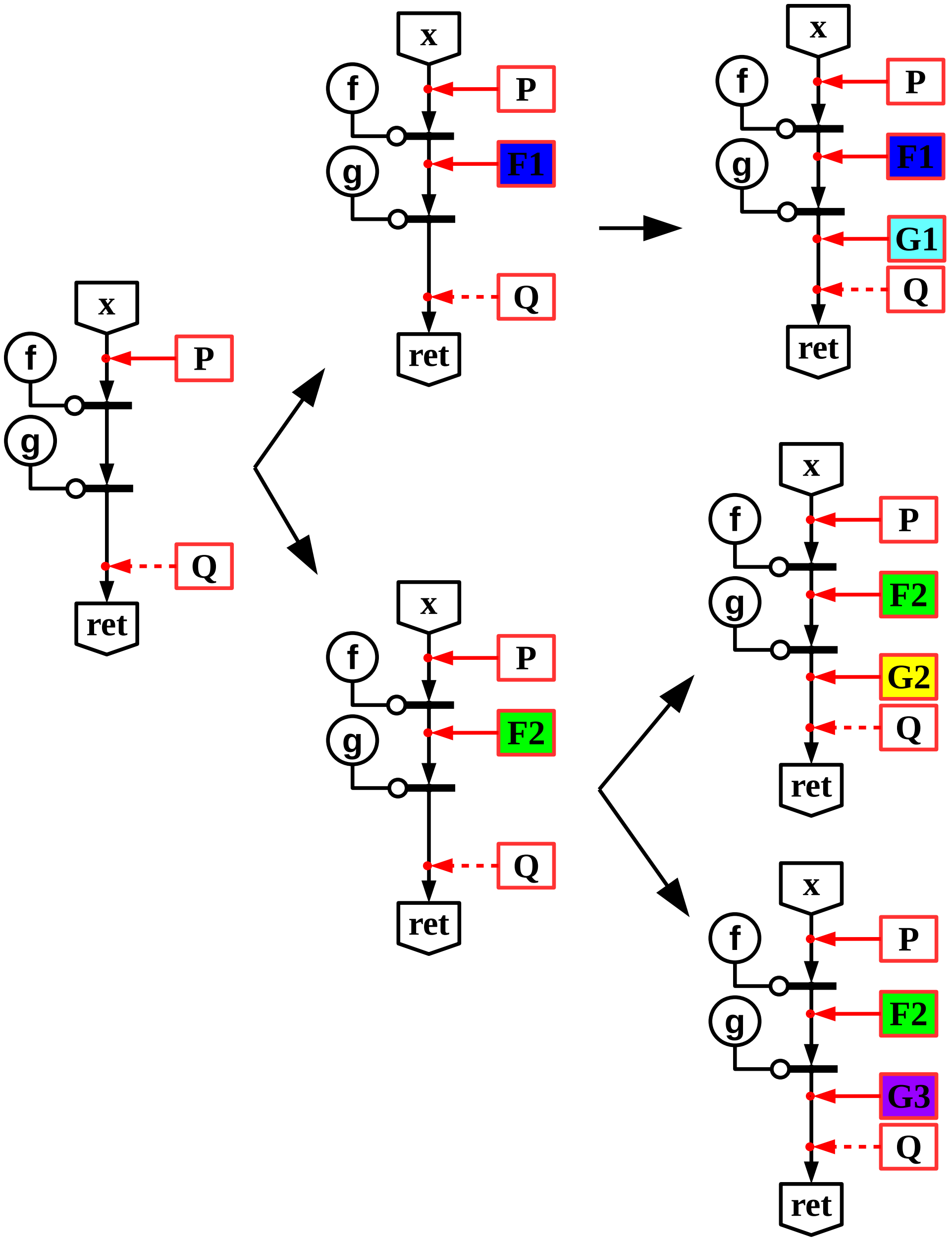
Check the satisfiability:

$P \Rightarrow P_1$ - true

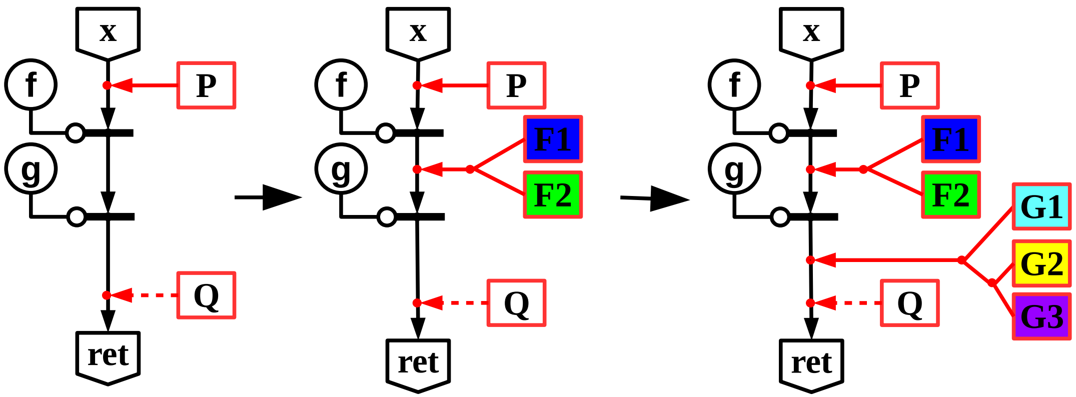
$P \Rightarrow P_2$ **- false**

$P \Rightarrow P_3$ **- false**

Marking Edges with Formulas

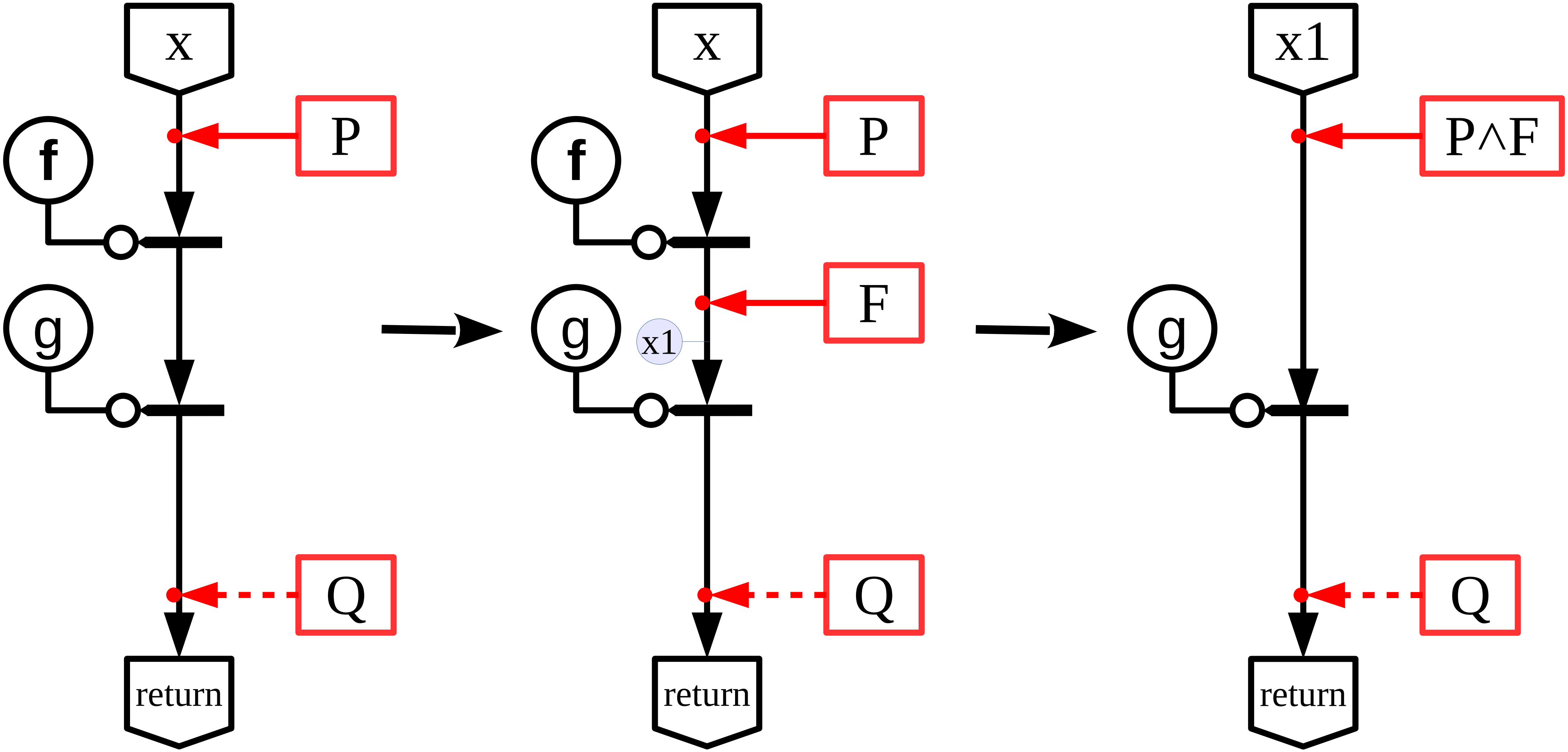


Compact representation of several DFGs



- several LDFGs with the same graphs are represented as a one LDFG,
- edges are marked with several formulas.

