A Note on Program Specialization

What Syntactical Properties of Residual Programs Can Reveal?

A. P. Lisitsa[1] A. P. Nemytykh[2]

[1]Department of Computer Science
The University of Liverpool

[2]Program Systems Institute
Russian Academy of Sciences

Workshop on Verification and Program Transformation
Vienna, 2014

Traditional Goals of Program Transformation

- The time efficiency of transformed programs
- The space-efficiency of transformed programs
- Translation
- Refactoring programs
- Formatting
- . . .

Traditional Goals of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

The first Futamura projection aims to:

- effective changing of the semantics of the programs;
- effective compilation from one programming language to another;
- the generation of efficient programs implementing inverse functions;
- Undecidability issue
- There still many very interesting examples

Traditional Goals of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

The first Futamura projection aims to:

- effective changing of the semantics of the programs;
- effective compilation from one programming language to another;
- the generation of efficient programs implementing inverse functions;
- Undecidability issue
- There still many very interesting examples

Traditional Goals of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

The first Futamura projection aims to:

- effective changing of the semantics of the programs;
- effective compilation from one programming language to another;
- the generation of efficient programs implementing inverse functions;
- Undecidability issue
- There still many very interesting examples

Traditional Goals of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

The first Futamura projection aims to:

- effective changing of the semantics of the programs;
- effective compilation from one programming language to another;
- the generation of efficient programs implementing inverse functions;
- Undecidability issue
- There still many very interesting examples

Traditional Uses of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

- The second Futamura projection aims to:
  - effective generation of compilers from the interpreters;
    - semi-automated (in some sense – interactive).
- The third Futamura projection aims to:
  - effective genaration of compiler generaters;
    - a source of intricate tasks to be solved.

Traditional Uses of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

- The second Futamura projection aims to:
  - effective generation of compilers from the interpreters;
    - semi-automated (in some sense – interactive).
- The third Futamura projection aims to:
  - effective genaration of compiler generaters;
    - a source of intricate tasks to be solved.

Traditional Uses of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

- The second Futamura projection aims to:
  - effective generation of compilers from the interpreters;
    - semi-automated (in some sense – interactive).
- The third Futamura projection aims to:
  - effective genaration of compiler generaters;
    - a source of intricate tasks to be solved.

Traditional Uses of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

- The second Futamura projection aims to:
  - effective generation of compilers from the interpreters;
    - semi-automated (in some sense – interactive).
- The third Futamura projection aims to:
  - effective genaration of compiler generaters;
    - a source of intricate tasks to be solved.

Traditional Uses of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

- The second Futamura projection aims to:
  - effective generation of compilers from the interpreters;
    - semi-automated (in some sense – interactive).
- The third Futamura projection aims to:
  - effective genaration of compiler generaters;
    - a source of intricate tasks to be solved.

Traditional Uses of Program Specialization.

The efficiency of the transformed programs w.r.t. time.

- The second Futamura projection aims to:
  - effective generation of compilers from the interpreters;
    - semi-automated (in some sense – interactive).
- The third Futamura projection aims to:
  - effective genaration of compiler generaters;
    - a source of intricate tasks to be solved.

There is Something More in Program Transformation

It can be used for:

- analysis of the programs;
- and more specifically for program verification.

Program Verification by Program Trasformation

Related Works - I

- M. Leuschel et al. (1999, . . .)
  suggested to apply a program specialization method for verification of various infinite state computing systems modeled in terms of logic programs;
- A. Pettorossi, M. Proietti et al. (2001, . . .) proposed to use constraint logic programs, which give more powerful means for dealing with infinite sets of states;
- A. Roychoudhury and C. R. Ramakrishnan (2004) used fold/unfold transformations of logic programs for the verification of parameterized concurrent systems;
- G. W. Hamilton (2007) used his distillation algorithm as a proof assistant for transformation of programs into a tail recursive form in which some properties of the programs can be easily verified by the application of inductive proof rules;

Program Verification by Program Trasformation

Related Works - III

- A. Lisitsa and A. Nemytykh (2005, . . .) studied functional modeling and verification (by supercompilation) of global *safety* properties of nondeterministic parameterized cache coherence protocols;
- A. Lisitsa and A. Nemytykh (2008), and A. Klimov (2012) applied supercompilation to verification of Petri Nets models;
- A. Ahmed (2008), A. Lisitsa and A. Nemytykh (2013) addressed the verification of the cryptographic protocols via supercompilation using the functional modeling;
- Antonina Nepeivoda (2013, 2014) modeled and verified a class of the ping-pong crypto-protocols by use of a generalization based on Turchin's relation (1988).

Solution of Two Combinatorial Problems

by Turchin's Supercompilation

Specialization can be used for the solution of combinatorial and algebraic problems encoded in the programs.

- Two examples illustrating use of Turchin's supercompilation for:
  - Describing the solution set of a word equation;
  - Solving "Missionaries and Cannibals" puzzle.

We hope that these examples will be able to motivate further research in (semi)automated program specialization.