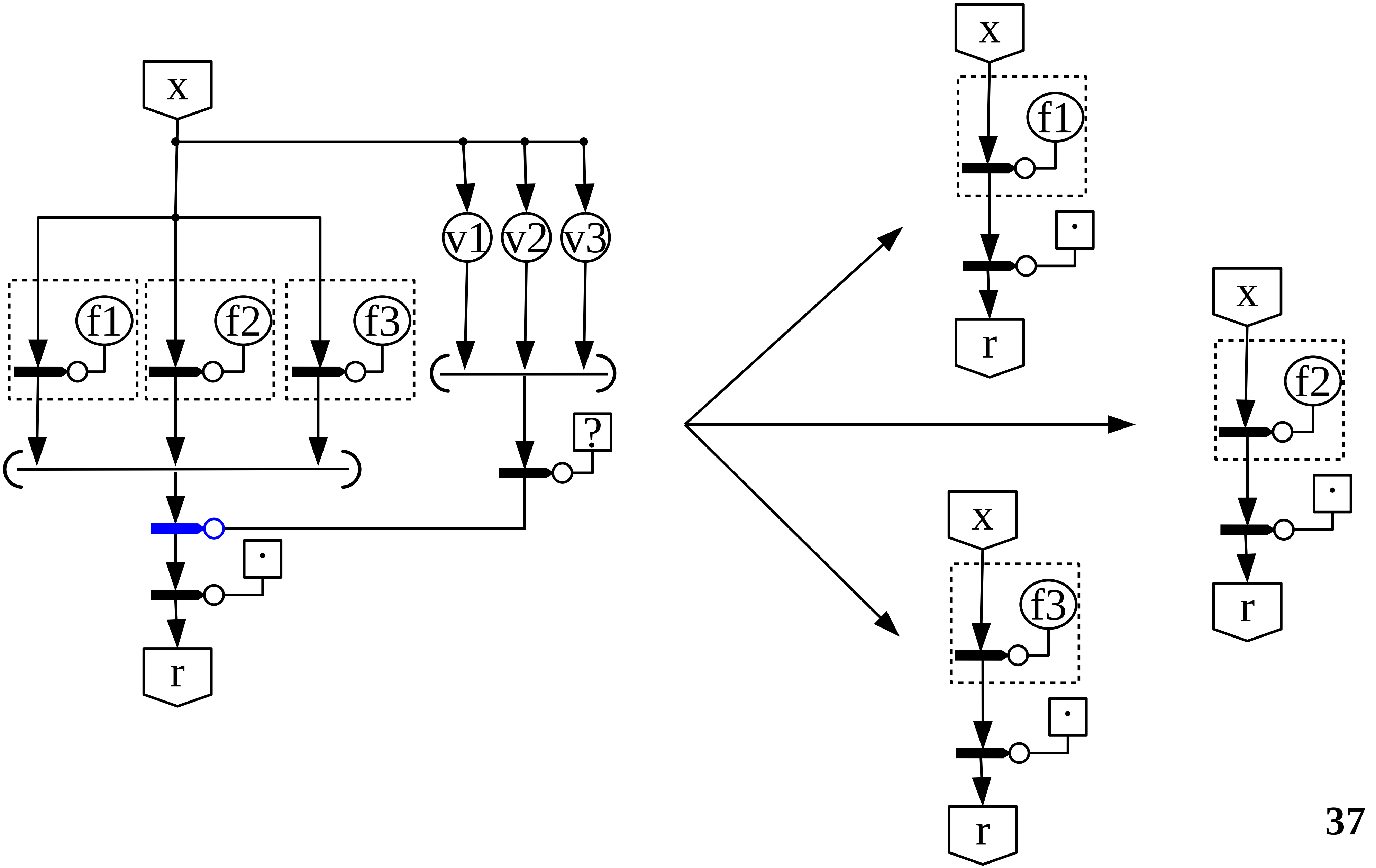
Folding of the Program



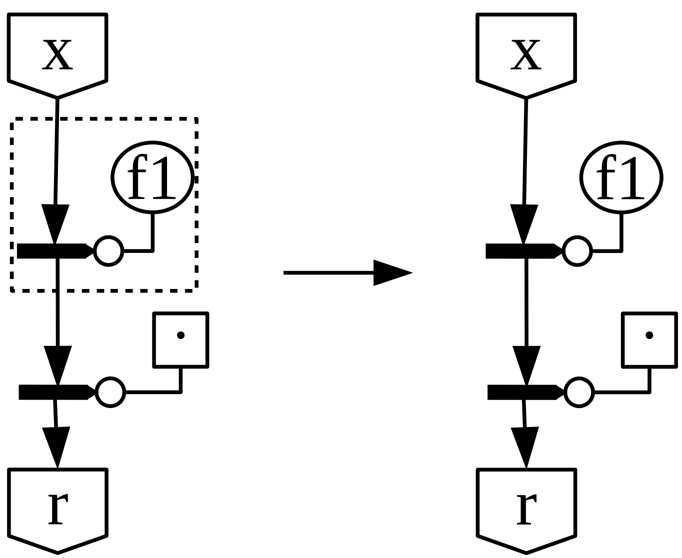
$\boxed{P}$ `x:f:g` → `r` $\boxed{Q}$

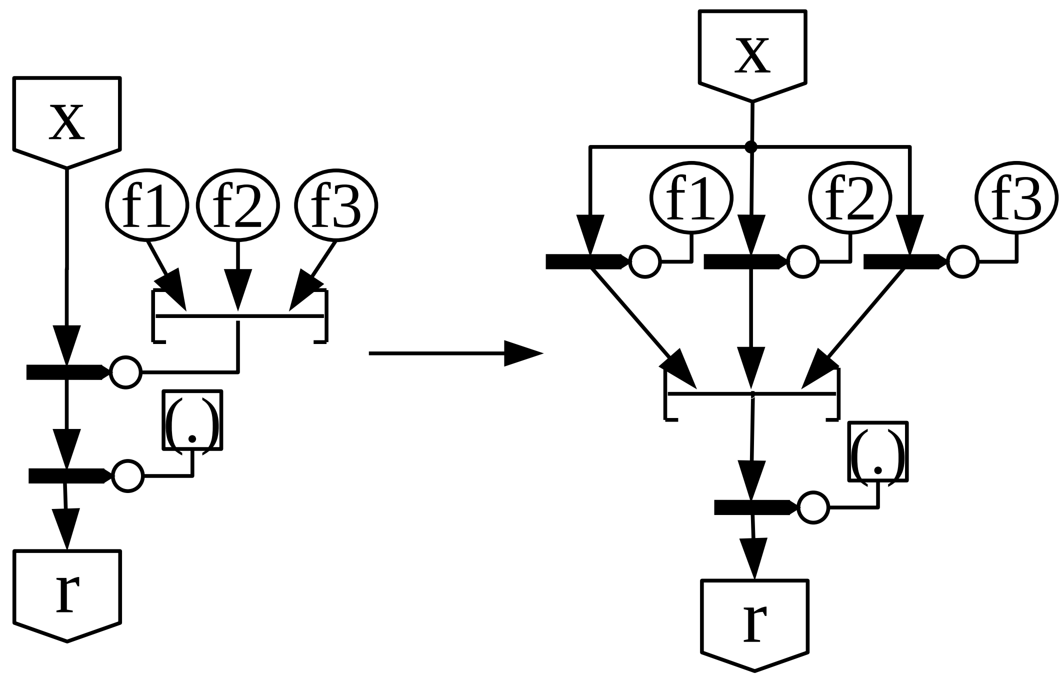
$\boxed{P \wedge F}$ `x1:g` → `r` $\boxed{Q}$

Modification of a Data-Flow Graph by Splitting



Modification of a Data-Flow Graph by Equivalent Transformation





Delay list release when coming to the operator of interpretation:

$\{X\}:F \rightarrow [X]:F$

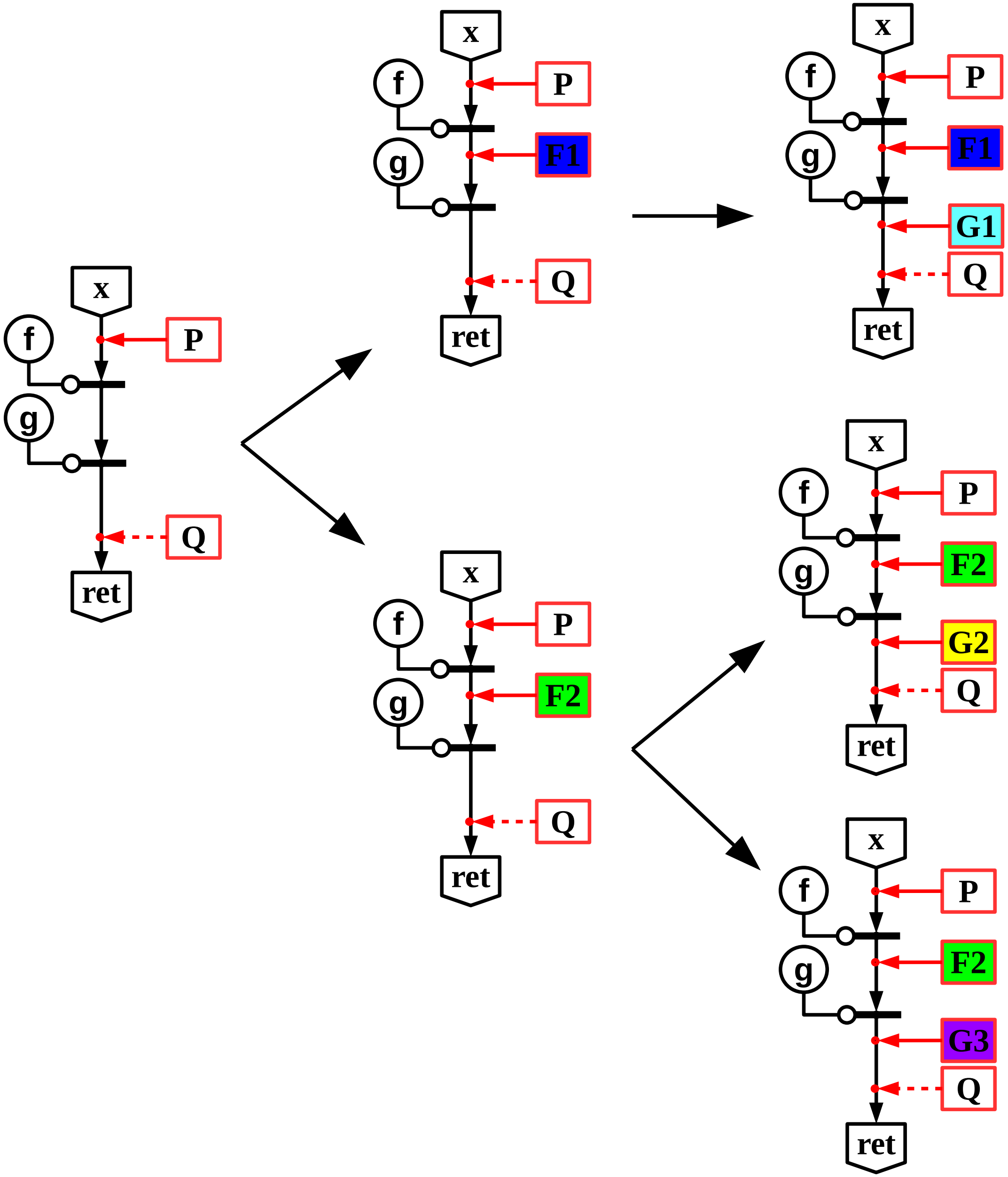
$X:\{F\} \rightarrow X:[F]$

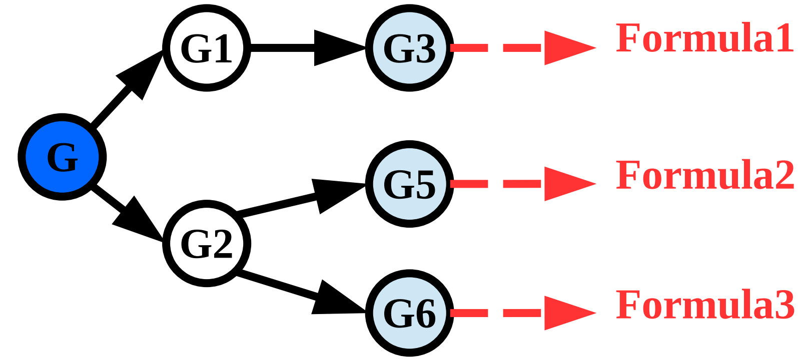
Parallel list release:

$[x_1,x_2, \ldots, x_n]:F \rightarrow [x_1:F, x_2:F, \ldots\ x_n:F]$

$X:[f_1, f_2, \ldots, f_n] \rightarrow [X:f_1, X:f_2, \ldots, X:f_n]$

Proof Tree



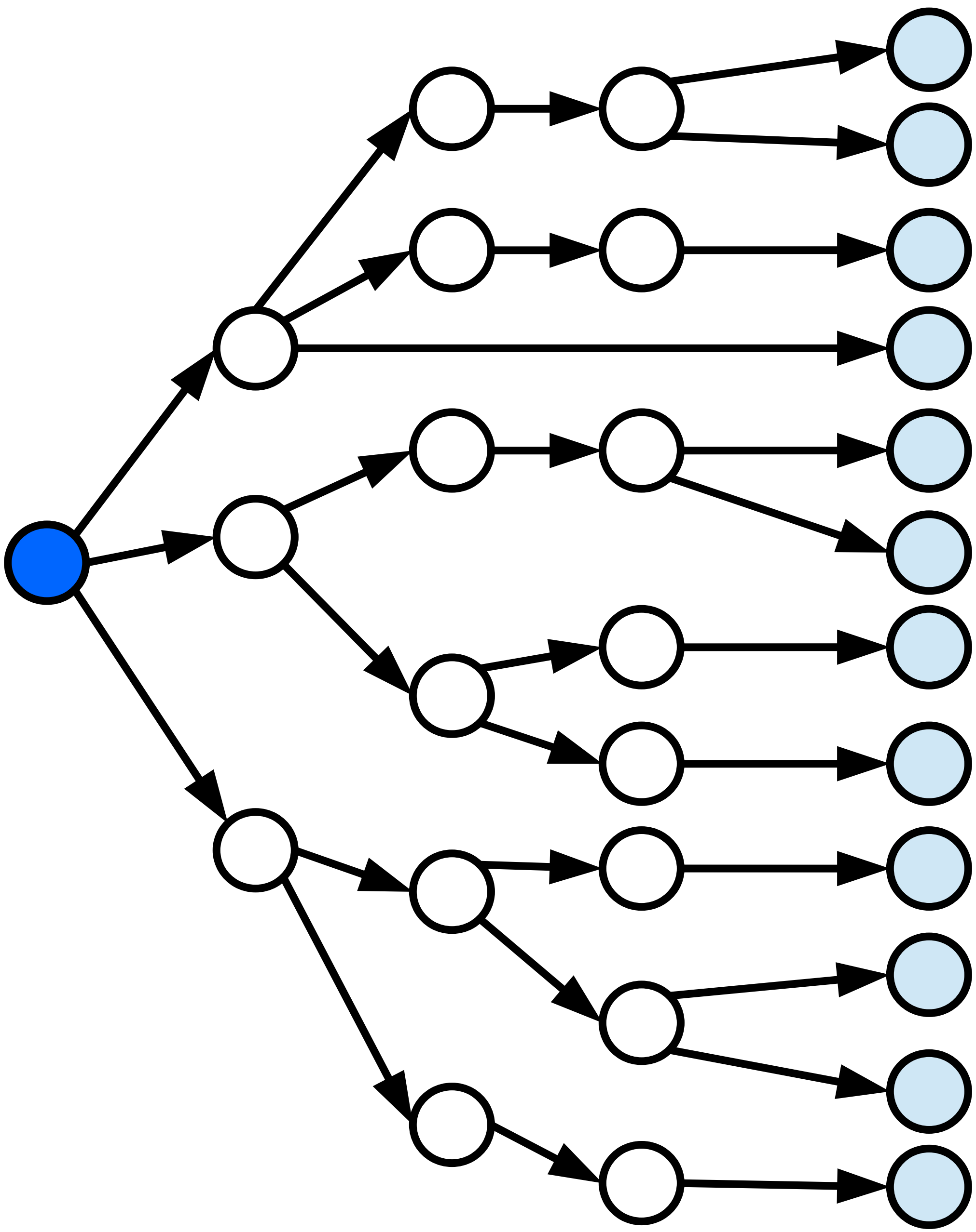


— initial triple

— partially marked graph

— totally marked graph

Usage of Satisfiability Condition for Axioms and Theorems

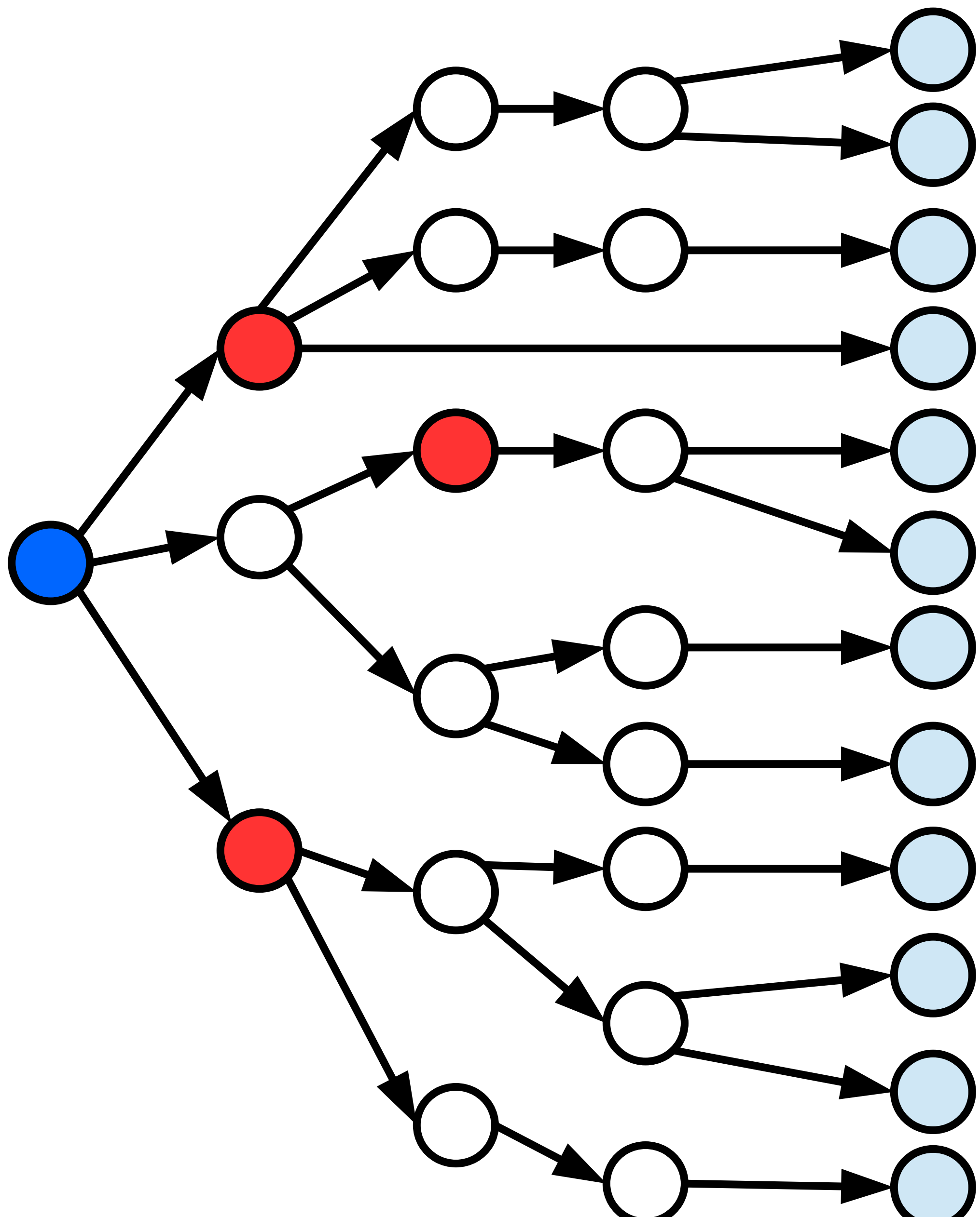


— initial triple

— partially marked graph

— totally marked graph

Usage of Satisfiability Condition for Axioms and Theorems



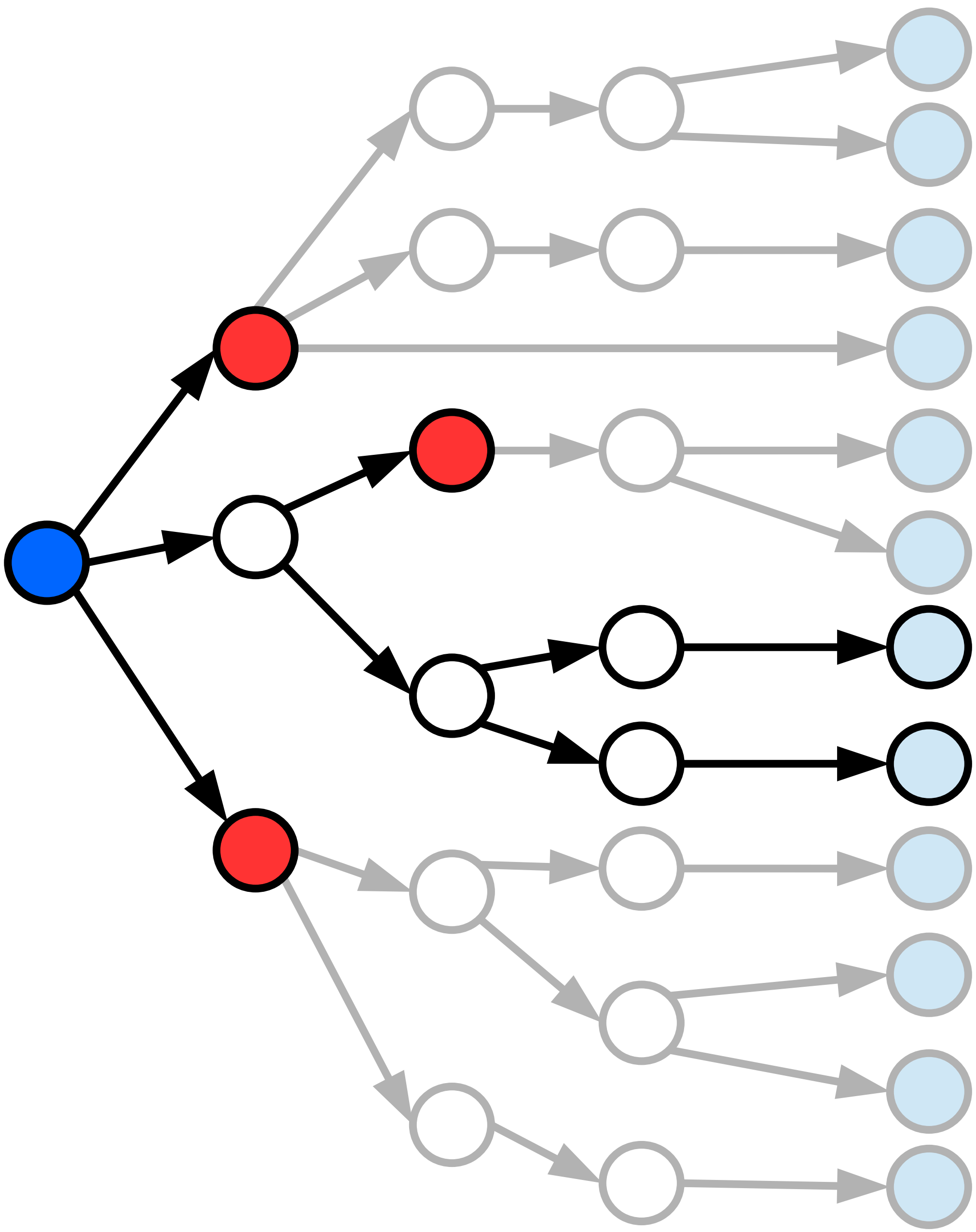
— initial triple

— partially marked graph

— totally marked graph

— ways of the program execution for which the satisfability condition is violated

Usage of Satisfiability Condition for Axioms and Theorems



— initial triple

— partially marked graph

— totally marked graph

— ways of the program execution for which the satisfability condition is violated

Proving the Recursive Function Correctness

The proof of recursive function **f(x){ ...f(t)… }** correctness is divided into two stages:

- proof of partial correctness;
- proof of program termination.

$$\boxed{P(x)}\ \texttt{f(x)} \to \texttt{r}\ \boxed{\ Q\ }$$

Proving of partial correctness

It is based on the principle of induction.

- The program is supposed to terminate.
- The basis of induction is the proof of the correctness of all trivial branches of recursion.
- The inductive assumption is the correctness of the proved triple for all recursive arguments:

$$\boxed{P(t)}\ \texttt{f(t)} \to \texttt{r}\ \boxed{\ Q\ }$$

Proving of Program Termination

The termination of a function is entirely determined by the input arguments of the recursive function:

f(x){ ...f(t)… }

1. **S** is a well-founded set (any non-empty subset has a minimal element).

2. **φ** is a bound function, whose arguments are the same as argument of the recursive function f, and values are from S

An example for factorial: $n! = n \cdot (n-1)!$

$4! =$
$4 \cdot 3! =$
$4 \cdot 3 \cdot 2! =$
$4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1$

Modification of Triples to Prove Program Termination

Initial Hoare triple:

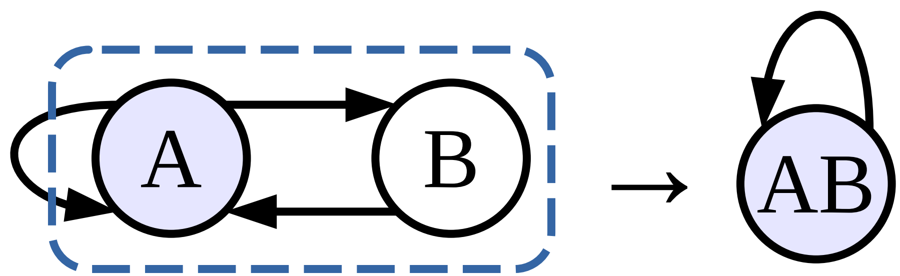
$\boxed{\text{Precondition}}$ f(x){....f(t)…} $\boxed{\text{Postcondition}}$

Modified Hoare triple:

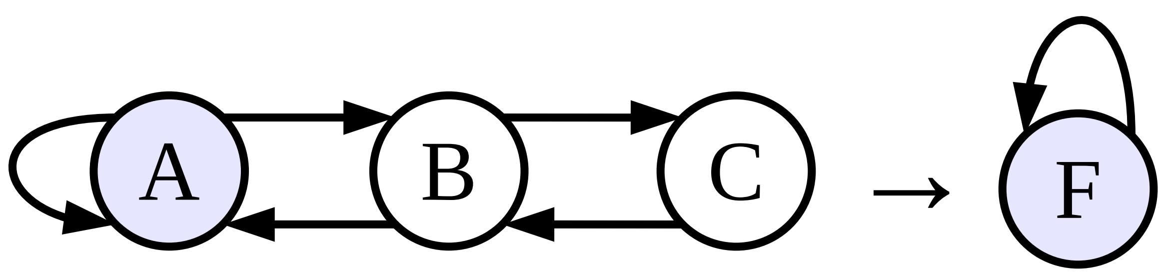
$\boxed{\text{Precondition}}$ f(x){....f(t)…} $\boxed{\begin{array}{c}\text{Postcondition}\\ \wedge \\ \varphi(x) > \varphi(t)\end{array}}$

Elimination of Mutual Recursion

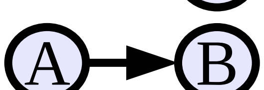
1. Code merging
   (simple case of mutual recursion)



2. Universal Recursive Function constructing

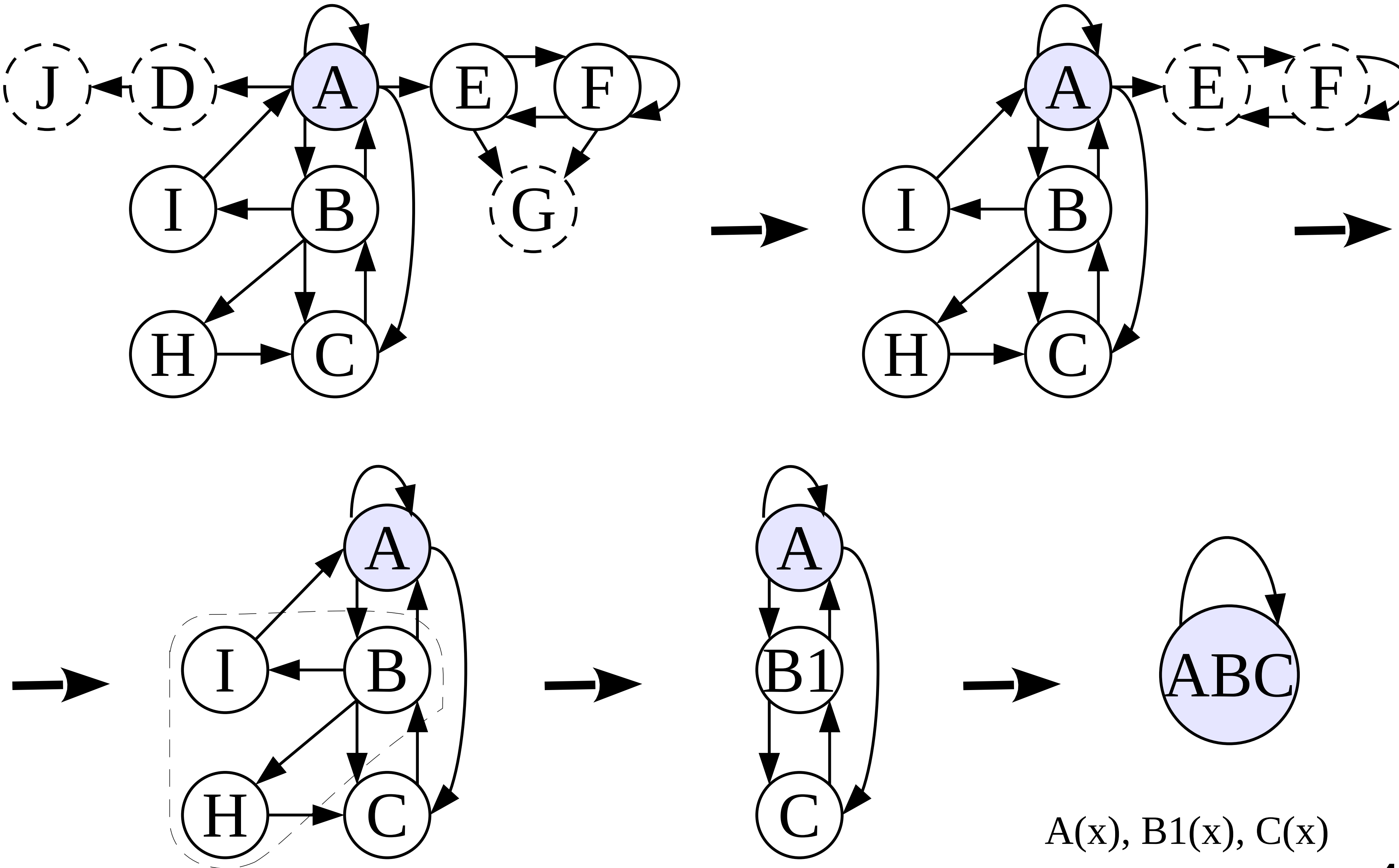


— schematic representation of a function A



— schematic representation of a function B call from a function A

An Arbitrary Recursive Function A
Transformation to the Direct Recursion

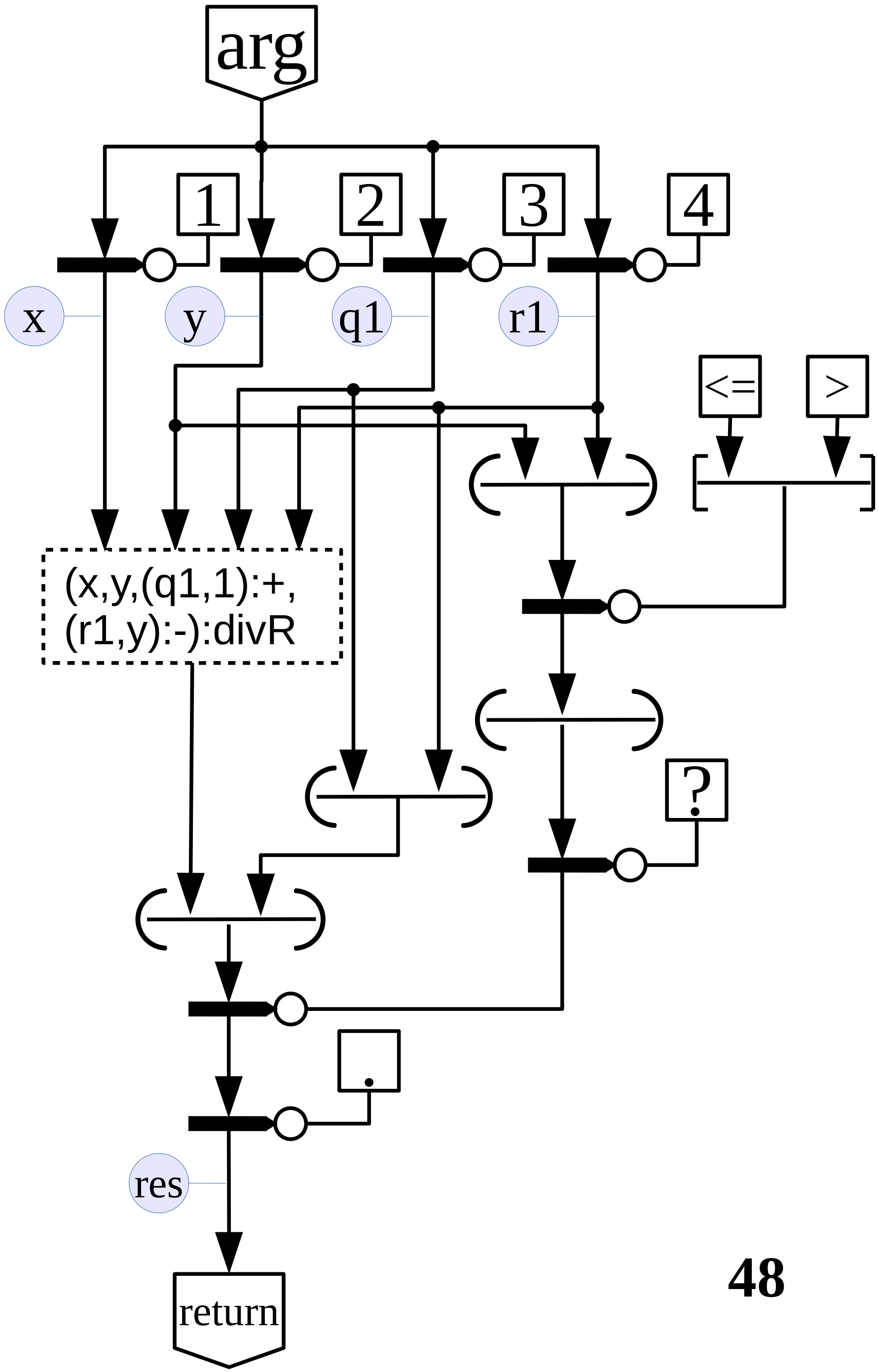


A(x), B1(x), C(x)