Syntactical Properties of Residual Programs

Produced by Supercompilation

Recipe

- Given a combinatorial problem, encode the corresponding dynamics by a functional iterpreter Int;
- Specialize Int w.r.t. the initial parametrized state of the computing system;
- Analyze relation between the given encoded problem and the syntactical properties of the residual program.

Syntactical Properties of Residual Programs

Produced by Supercompilation

Recipe

- Given a combinatorial problem, encode the corresponding dynamics by a functional iterpreter Int;
- Specialize Int w.r.t. the initial parametrized state of this computing system;
- Analyze relation between the given encoded problem and the syntactical properties of the residual program.

Syntactical Properties of Residual Programs

Produced by Supercompilation

Recipe

- Given a combinatorial problem, encode the corresponding dynamics by a functional iterpreter Int;
- Specialize Int w.r.t. the initial parametrized state of this computing system;
- Analyze relation between the given encoded problem and the syntactical properties of the residual program.

Interactive Use of a Supercompiler

on the Base of the Analysis of Intermediate Residual Programs

If necessary:

- Correct Int, basing on the analysis of the syntactical properties of the (intermediate) residual program;
- Specialize the corrected Int encoding the problem to be solved;
- Repeat the previous two steps.

Until the obtained residual programs do display the solution of the given combinatorial problem explicitly.

Interactive Use of a Supercompiler

on the Base of the Analysis of Intermediate Residual Programs

If necessary:

- Correct Int, basing on the analysis of the syntactical properties of the (intermediate) residual program;
  - thus we give the supercompiler
    a hint of clarification of the problem definition;
- Specialize the corrected Int encoding the problem to be solved;
- Repeat the previous two steps.

Until the obtained residual programs do display the solution of the given combinatorial problem explicitly.

Interactive Use of a Supercompiler

on the Base of the Analysis of Intermediate Residual Programs

If necessary:

- Correct Int, basing on the analysis of the syntactical properties of the (intermediate) residual program;
  - thus we give the supercompiler
    a hint of clarification of the problem definition;
- Specialize the corrected Int encoding the problem to be solved;
- Repeat the previous two steps.

Until the obtained residual programs do display the solution of the given combinatorial problem explicitly.

Interactive Use of a Supercompiler

on the Base of the Analysis of Intermediate Residual Programs

If necessary:

- Correct Int, basing on the analysis of the syntactical properties of the (intermediate) residual program;
  - thus we give the supercompiler
    a hint of clarification of the problem definition;
- Specialize the corrected Int encoding the problem to be solved;
- Repeat the previous two steps.

Until the obtained residual programs do display the solution of the given combinatorial problem explicitly.

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` $\equiv$ `'ba'`

`'aba':e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` ≡ `'ba'`

`'aba':e.x` ≡ `'a':'b':'a':e.x`

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` $\equiv$ `'ba'`

`'aba':e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` $\equiv$ `'ba'`

`'aba':e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` ≡ `'ba'`

`'aba':e.x` ≡ `'a':'b':'a':e.x`

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` ≡ `'ba'`

`'aba':e.x` ≡ `'a':'b':'a':e.x`

Term Rewriting Systems

Based on Pattern Matching

Example Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- The sentences are ordered to be matched;
- Two kinds of variables:
  - s.-variables;
    - range over characters and identifiers;
  - e.-variables;
    - range over the whole set of the S-expressions;

`'b':'a':[]` $\equiv$ `'ba'`

`'aba':e.x` $\equiv$ `'a':'b':'a':e.x`

Definitions

Definition

Given a finite alphabet Σ of constants and a set of variables X disjoint with Σ, a word expression is $L = R$, where $L, R \in (\Sigma \cup X)^*$.

Definition

A solution of a word equation $L = R$ is any substitution of its unknowns in the word equation by words from Σ^* that turns the equation $L = R$ into literal/syntactic equality.

Simplest Examples

Example

```
'ab' ++ e.xs = e.xs ++ 'ba'
```

- `e.xs := 'a'` is a solution;
- `e.xs := 'aba'` is a solution;
- ...

Example

```
'ab' ++ e.xs ++ 'a' = e.xs ++ 'ba'
```

There is no solution.

On Makanin's Algorithm

Problem

Given a word equation, describe its solution set by a constructive (transparent) way.

- Yu. I. Khmelevskiĭ (the late 1960's) an important contribution in solving the problem;
- G. S. Makanin (1970's) suggested an algorithm deciding whether or not there exists a solution of any given word equation.
- There is an algorithm based on Makanin's ideas, which for a given word equation generates a finite graph describing the corresponding solution set.
- Supercompilation is able to generate the same graph as Makanin's algorithm does for some word equations.

On Makanin's Algorithm

Problem

Given a word equation, describe its solution set by a constructive (transparent) way.

- Yu. I. Khmelevskiĭ (the late 1960's) an important contribution in solving the problem;
- G. S. Makanin (1970's) suggested an algorithm deciding whether or not there exists a solution of any given word equation.