ABC(n,x), n=1,2,3

The Example of Function divR Verification

```
divR << funcdef arg {
  x<<arg:1; y<<arg:2; q1<<arg:3; r1<<arg:4;
  (
    {(x,y,(q1,1):+,(r1,y):-):divR},
    (q1,r1)
  ):[((y,r1):[<=, >]):?]:. >> return
}
```



arg — input argument;

arg = (x, y, q1, r1)

x = q1· y + r1

r1 is decreased by **y**

q1 is increased by **1**

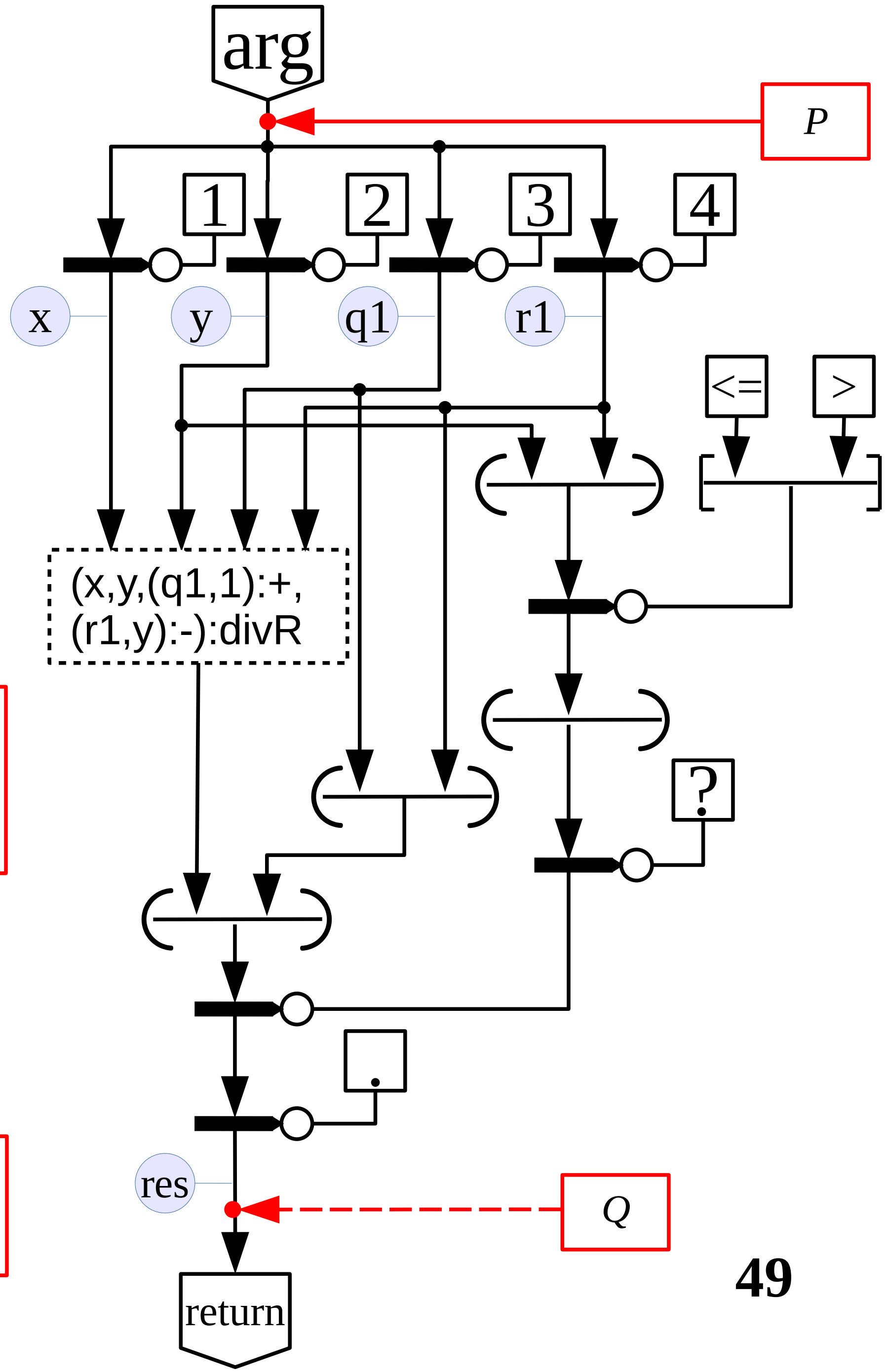
repeat while **r1>=y**

Auxiliary function DIV

```
DIV << funcdef arg {
  x<<arg:1;  y<<arg:2;
  (x,y,0,x):divR >> return
}
```

The Example of Function divR Verification

```
divR << funcdef arg {
  x<<arg:1; y<<arg:2; q1<<arg:3; r1<<arg:4;
  (
    {(x,y,(q1,1):+,(r1,y):-):divR},
    (q1,r1)
  ):[((y,r1):[<=, >]):?]:. >> return
}
```



The Hoare triple for divR:

$$arg=(x:\text{int},y:\text{int},\ q_1:\text{int},r_1:\text{int})\wedge (x\geqslant 0)\wedge(y>0)\wedge (q_1\geqslant 0)\wedge(r_1\geqslant 0)\wedge (x=y\cdot q_1+r_1)$$

arg:divR → res

$$res=(q:\text{int},r:\text{int})\wedge (q\geqslant 0)\wedge(r\geqslant 0)\wedge (x=y\cdot q+r)\wedge(r<y)$$

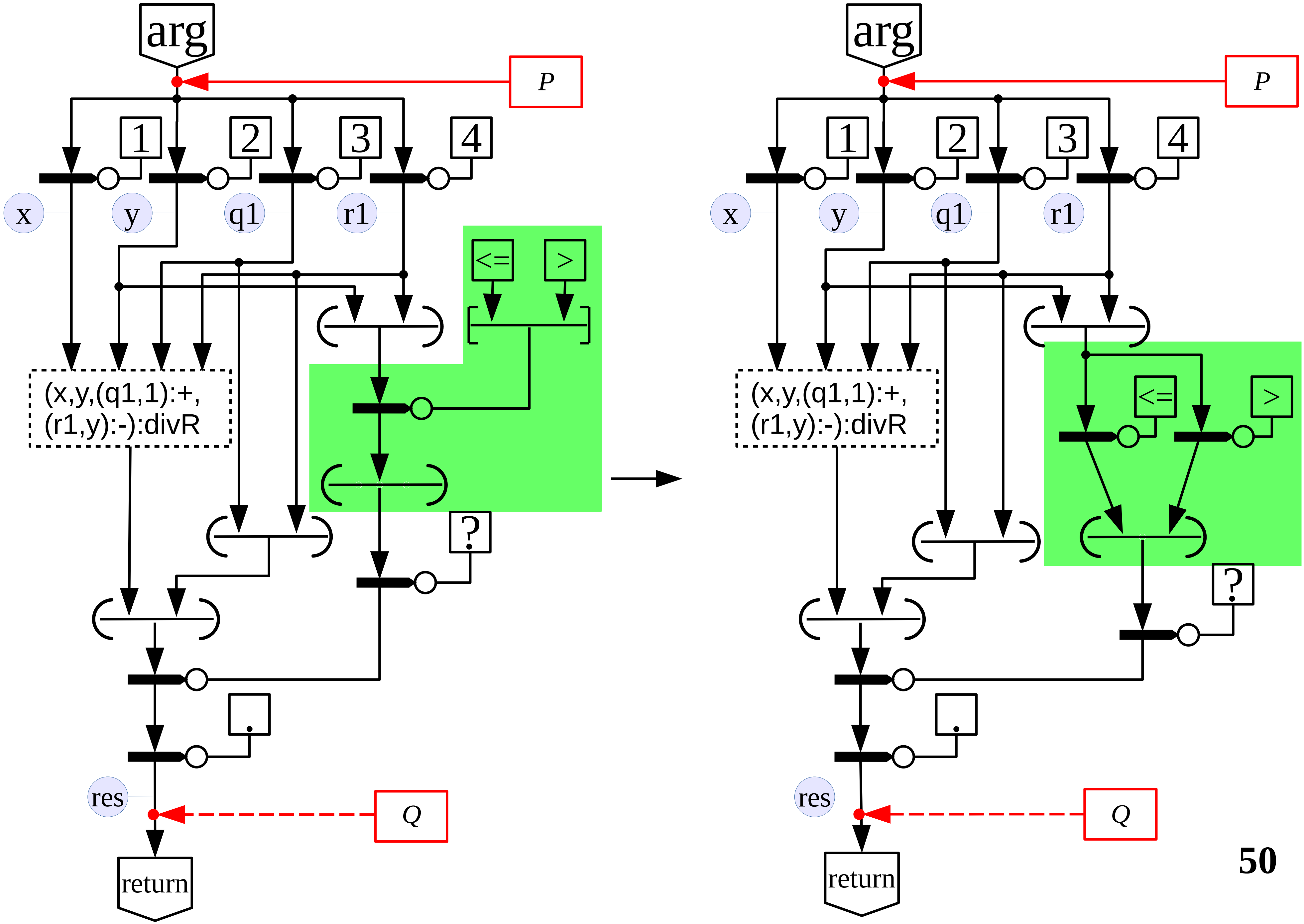
The Hoare triple for DIV:

$$arg=(x:\text{int},y:\text{int}) \ (x\geqslant 0)\wedge(y>0)$$

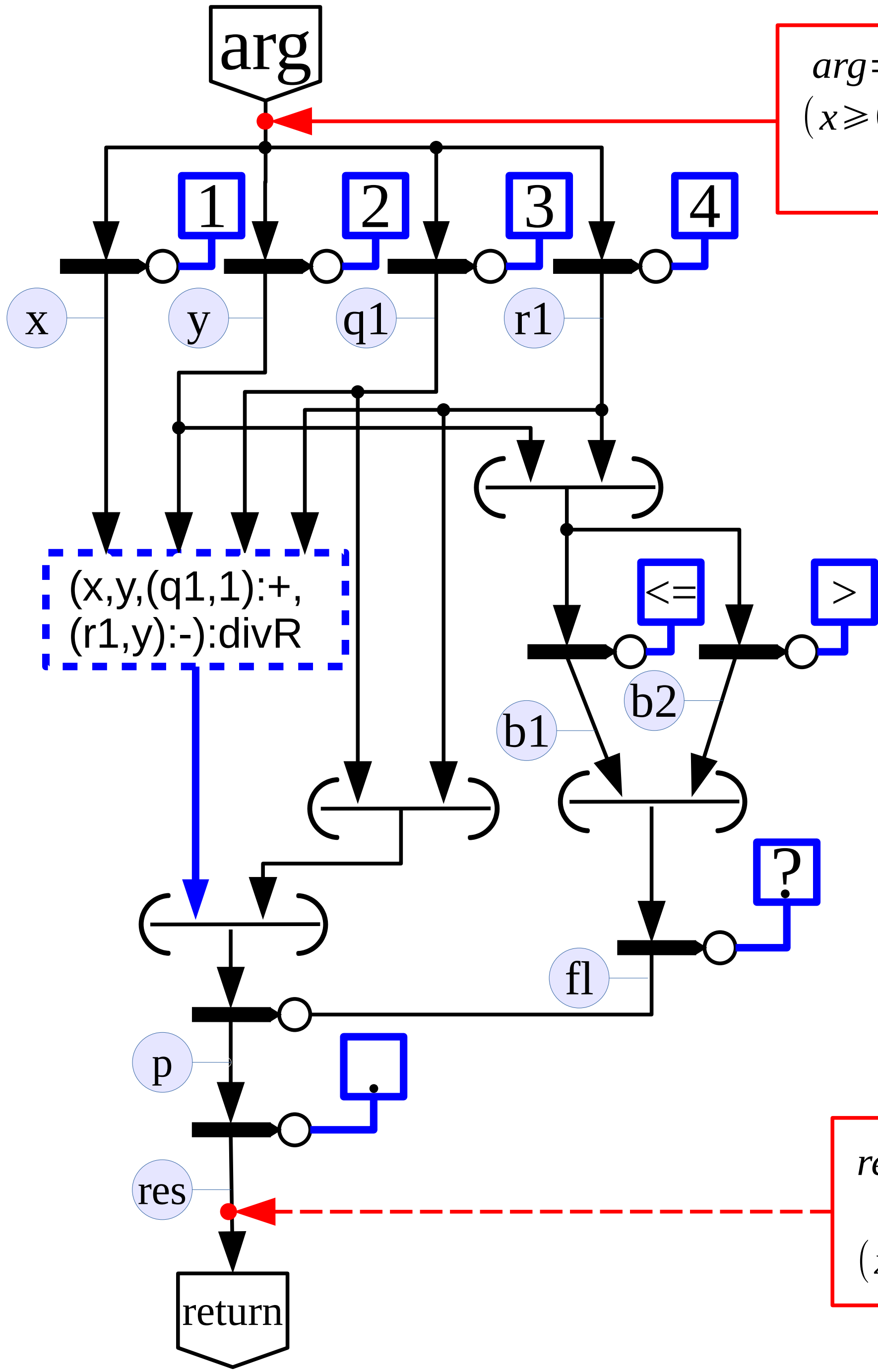
arg:DIV → res

$$res=(q:\text{int},r:\text{int})\wedge (x=y\cdot q+r)\wedge(r<y)$$

The Example of Function divR Verification



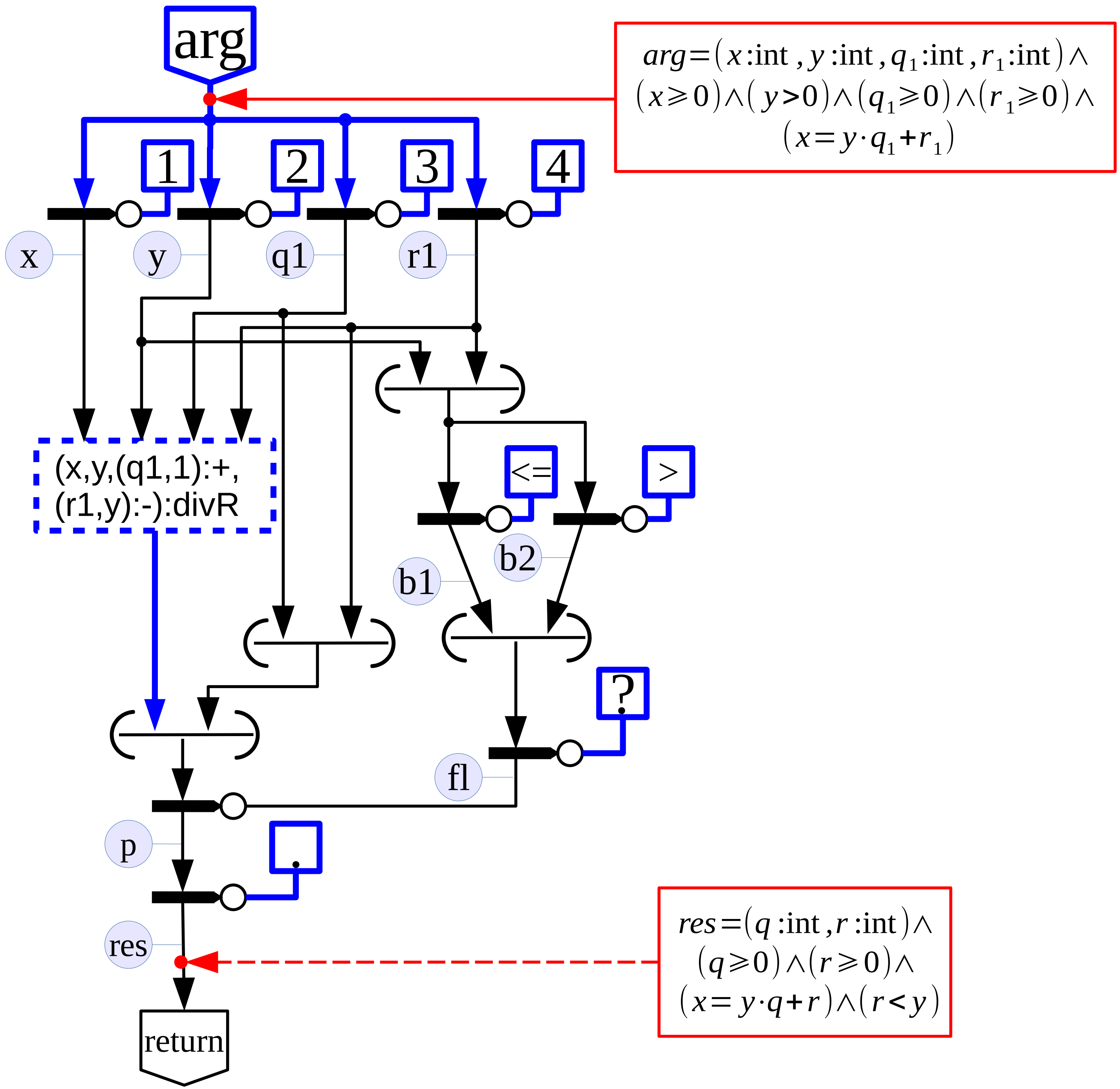
The Example of Function divR Verification



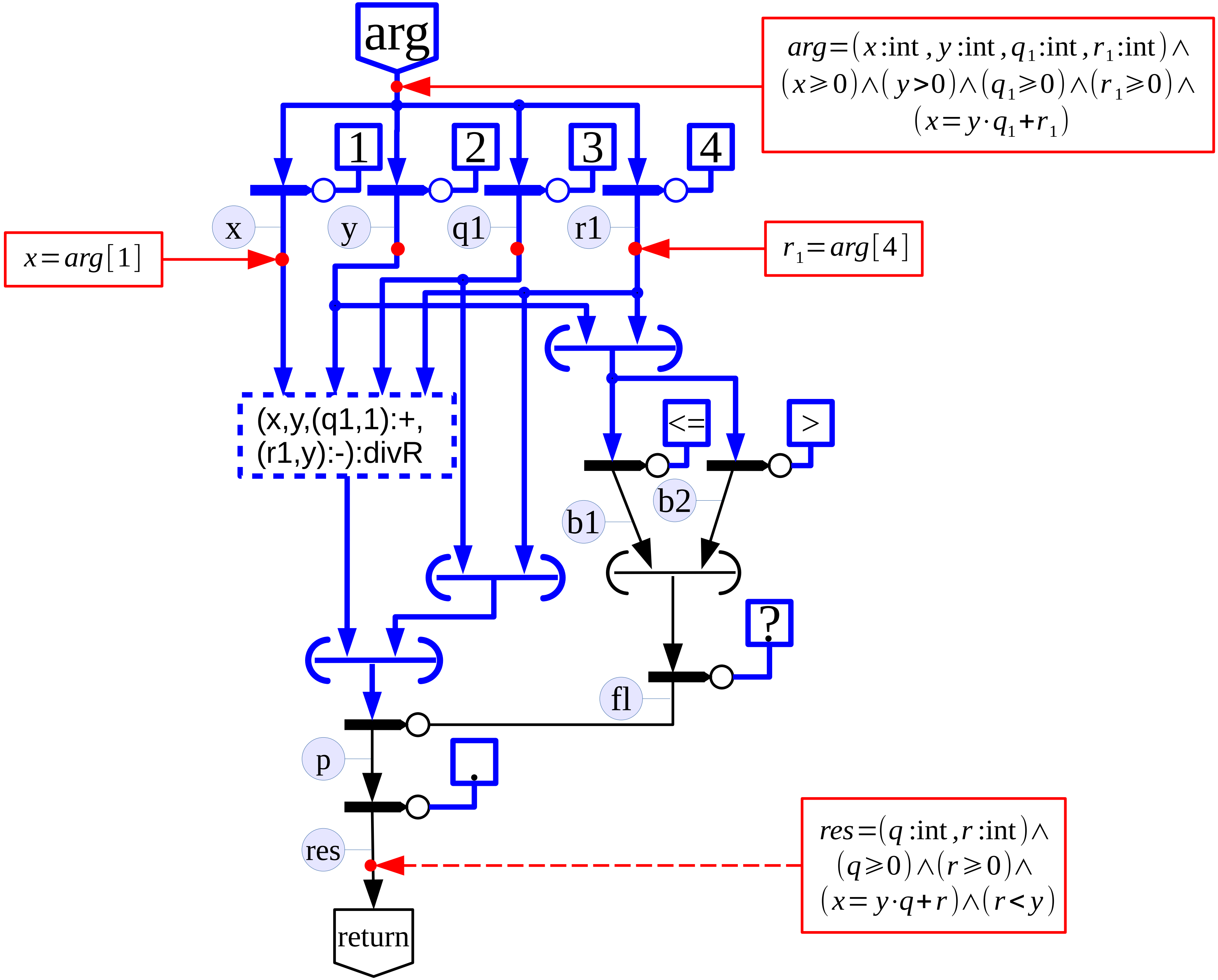
$arg=(x:\text{int}, y:\text{int}, q_1:\text{int}, r_1:\text{int})\wedge$
$(x\geqslant 0)\wedge(y>0)\wedge(q_1\geqslant 0)\wedge(r_1\geqslant 0)\wedge$
$(x=y\cdot q_1+r_1)$

$res=(q:\text{int}, r:\text{int})\wedge$
$(q\geqslant 0)\wedge(r\geqslant 0)\wedge$
$(x=y\cdot q+r)\wedge(r<y)$

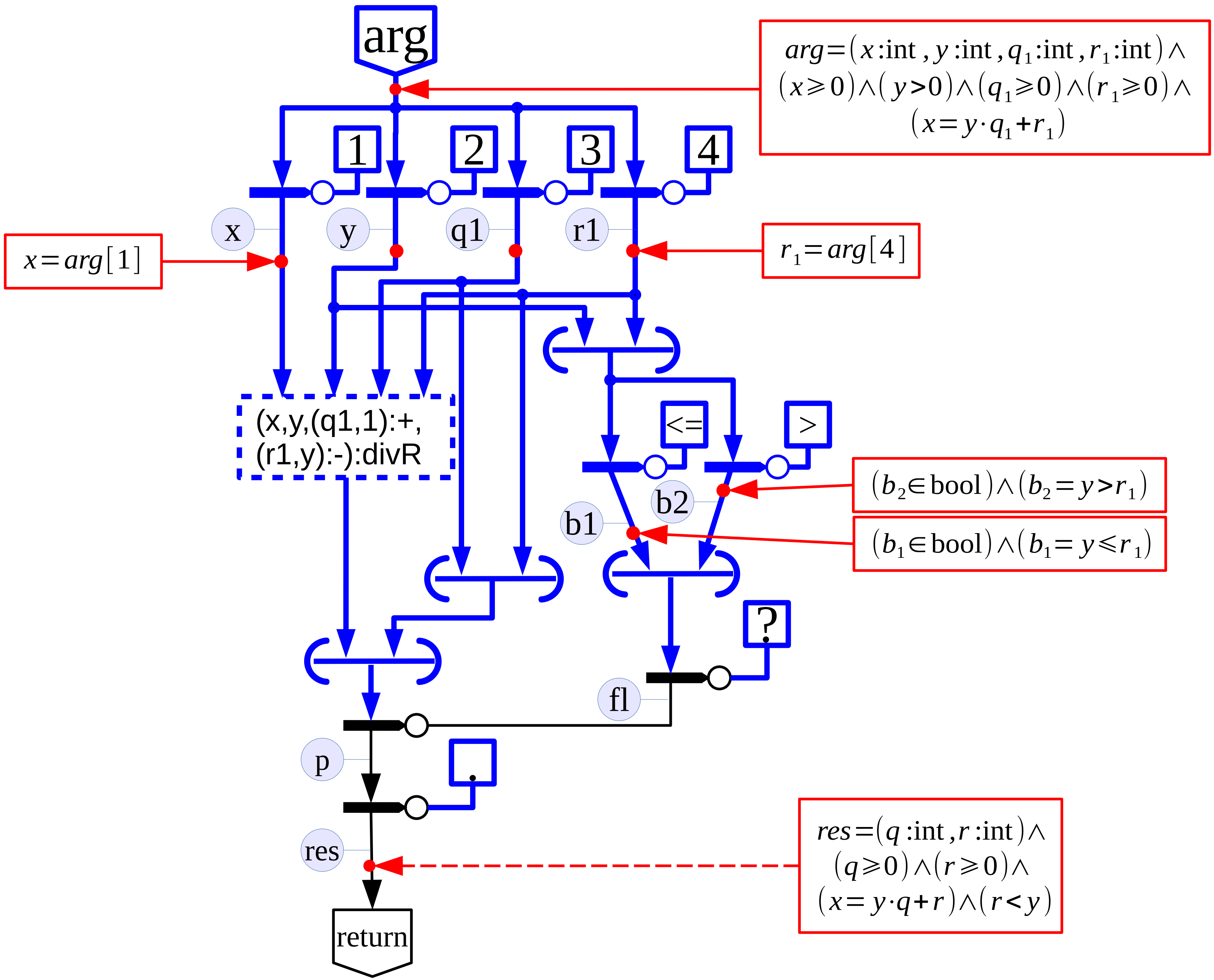
The Example of Function divR Verification