- There is an algorithm based on Makanin's ideas, which for a given word equation generates a finite graph describing the corresponding solution set.
- Supercompilation is able to generate the same graph as Makanin's algorithm does for some word equations.

On Makanin's Algorithm

Problem

Given a word equation, describe its solution set by a constructive (transparent) way.

- Yu. I. Khmelevskiĭ (the late 1960's) an important contribution in solving the problem;
- G. S. Makanin (1970's) suggested an algorithm deciding whether or not there exists a solution of any given word equation.

- There is an algorithm based on Makanin's ideas, which for a given word equation generates a finite graph describing the corresponding solution set.
- Supercompilation is able to generate the same graph as Makanin's algorithm does for some word equations.

Solving Word Equations

by Supercompilation

Input Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- `equal` is a predicate checking whether two given words are equal.
- `main` is a predicate testing whether a given word is a solution of the equation:
  - `'ab' ++ e.xs = e.xs ++ 'ba'`

Solving Word Equations

by Supercompilation

Input Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- `equal` is a predicate checking whether two given words are equal.
- `main` is a predicate testing whether a given word is a solution of the equation:
  - `'ab' ++ e.xs = e.xs ++ 'ba'`

Solving Word Equations

by Supercompilation

Input Program

```
main(e.xs) = equal('ab' ++ e.xs, e.xs ++ 'ba');
equal(s.x:e.xs, s.x:e.ys) = equal(e.xs, e.ys);
equal([], []) = True;
equal(e.xs, e.ys) = False;
```

- `equal` is a predicate checking whether two given words are equal.
- `main` is a predicate testing whether a given word is a solution of the equation:
  - `'ab' ++ e.xs = e.xs ++ 'ba'`

Solving Word Equations

by Supercompilation

Residual program `P` produced by the supercompiler SCP4

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
main( e.xs ) = False;
```

- The last sentence is redundant for description of the solution set.
- This sentence will be absent, if we will remove the last sentence of the original function `equal`.

Solving Word Equations

by Supercompilation

Residual program `P` produced by the supercompiler SCP4

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
main( e.xs ) = False;
```

- The last sentence is redundant for description of the solution set.
- This sentence will be absent, if we will remove the last sentence of the original function `equal`.

Solving Word Equations

by Supercompilation

Residual program `P`

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
```

`P` (as any program!) can be seen as a graph:

- vertices of the graph correspond to:
  - the function calls;
  - and the return expressions;
- edges are labeled with:
  - the case expressions
  - or assignments.

Solving Word Equations

by Supercompilation

Residual program `P`

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
```

`P` (as any program!) can be seen as a graph:

- vertices of the graph correspond to:
  - the function calls;
  - and the return expressions;
- edges are labeled with:
  - the case expressions
  - or assignments.

Solving Word Equations

by Supercompilation

Residual program `P`

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
```

`P` (as any program!) can be seen as a graph:

- vertices of the graph correspond to:
  - the function calls;
  - and the return expressions;
- edges are labeled with:
  - the case expressions
  - or assignments.

Solving Word Equations

by Supercompilation

Residual program `P`

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
```

`P` (as any program!) can be seen as a graph:

- vertices of the graph correspond to:
  - the function calls;
  - and the return expressions;
- edges are labeled with:
  - the case expressions
  - or assignments.

Solving Word Equations

by Supercompilation

Residual program `P`

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
```

`P` (as any program!) can be seen as a graph:

- vertices of the graph correspond to:
  - the function calls;
  - and the return expressions;
- edges are labeled with:
  - the case expressions
  - or assignments.

Solving Word Equations

Syntactical Properties of Residual Programs

Residual program P

```
main( 'a':'b':e.xs ) = main( e.xs );
main( 'a':[] ) = True;
```

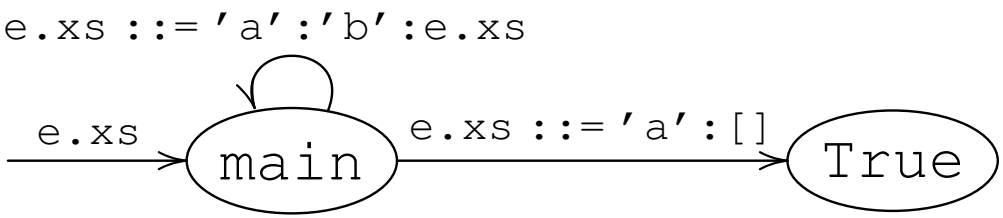

Figure: Graph describing the solution set `('ab')*'a'` of the word equation.

Solving Word Equations

Supercompilation vs. Makanin's algorithm

Word Equation