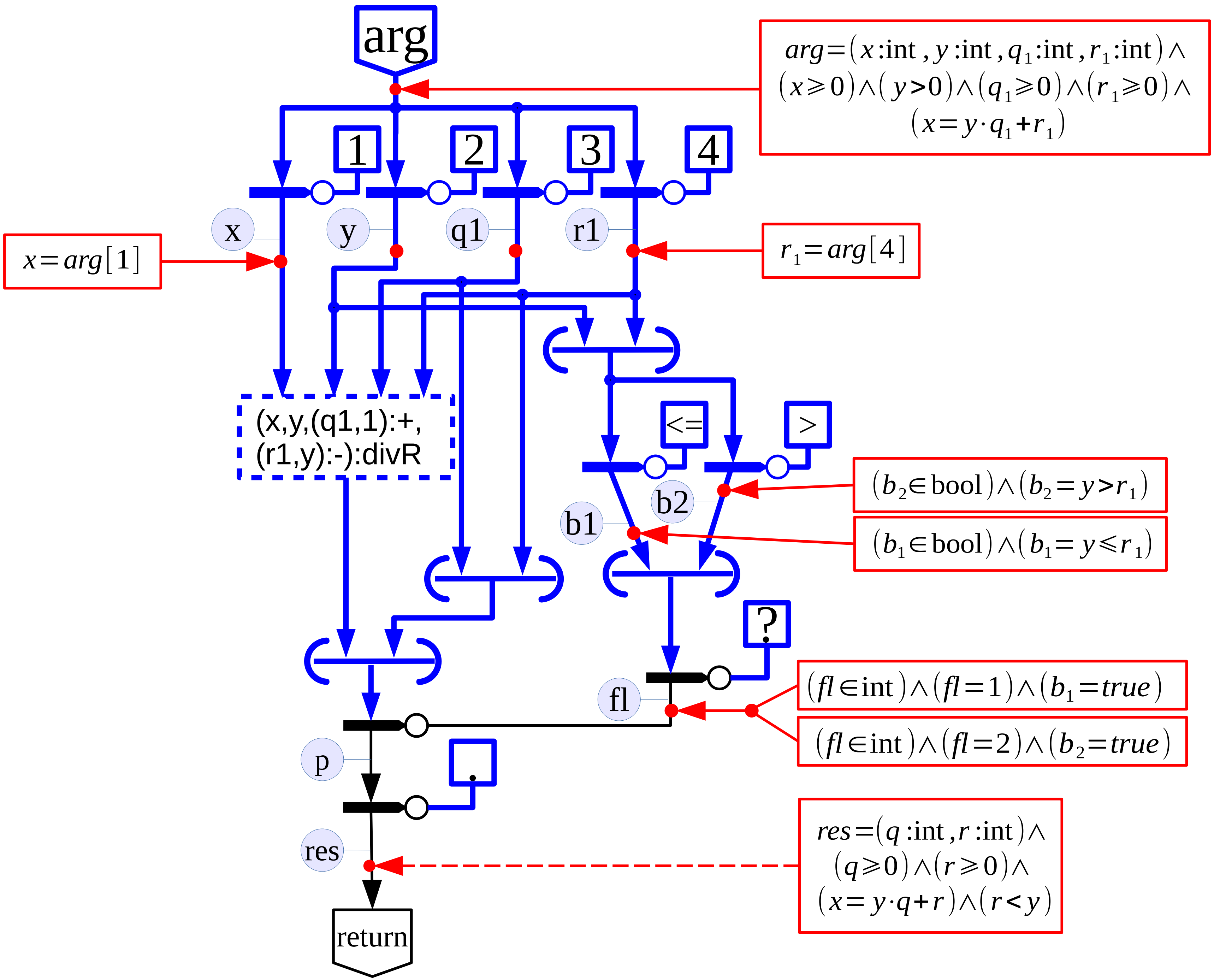
The Example of Function divR Verification



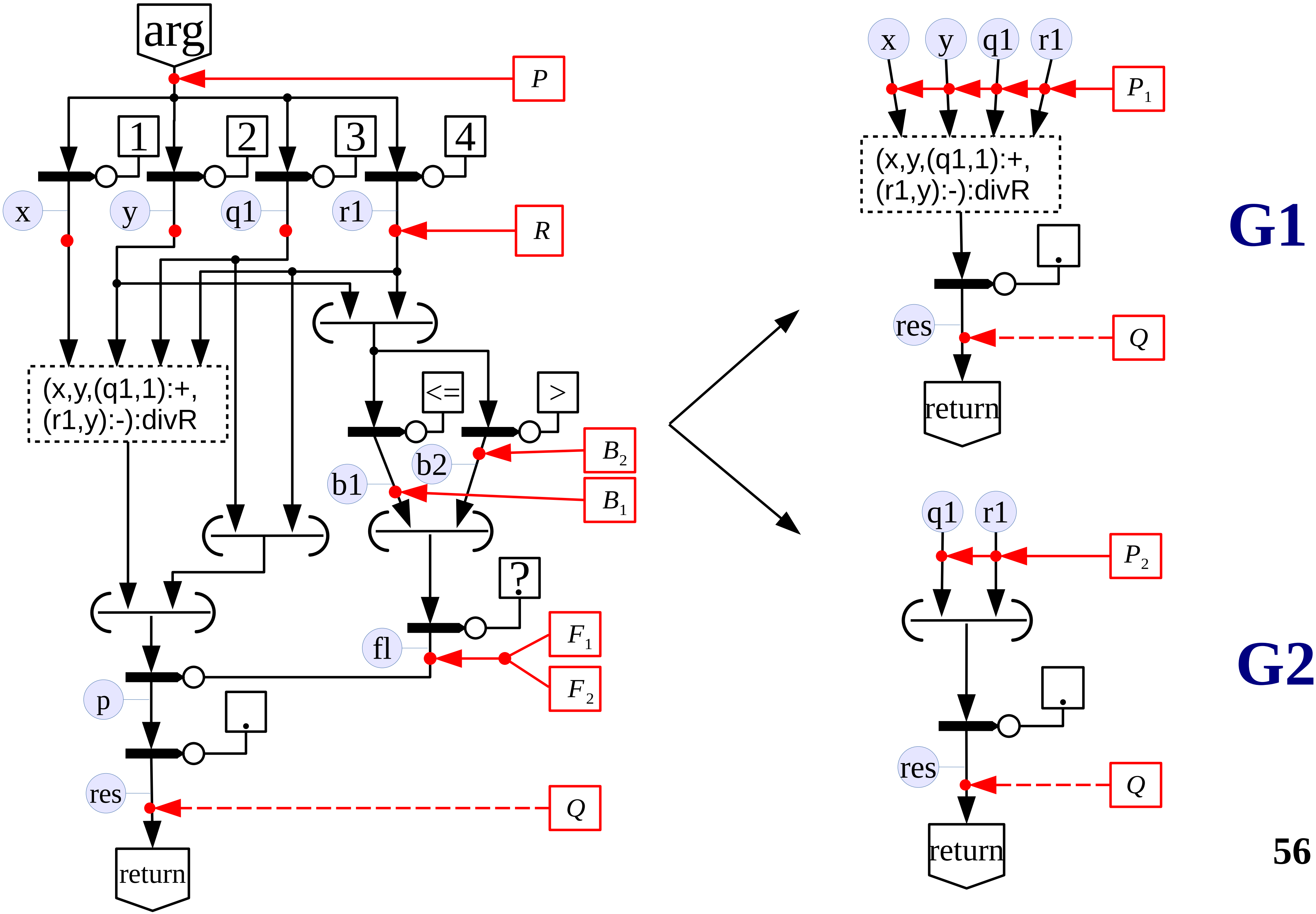
The Example of Function divR Verification



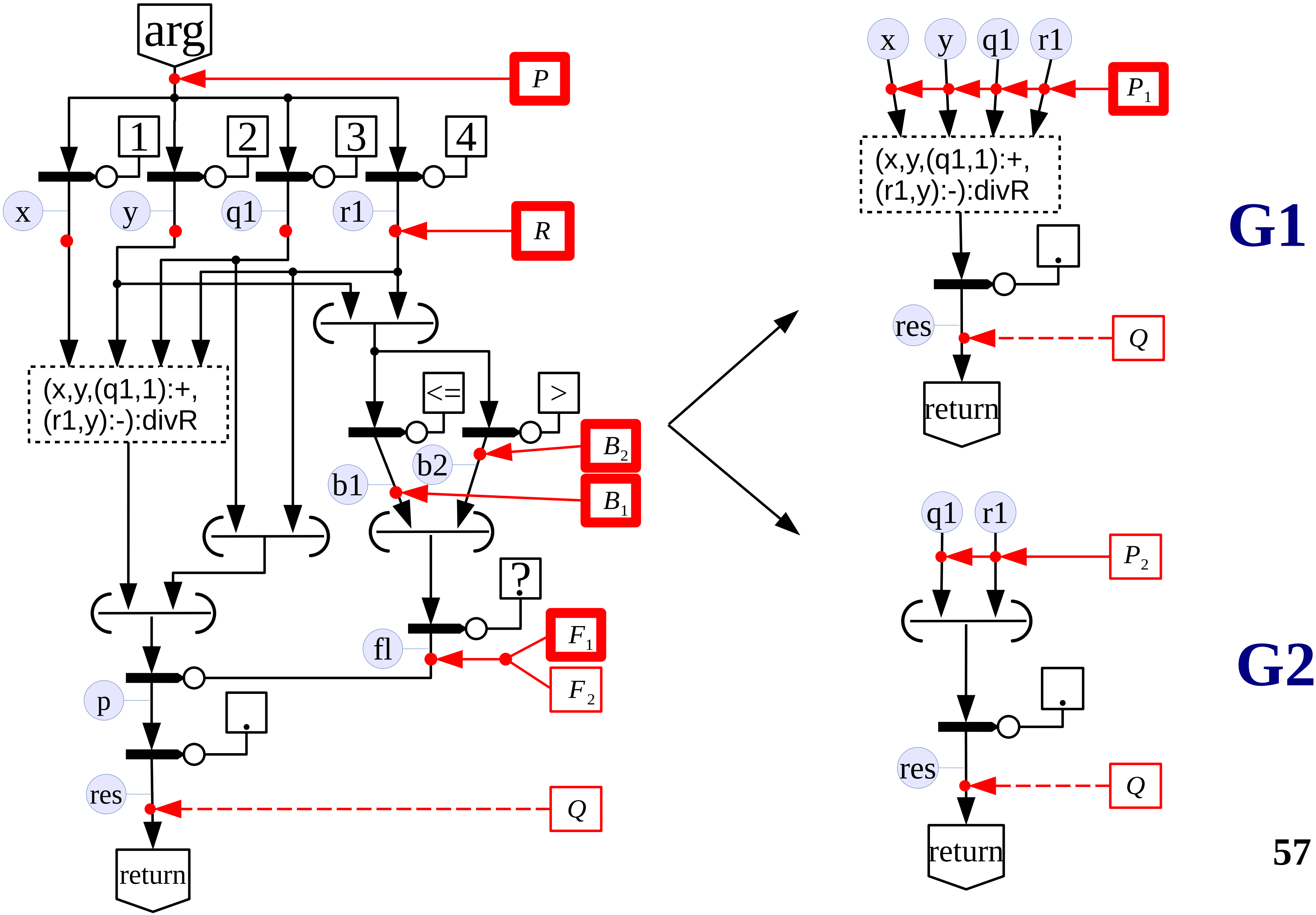
The Example of Function divR Verification



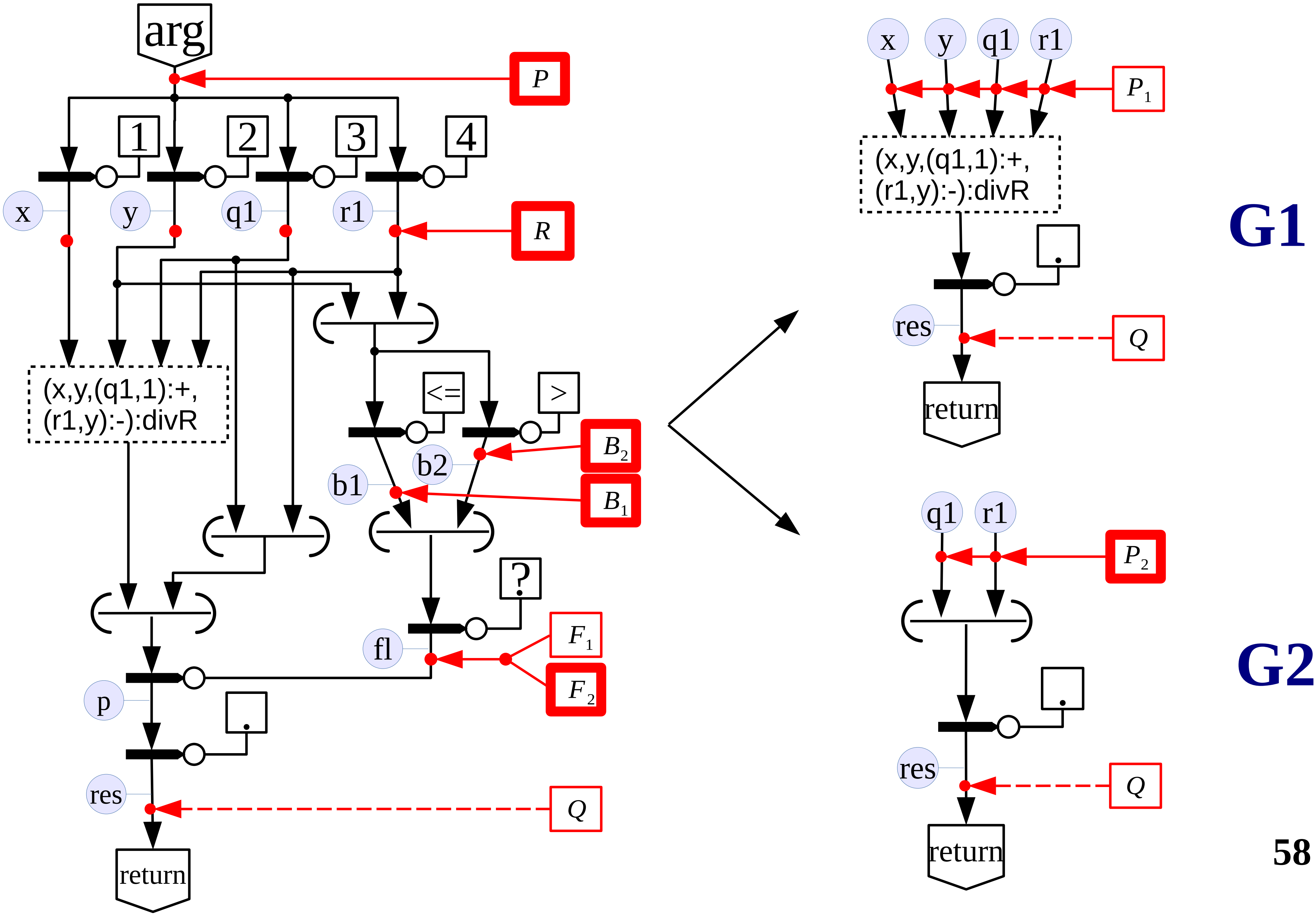
The Example of Function divR Verification