```
'ab' ++ e.xs = e.xs ++ 'ba'
```

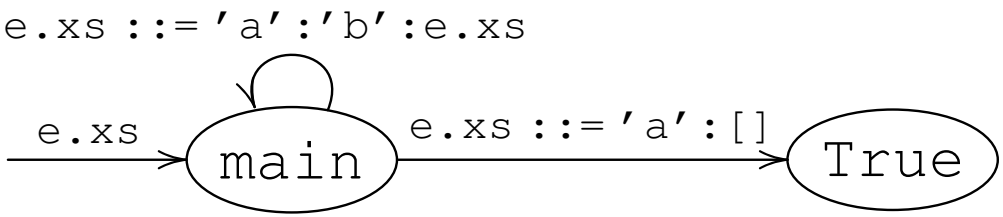

Figure: Graph describing the solution set `('ab')*'a'` of the word equation.

The obtained graph coincides with the graph generated by Makanin's algorithm.

Solving Word Equations

Supercompilation vs. Makanin's algorithm

Word Equation

```
'ab' ++ e.xs ++ 'a' = e.xs ++ 'ba'
```

Supercompilation of the following task (by the supercompiler SCP4)

```
main(e.xs) = equal('ab'++e.xs++'a',e.xs++'ba');
```

yields the residual program:

```
main(e.xs) = False;
```

Solving Word Equations

Supercompilation vs. Makanin's algorithm

- A result of specialization may be interesting not only because of effective performance of the residual program as compared to a source program.
- The syntactical structure of the result may serve as a solution to the problem encoded in the source program.
- For arbitrary word equations the supercompiler SCP4, in general, is not able to reproduce the same graph as Makanin's algorithm does.
- The ideas underlying Makanin's algorithm may be borrowed for further development of the supercompilation method.

Solving Word Equations

Supercompilation vs. Makanin's algorithm

- A result of specialization may be interesting not only because of effective performance of the residual program as compared to a source program.
- The syntactical structure of the result may serve as a solution to the problem encoded in the source program.
- For arbitrary word equations the supercompiler SCP4, in general, is not able to reproduce the same graph as Makanin's algorithm does.
- The ideas underlying Makanin's algorithm may be borrowed for further development of the supercompilation method.

Solving Word Equations

Supercompilation vs. Makanin's algorithm

- A result of specialization may be interesting not only because of effective performance of the residual program as compared to a source program.
- The syntactical structure of the result may serve as a solution to the problem encoded in the source program.
- For arbitrary word equations the supercompiler SCP4, in general, is not able to reproduce the same graph as Makanin's algorithm does.
- The ideas underlying Makanin's algorithm may be borrowed for further development of the supercompilation method.

Solving Word Equations

Supercompilation vs. Makanin's algorithm

- A result of specialization may be interesting not only because of effective performance of the residual program as compared to a source program.
- The syntactical structure of the result may serve as a solution to the problem encoded in the source program.
- For arbitrary word equations the supercompiler SCP4, in general, is not able to reproduce the same graph as Makanin's algorithm does.
- The ideas underlying Makanin's algorithm may be borrowed for further development of the supercompilation method.

“Missionaries and Cannibals” Puzzle

The Classical Problem

Three missionaries and three cannibals come to the bank of a river and see a boat. They want to cross the river. The boat can carry no more than two people. At no time should the number of cannibals on either bank of the river (including the moored boat) exceed the number of missionaries. How (if at all) is it possible to cross the river?

Generalization

Given n missionaries and k cannibals is it possible to cross the river and to save all the missionaries?
If the answer is true, then we are interested in the algorithm carrying the strange crowd.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
  - If the answer is:
    - `True:e.p`, then the rest of the path, which did not take part in moving the crowd;
    - `False:e.q`, then the part of the crowd brought to the right bank.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
    If the answer is:
    - True, then the rest of the path, which did not take a part in moving the crowd;
    - False, then the part of the crowd brought to the right bank.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
    If the answer is:
    - True, then the rest of the path, which did not take a part in moving the crowd;
    - False, then the part of the crowd brought to the right bank.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
    If the answer is:
    - True, then the rest of the path, which did not take a part in moving the crowd;
    - False, then the part of the crowd brought to the right bank.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
    If the answer is:
    - True, then the rest of the path, which did not take a part in moving the crowd;
    - False, then the part of the crowd brought to the right bank.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
    If the answer is:
    - True, then the rest of the path, which did not take a part in moving the crowd;
    - False, then the part of the crowd brought to the right bank.

Encoding

A. V. Korlyukov’s Idea

The dynamic system moving the crowd from the left bank to the right.

```
mainInt(e.l, e.path) ⇒ True:e.p | False:e.q
```

- Takes on:
  - the pair n, k describing the initial crowd on the left bank;
  - a finite sequence of the boat states (an evaluation path).
- Returns:
  - does a prefix of the path bring the crowd to the right bank?
    If the answer is:
    - True, then the rest of the path, which did not take a part in moving the crowd;
    - False, then the part of the crowd brought to the right bank.

The Banks' States n, k (the First Trick)

A. V. Korlyukov's Idea

The pair n, k is encoded as a triple of nonnegative integers m, p, c such that

$$m = \max\{n - k, 0\}, c = \max\{k - n, 0\}, n = m + p, k = c + p$$

Unary notation is used to represent the integers m, p, c.

- m is is the overweight of missionaries compared to cannibals;
- c is the same vice versa;
- p is the number of those who are outnumbered.

The Banks' States n, k (the First Trick)

A. V. Korlyukov's Idea

The pair n, k is encoded as a triple of nonnegative integers m, p, c such that

$$m = \max\{n - k, 0\}, c = \max\{k - n, 0\}, n = m + p, k = c + p$$

Unary notation is used to represent the integers m, p, c.

- m is is the overweight of missionaries compared to cannibals;
- c is the same vice versa;
- p is the number of those who are outnumbered.

The Banks' States n, k (the First Trick)

A. V. Korlyukov's Idea

The pair n, k is encoded as a triple of nonnegative integers m, p, c such that

$$m = \max\{n - k, 0\}, c = \max\{k - n, 0\}, n = m + p, k = c + p$$

Unary notation is used to represent the integers m, p, c.

- m is is the overweight of missionaries compared to cannibals;
- c is the same vice versa;
- p is the number of those who are outnumbered.

The Banks' States n, k (the First Trick)

A. V. Korlyukov's Idea

The pair n, k is encoded as a triple of nonnegative integers m, p, c such that

$$m = \max\{n - k, 0\}, c = \max\{k - n, 0\}, n = m + p, k = c + p$$

Unary notation is used to represent the integers m, p, c.

- m is is the overweight of missionaries compared to cannibals;
- c is the same vice versa;
- p is the number of those who are outnumbered.

The Banks' States n, k (the First Trick)

A. V. Korlyukov's Idea

The pair n, k is encoded as a triple of nonnegative integers m, p, c such that

$$m = \max\{n - k, 0\}, c = \max\{k - n, 0\}, n = m + p, k = c + p$$

Example

`[['mm'],['ppp'],[]]` – five missionaries and three cannibals.

Example

`[[],['pp'],['c']]` – two missionaries and three cannibals.

The Boat States

A. V. Korlyukov’s Idea

A state of the boat is encoded as one identifier, the name of which consists of the first capital letters of its passengers.

- `MM` – the state with two missionaries on the boat;
- `C` – the state with one cannibal.

The Second Trick

A. V. Korlyukov’s Idea

Interpretation Example

```
mainInt( [[],['ppp'],[]],
  [CC,C,CC,C,MM,MC,MM,C,CC,M,MC,MC] ) = [True,[MC]];
```

- By definition, the last `MC` does matter. It is not used to solve the puzzle.
- `[CC,C,CC,C,MM,MC,MM,C,CC,M,MC]` is a solution path.

This property of the interpreter is one of the tricks incidental to the supercompiler SCP4.

The Second Trick

A. V. Korlyukov’s Idea

Interpretation Example

```
mainInt( [[],['ppp'],[]],
  [CC,C,CC,C,MM,MC,MM,C,CC,M,MC,MC] ) = [True,[MC]];
```

- By definition, the last `MC` does matter. It is not used to solve the puzzle.
- `[CC,C,CC,C,MM,MC,MM,C,CC,M,MC]` is a solution path.

This property of the interpreter is one of the tricks incidental to the supercompiler SCP4.

The Second Trick

A. V. Korlyukov’s Idea

Interpretation Example

```
mainInt( [[],['ppp'],[]],
  [CC,C,CC,C,MM,MC,MM,C,CC,M,MC,MC] ) = [True,[MC]];
```

- By definition, the last `MC` does matter. It is not used to solve the puzzle.
- `[CC,C,CC,C,MM,MC,MM,C,CC,M,MC]` is a solution path.

This property of the interpreter is one of the tricks incidental to the supercompiler SCP4.

The Dynamic System - I

A. V. Korlyukov’s Idea

```
mainInt(e.l, [s.a, e.path]) =
    Int(s.a, Move(s.a,L,e.l,[[],[],[]]),e.path);

/* The boat on the left bank. <.,.,.> is a tuple.*/
Move( s.a, L, e.l, e.r ) =
    <R, Minus(s.a, e.l), Plus(s.a, e.r)>;
/* The boat on the right bank. */
Move( s.a, R, e.l, e.r ) =
    <L, Minus(s.a, e.l), Plus(s.a, e.r)>;
.................
```

- Move modifies the banks’ states and returns the new active bank and the two modified states.

The Dynamic System - I

A. V. Korlyukov's Idea

```
mainInt(e.l, [s.a, e.path]) =
    Int(s.a, Move(s.a,L,e.l,[[],[],[]]),e.path);

/* The boat on the left bank. <.,.,.> is a tuple.*/
Move( s.a, L, e.l, e.r ) =
    <R, Minus(s.a, e.l), Plus(s.a, e.r)>;
/* The boat on the right bank. */
Move( s.a, R, e.l, e.r ) =
    <L, Minus(s.a, e.l), Plus(s.a, e.r)>;
.................
```

- `Move` modifies the banks' states and returns the new active bank and the two modified states.

The Dynamic System - II

A. V. Korlyukov’s Idea