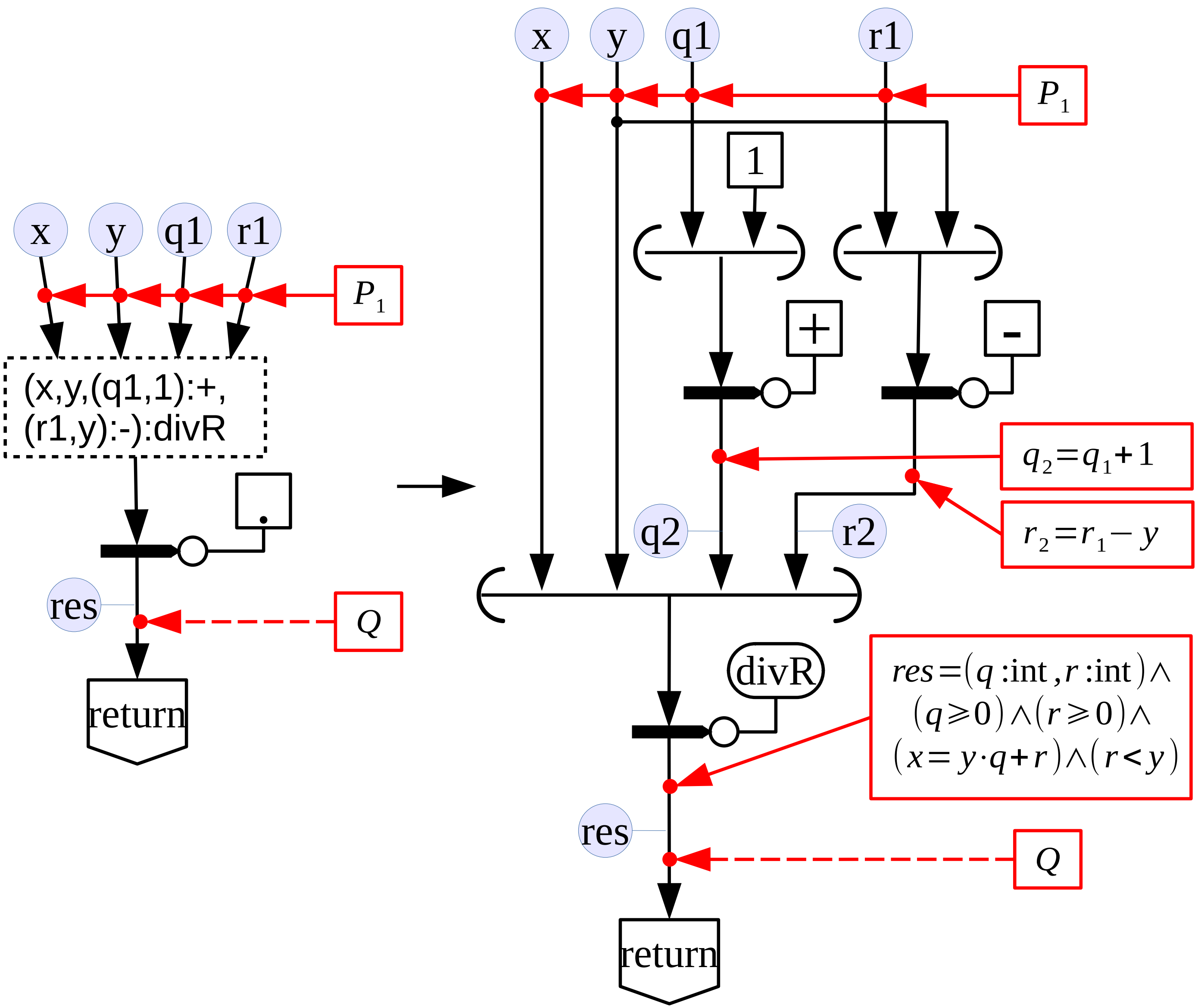
The Example of Function divR Verification



The Example of Function divR Verification



The Example of Function divR Verification



Induction Step:
Assume that all recursive calls are correct

(x,y,q_2,r_2) should satisfy the precondition of the function **`divR`**

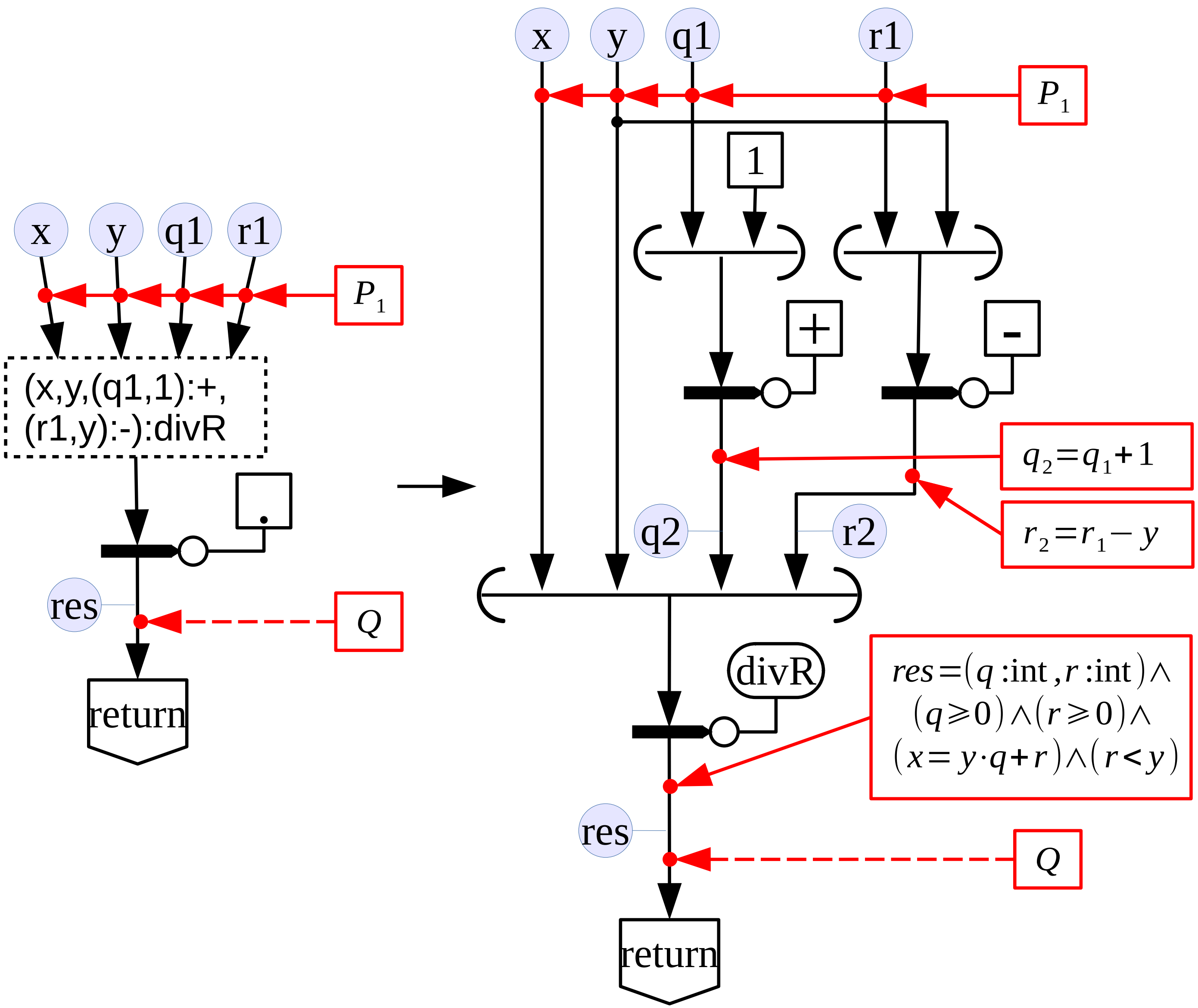
$$(x\geqslant 0)\wedge(y>0)\wedge(q_2\geqslant 0)\wedge(r_2\geqslant 0)\wedge(x=y\cdot q_2+r_2)$$

Then the output edge **res** is marked with postcondition of function **`divR`**

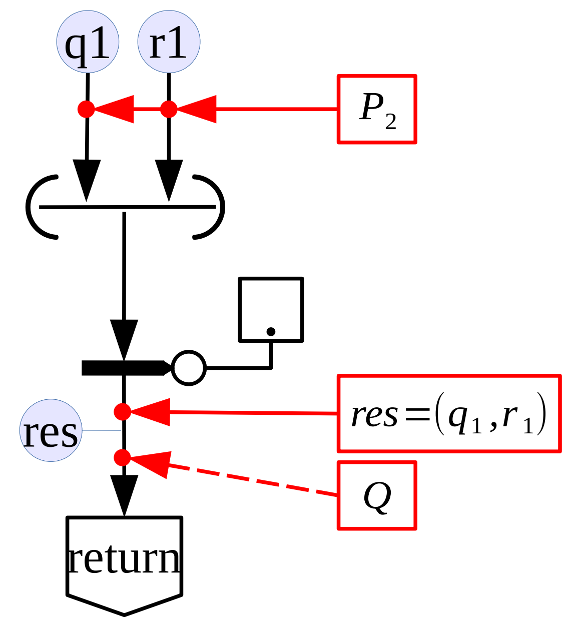
$$res=(q:\text{int},r:\text{int})\wedge(q\geqslant 0)\wedge(r\geqslant 0)\wedge(x=y\cdot q+r)\wedge(r<y)$$

G1

The Example of Function divR Verification



G1



G2

The Example of Function divR Verification

$$(x \geqslant 0) \wedge (y > 0) \wedge (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge$$
$$(x = y \cdot q_1 + r_1) \wedge (b_1 = y \leq r_1) \wedge (b_2 = y > r_1) \wedge$$
$$(fl = 1) \wedge (b_1 = true)$$

$$\wedge$$

$$q_2 = q_1 + 1$$

$$\wedge$$

$$r_2 = r_1 - y$$

$$\wedge$$

$$res = (q : \text{int}, r : \text{int}) \wedge$$
$$(q \geqslant 0) \wedge (r \geqslant 0) \wedge$$
$$(x = y \cdot q + r) \wedge (r < y)$$

$$\Rightarrow$$

$$res = (q : \text{int}, r : \text{int}) \wedge$$
$$(q \geqslant 0) \wedge (r \geqslant 0) \wedge$$
$$(x = y \cdot q + r) \wedge (r < y)$$

G1

$$(x \geqslant 0) \wedge (y > 0) \wedge (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge$$
$$(x = y \cdot q_1 + r_1) \wedge (b_1 = y \leq r_1) \wedge (b_2 = y > r_1) \wedge$$
$$(fl = 2) \wedge (b_2 = true)$$

$$\wedge$$

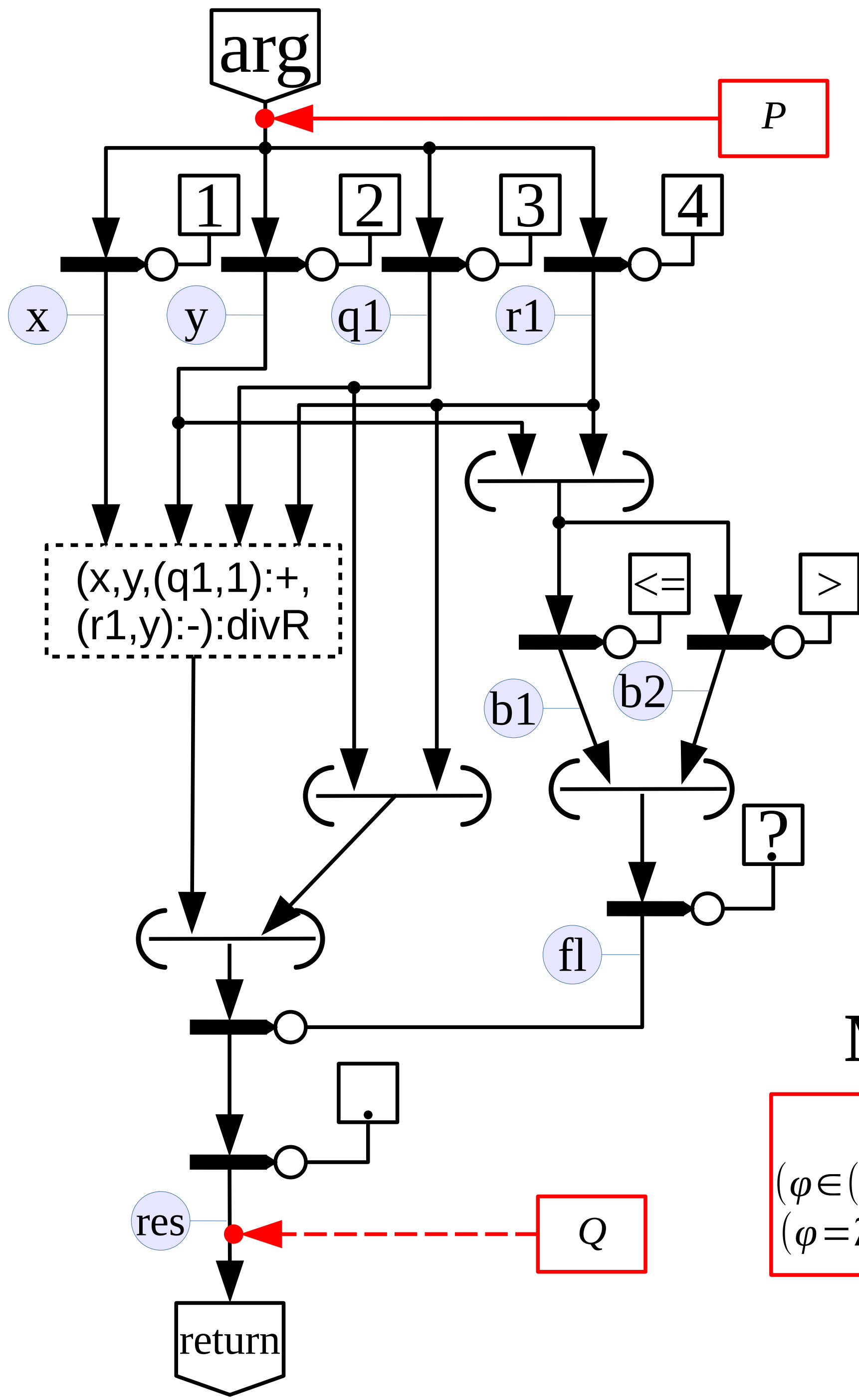
$$res = (q_1, r_1)$$

$$\Rightarrow$$

$$((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee$$
$$((r = 1) \wedge (x = 0))$$

G2

The Example of Function divR Verification



divR(x,y,q_1,r_1)