```
Int( s.pa, R, [[],[],[]], e.r, e.path ) =
    True : e.path;
Int( s.pa, s.d, e.l, e.r, [] ) = False : e.r;
Int( s.pa, s.d, e.l, e.r, [] ) =
    CutFalse([]);
Int( s.pa, s.d, e.l, e.r, [s.pa,e.path] ) =
    BlockRepetition([]);
Int( s.pa, s.d, e.l, e.r, [s.x,e.path] ) =
    Int(s.x, Move(s.x, s.l, e.l, e.r), e.path);
```

- We skip over the functions `CutFalse`, `BlockRepetition` and will consider them later.

The Dynamic System - II

A. V. Korlyukov’s Idea

```
Int( s.pa, R, [[],[],[]], e.r, e.path ) =
    True : e.path;
Int( s.pa, s.d, e.l, e.r, [] ) = False : e.r;
Int( s.pa, s.d, e.l, e.r, [] ) =
    CutFalse([]);
Int( s.pa, s.d, e.l, e.r, [s.pa,e.path] ) =
    BlockRepetition([]);
Int( s.pa, s.d, e.l, e.r, [s.x,e.path] ) =
    Int(s.x, Move(s.x, s.l, e.l, e.r), e.path);
```

- We skip over the functions `CutFalse, BlockRepetition` and will consider them later.

Unary Arithmetic

A. V. Korlyukov’s Idea

```
Minus(MM, [['mm',e.m],e.p,[]]) = [e.m,e.p,[]];
Minus(MM, [[],['pp'],[]]) = [[],[],['cc']];
Minus(MM, [['m'],['p'],[]]) = [[],[],['c']];
Minus(CC, [[],[],['cc', e.c]]) = [[],[],e.c];
Minus(CC,[e.m,['pp',e.p],[]])= [['mm',e.m],e.p,[]];
Minus(MC, [e.m,['p',e.p],[]]) = [e.m,e.p,[]];
Minus(M, [[],['p'],[]]) = [[],[],['c']];
Minus(M, [['m',e.m],e.p,[]]) = [e.m,e.p,[]];
Minus(C, [[],[],['c',e.c]]) = [[],[],e.c];
Minus(C, [e.m,['p',e.p],[]]) = [['m',e.m],e.p,[]];

Plus(...) = ...;
.................
```

The Specialization Task

A. V. Korlyukov's Idea

The Problem in Program Terms

For the parameterized call `mainInt(e.l`$_0$`, e.path)` and a given state `e.l`$_0$ on the left bank we are interested in an answer to the question:
Does a path `e.path`$_0$ exist such that (a part of) the result of the call is `True`?

We answer the given question interactively using the supercompiler SCP4 and analyzing the residual programs produced by the supercompiler.

The Specialization Task

A. V. Korlyukov’s Idea

By the precondition we narrow the initial left state’s set

```
e.left = [[e.m],[e.p],[e.c]]
```

to one of the forms:

`[[e.m],[e.p],[]]` and `[[],[],[e.c]]`

Thus we have the two tasks:

- `mainInt([[e.m],[e.p],[]], e.path)`
- `mainInt([[],[],[e.c]], e.path)`

to be solved.

Two More Tricks

A. V. Korlyukov's Idea

To make the residual programs more compact we use:

The Third Trick

Among the paths, moving the boat, there are meaningless paths. The simplest of them contain at least two identical states staying side by side in the path.
We exclude such paths. We do that by calling the function `BlockRepetition` with an argument which never matches the function definition.

```
... MC,MC, ...
```

Two More Tricks

A. V. Korlyukov’s Idea

To make the residual programs more compact we use:

```
.................
Int( s.pa, s.d, e.l, e.r, [s.pa,e.path] ) =
    BlockRepetition([]);
.................
BlockRepetition( Deadlock ) = [];
.................
```

- The first argument is a pair of the form [s.pa,e.path] to recognize two identical states staying side by side in a given path.

Two More Tricks

A. V. Korlyukov’s Idea

To make the residual programs more compact we use:

```
.................
Int( s.pa, s.d, e.l, e.r, [s.pa,e.path] ) =
    BlockRepetition([]);
.................
BlockRepetition( Deadlock ) = [];
.................
```

- The first argument `s.pa` of the function `Int` serves to recognize two identical states staying side by side in a given path.

Yet Two Tricks

A. V. Korlyukov’s Idea

For the same reason, whenever we hope for a positive answer to the problem question we will replace the following sentence of `Int`

```
Int( s.pa, s.d, e.l, e.r, [] ) = False : e.r;
```

with the sentence

```
.................
Int( s.pa, s.d, e.l, e.r, [] ) = CutFalse([]);
.................
CutFalse( Deadlock ) = [];
.................
```

Diagonal Cases

A. V. Korlyukov’s Idea

```
mainInt([[],[e.p],[]], e.path)
```

Our first experiment is for two missionaries and two cannibals:

```
mainInt([[],['pp'],[]], e.path)
```

The program contains the sentence with the call `CutFalse([])`.

Diagonal Cases

`mainInt([[],['pp'],[]], e.path)`

Supercompilation produces the following residual program:

```
mainInt'([s.x , e.path]) = True:[f(s.x,e.path)];

f(CC, [C,MM,M,MC,e.path]) = e.path;
f(CC, [C,MM,C,CC,e.path]) = e.path;
f(CC, [C,M,MC,s.x,e.path]) = f(s.x, e.path);
f(MC, [M,MM,M,MC,e.path]) = e.path;
f(MC, [M,MM,C,CC,e.path]) = e.path;
f(MC, [M,C,CC,s.x,e.path]) = f(s.x, e.path);
```

- The exits from the recursion show the shortest path, moving the crowd to the right bank.

Diagonal Cases

`mainInt([[],['pp'],[]], e.path)`

Supercompilation produces the following residual program:

```
mainInt'([s.x , e.path]) = True:[f(s.x,e.path)];

f(CC, [C,MM,M,MC,e.path]) = e.path;
f(CC, [C,MM,C,CC,e.path]) = e.path;
f(CC, [C,M,MC,s.x,e.path]) = f(s.x, e.path);
f(MC, [M,MM,M,MC,e.path]) = e.path;
f(MC, [M,MM,C,CC,e.path]) = e.path;
f(MC, [M,C,CC,s.x,e.path]) = f(s.x, e.path);
```

- The exits from the recursion show the shortest paths moving the crowd to the right bank.

Diagonal Cases

`mainInt([[],['p'],[]], e.path)`

One missionary and one cannibal.

The program contains the sentence with the call `CutFalse([])`.

The residual program:

```
mainInt'([MC , e.path]) = True : [e.path];
```

Diagonal Cases

`mainInt([[],['ppp'],[]], e.path)`

The program contains the sentence with the call `CutFalse([])`.
The residual program:

```
mainInt'([s.x , e.path]) = True : [f(s.x,e.path)];

f(CC,[C,CC,C,MM,MC,MM,C,CC,M,MC,e.path]) = e.path;
f(CC,[C,CC,C,MM,MC,MM,C,CC,C,CC,e.path]) = e.path;
f(CC,[C,M,MC,s.x,e.path]) = f(s.x, e.path);
f(MC,[M,CC,C,MM,MC,MM,C,CC,M,MC,e.path]) = e.path;
f(MC,[M,CC,C,MM,MC,MM,C,CC,C,CC,e.path]) = e.path;
f(MC,[M,C,CC,s.x,e.path]) = f(s.x, e.path);
```

- The residual program specifies the invariable of the problem.

Diagonal Cases

`mainInt([[],['ppp'],[]], e.path)`

The program contains the sentence with the call `CutFalse([])`.
The residual program:

```
mainInt'([s.x , e.path]) = True : [f(s.x,e.path)];

f(CC,[C,CC,C,MM,MC,MM,C,CC,M,MC,e.path]) = e.path;
f(CC,[C,CC,C,MM,MC,MM,C,CC,C,CC,e.path]) = e.path;
f(CC,[C,M,MC,s.x,e.path]) = f(s.x, e.path);
f(MC,[M,CC,C,MM,MC,MM,C,CC,M,MC,e.path]) = e.path;
f(MC,[M,CC,C,MM,MC,MM,C,CC,C,CC,e.path]) = e.path;
f(MC,[M,C,CC,s.x,e.path]) = f(s.x, e.path);
```

- The residual program specifies the whole set of the successful paths.

Diagonal Cases

```
mainInt([[],['pppp', e.p],[]], e.path)
```

An equal number of missionaries and cannibals, and it > 3.

If the source program contains the sentence with:

- `False`, then the residual never returns `True` and that is a syntactical property of the program.
  The source program terminates for any given arguments.
  Hence there exist no paths moving the crowd to the right bank.
- The call `Can Pass e.p`, then the residual program is equal to a trivial program with the empty domain and `True`.
  Syntactically we might infer the negative response to the problem question from the emptiness of the residual program.

Diagonal Cases

```
mainInt([[],['pppp', e.p],[]], e.path)
```

An equal number of missionaries and cannibals, and it > 3.

If the source program contains the sentence with:

- `False`, then the residual never returns `True` and that is a syntactical property of the program.
  The source program terminates for any given arguments.
  Hence there exist no paths moving the crowd to the right bank.
- The call `CutFalse([])`, then the residual program is empty: a trivial program with the empty domain and this property is syntactical. We might infer the negative answer to the problem question from the emptiness of the residual program.

The Number of Missionaries is Greater Than the Number of Cannibals

`mainInt([['m',e.m],[e.p],[]], e.path)`

In all experiments concerning this case the source programs contain the sentence with the call `CutFalse([])`.

SCP4 produces quite a large residual program:

- Unfortunately we have to analyze the residual program.
- The method used in the first example cannot be applied here, because it is unknown in advance to which pair of the numbers a given exit from the recursion corresponds, and maybe for a pair there exist no exits from the recursion at all (i.e. all paths starting with a given pair lead to an abnormal stop of the program).

The Number of Missionaries is Greater Than the Number of Cannibals