{ … divR(x,y,q_1+1,r_1-y) … }

1. S is the set of natural numbers.
2. Bound function

   $\varphi(x,y,q_1,r_1) = x - (y \cdot q_1)$
3. It is necessary to show that

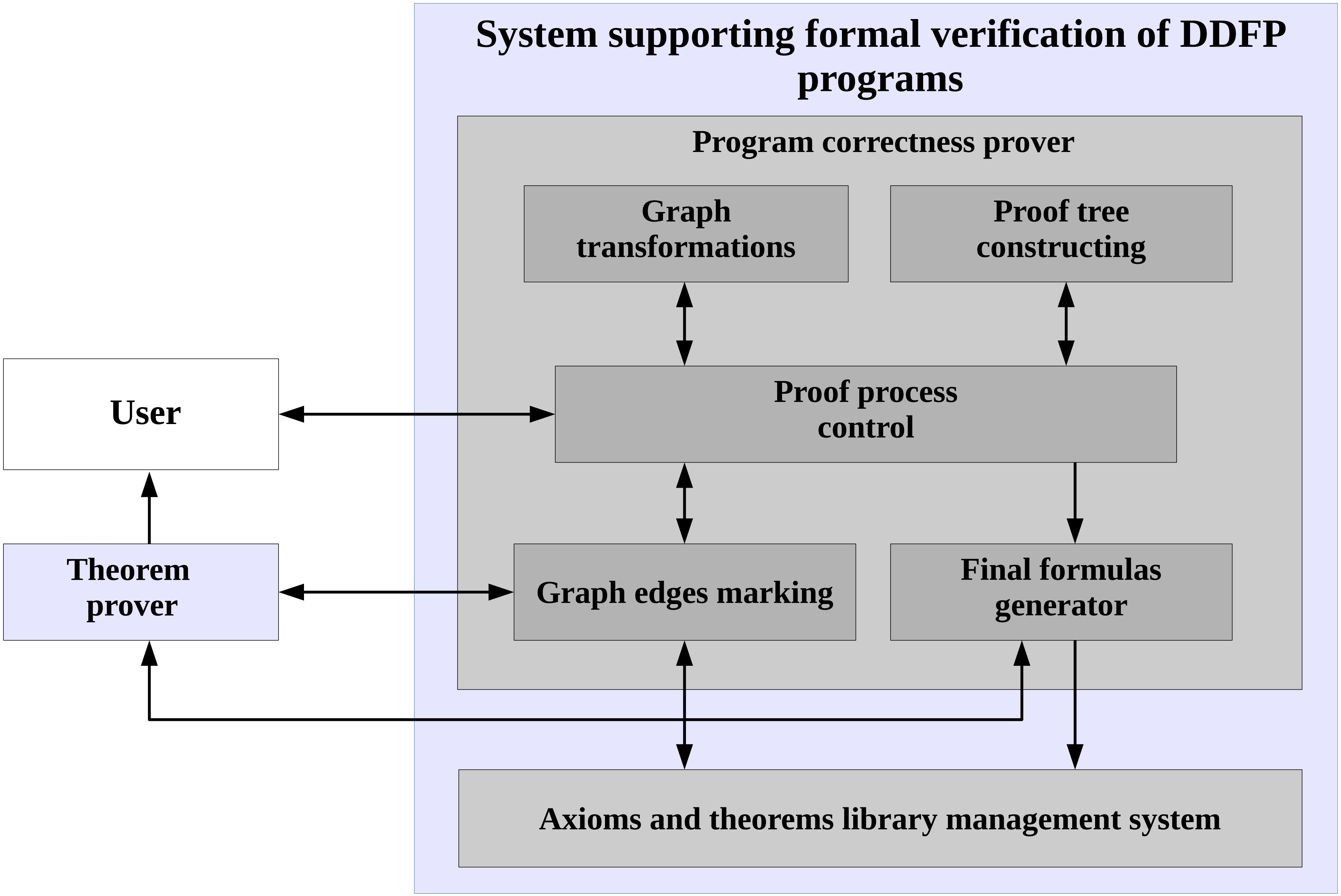
   $\varphi(x,y,q_1,r_1) > \varphi(x,y,q_1+1, r_1-y)$

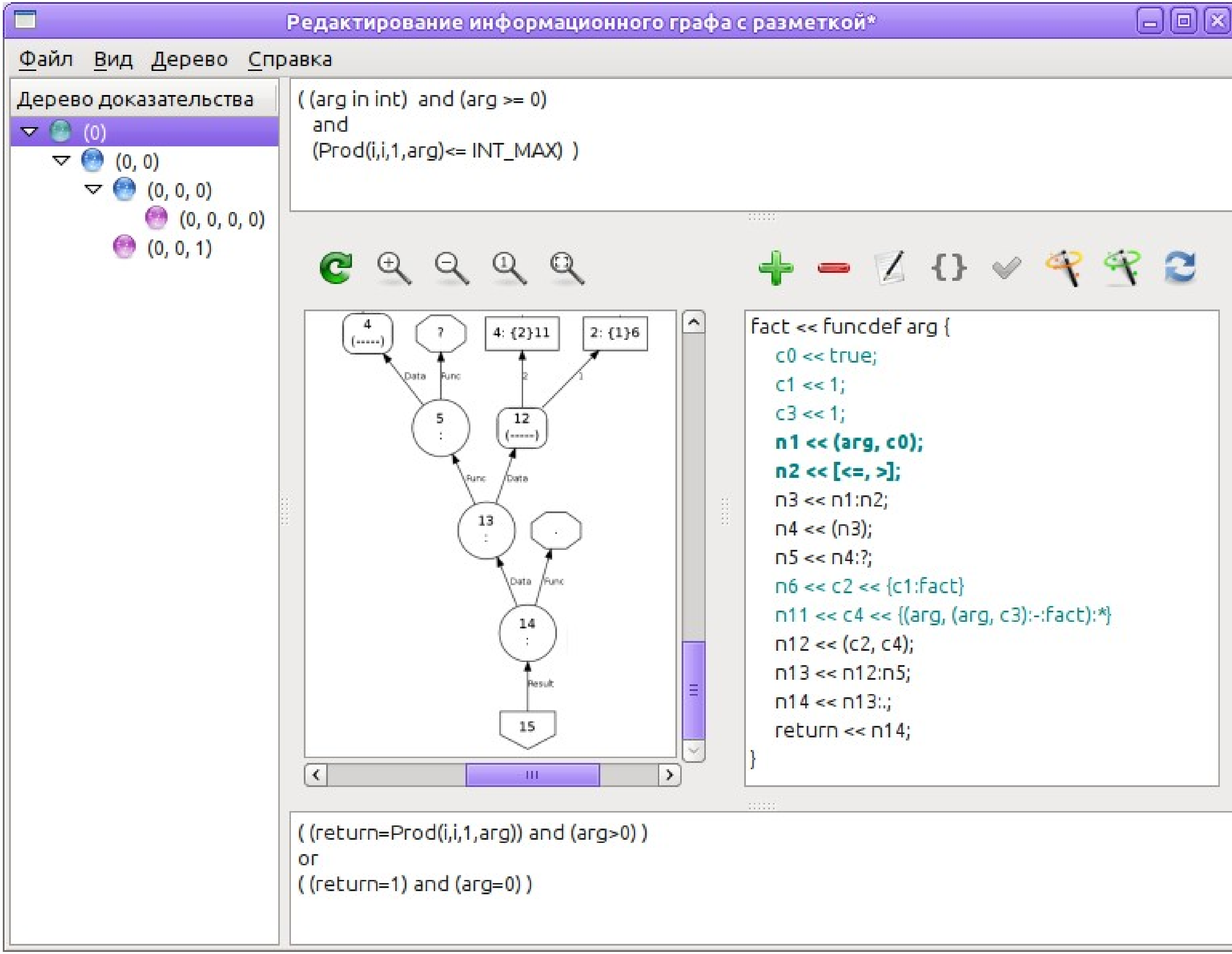
   $x - y \cdot q_1 > x - y \cdot (q_1+1)$

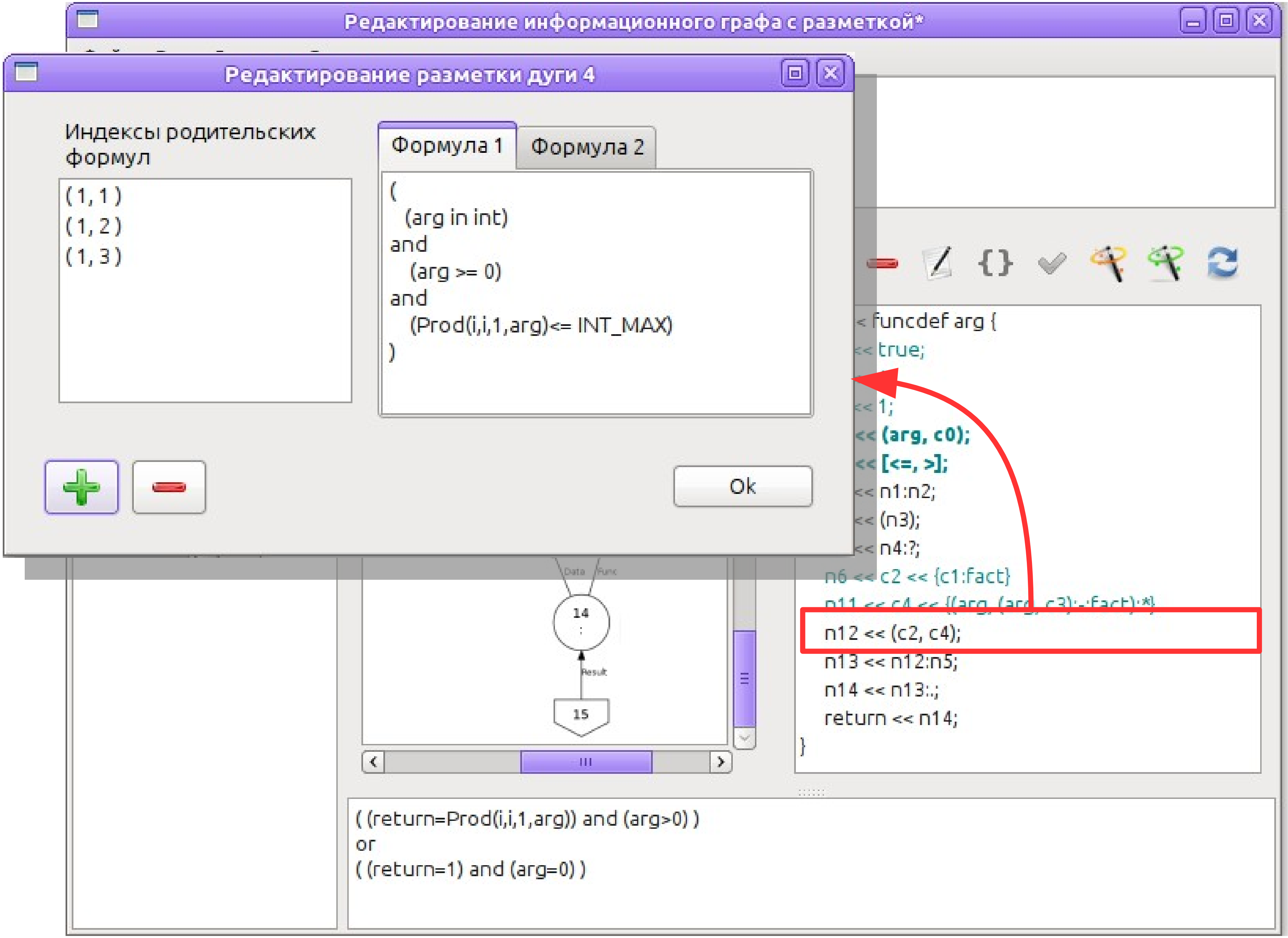
Modified Hoare triple for divR:

$P(x) \wedge (\varphi \in (\text{int} \to \text{int} \to \text{int} \to \text{int} \to \text{int})) \wedge (\varphi = \lambda(x,y,q_1,r_1):\text{int}.x - y \cdot q_1)$ **arg:divR → res** $Q(arg,res) \wedge \varphi(x,y,q_1,r_1) > \varphi(x,y,q_1+1,r_1-y)$

General Scheme of the Toolkit for Supporting Formal Verification of DDFP Programs




Редактирование информационного графа с разметкой*
Файл Вид Дерево Справка
Дерево доказательства
(0)
(0, 0)
(0, 0, 0)
(0, 0, 0, 0)
(0, 0, 1)
((arg in int) and (arg >= 0)
and
(Prod(i,i,1,arg)<= INT_MAX))
fact << funcdef arg {
c0 << true;
c1 << 1;
c3 << 1;
n1 << (arg, c0);
n2 << [<=, >];
n3 << n1:n2;
n4 << (n3);
n5 << n4:?;
n6 << c2 << {c1:fact}
n11 << c4 << {(arg, (arg, c3):-:fact):*}
n12 << (c2, c4);
n13 << n12:n5;
n14 << n13:.;
return << n14;
}
((return=Prod(i,i,1,arg)) and (arg>0))
or
((return=1) and (arg=0))



Редактирование информационного графа с разметкой*
Редактирование разметки дуги 4
Индексы родительских формул
(1, 1)
(1, 2)
(1, 3)
Формула 1
Формула 2
(
(arg in int)
and
(arg >= 0)
and
(Prod(i,i,1,arg)<= INT_MAX)
)
Ok
funcdef arg {
<< true;
<< 1;
<< (arg, c0);
<< [<=, >];
<< n1:n2;
<< (n3);
<< n4:?;
n6 << c2 << {c1:fact}
n12 << (c2, c4);
n13 << n12:n5;
n14 << n13:.;
return << n14;
}
Data
Func
14
Result
15
((return=Prod(i,i,1,arg)) and (arg>0))
or
((return=1) and (arg=0))

Main results and conclusions

- A method based on the Hoare logic for verification of DDFP programs in the Pifagor language has been developed.
  - the semantics of the Pifagor language is formalized,
  - a language for the specification of program properties has been developed,
  - an axiomatic theory based on the Hoare logic was created.
- A method for proving the termination of programs in the Pifagor language is proposed.
- A method for removing the mutual recursion of several functions of the DDFP program is proposed.
- The architecture of the toolkit for supporting the formal verification of DDFP programs is developed.
- A prototype of the toolkit has been developed.

Future development

- **Verification of programs in Pifagor**

  — integrate a theorem proving assistant;

  — aggregate a library of programs with unlimited parallelism;

  — verification of the process of program transferring to real-world architectures.

- **Verification of programs in Smile (a statically-typed successor of Pifagor)**

  — modify the proposed methods to use it for Smile;

  — updating the verification toolkit.

Thank you for your attention!