```
mainInt([['m',e.m],[e.p],[]], e.path)
```

In all experiments concerning this case the source programs contain the sentence with the call `CutFalse([])`.

SCP4 produces quite a large residual program:

- Unfortunately we have to analyze the residual program.
- The method used in the first example cannot be applied here, because it is unknown in advance to which pair of the numbers a given exit from the recursion corresponds, and maybe for a pair there exist no exits from the recursion at all (i.e. all paths starting with a given pair lead to an abnormal stop of the program).

The Number of Missionaries is Greater Than the Number of Cannibals

```
mainInt([['m',e.m],[e.p],[]], e.path)
```

A large residual program may be indirect evidence of the fact that the set of the successful paths is too large.

Assuming that we may narrow the paths' set, where we are looking for the successful path's witnesses. The idea is to increase stepwise the number of the people on the right bank and in such a way to approach a resolution of the task.

The boat crossing the river in the direction to the right bank may have only three states `MM`, `MC`, `MC`, while it crossing the river in the opposite direction may have only two states `M`, `C`.

The Number of Missionaries is Greater Than the Number of Cannibals

`mainInt([['m',e.m],[e.p],[]], e.path)`

With such an aim we restrict the boat states. We change the original function `Move` with the following:

```
/* The boat on the left bank. */
Move( MM, L, e.l, e.r ) =
    <R, Minus(MM, e.l), Plus(MM, e.r)>;
Move( MC, L, e.l, e.r ) =
    <R, Minus(MC, e.l), Plus(MC, e.r)>;
Move( CC, L, e.l, e.r ) =
    <R, Minus(CC, e.l), Plus(CC, e.r)>;
```

This function filters out the forbidden states of the boat.

The Number of Missionaries is Greater Than the Number of Cannibals

```
mainInt([['m',e.m],[e.p],[]], e.path)
```

This function filters out the forbidden states of the boat.

```
/* The boat on the right bank. */
Move( M, R, e.l, e.r ) =
    <L, Minus(M, e.l), Plus(M, e.r)>;
Move( C, R, e.l, e.r ) =
    <L, Minus(C, e.l), Plus(C, e.r)>;
```

The Number of Missionaries is Greater Than the Number of Cannibals

`mainInt([['m'],[ 'pp'],[]], e.path)`

The residual program:

```
mainInt'([MC,M,MM,M,MM,M,MC,e.path])=True:[e.path];
mainInt'([MC,M,MM,M,MM,C,CC,e.path])=True:[e.path];
mainInt'([MC,M,MM,M,MC,C,MC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MM,M,MC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MM,C,CC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MC,C,MC,e.path])=True:[e.path];
mainInt'([CC,C,MM,M,MM,M,MC,e.path])=True:[e.path];
mainInt'([CC,C,MM,M,MM,C,CC,e.path])=True:[e.path];
mainInt'([CC,C,MM,M,MC,C,MC,e.path])=True:[e.path];
```

The Number of Missionaries is Greater Than the Number of Cannibals

```
mainInt([['m'],[ 'pp'],[]], e.path)
```

```
.................
mainInt'([MC,C,MC,M,MM,M,MC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MM,C,CC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MC,C,MC,e.path])=True:[e.path];
.................
```

- If the boat is on the left bank, then the path `[MC,C,MC,M]` adds a pair (M-C) on the right bank, provided that on the left bank the number of the missionaries greater than the number of the cannibals. Repeated iteration of such a path leads to success.
- The path `[MC,C,MC,C]` decides the problem of two missionaries and one cannibal.

The Number of Missionaries is Greater Than the Number of Cannibals

```
mainInt([['m'],[ 'pp'],[]], e.path)
```

```
.................
mainInt'([MC,C,MC,M,MM,M,MC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MM,C,CC,e.path])=True:[e.path];
mainInt'([MC,C,MC,M,MC,C,MC,e.path])=True:[e.path];
.................
```

- If the boat is on the left bank, then the path `[MC,C,MC,M]` adds a pair (M-C) on the right bank, provided that on the left bank the number of the missionaries greater than the number of the cannibals. Repeated iteration of such a path leads to success.
- The path `[MC,C,MC]` decides the problem with two missionaries and one cannibal.

The Other Cases

- the number of the cannibals is greater than the number of the missionaries;
- no cannibals;
- no missionaries.

These cases are trivial.

Summary

M.→ C.↓	0	1	2	3	4	5	6	7
0	True	True	True	True	True	True	True	True
1	True	True	True	True	True	True	True	True
2	True		True	True	True	True	True	True
3	True			True	True	True	True	True
4	True					True	True	True
5	True						True	True
6	True							True
7	True							

Conclusion

- We explored syntactical properties of the residual programs.
- SCP4 was exploited as a kind of a “PROLOG interpreter”.
- Unlike PROLOG, when the substitution set cannot be described as a finite union of parameterized lists, SCP4 produces a finite residual program describing the substitution set more transparently as compared with the source program.
- The dynamics system “missionaries-cannibals” may be considered as a nondeterministic protocol.
- Actually all our experiments were done in a strict functional programming language REFAL.

Thank You