# A Note on Program Specialization

## What Syntactical Properties of Residual Programs Can Reveal?

Alexei P. Lisitsa[1] and Andrei P. Nemytykh[2]*

[1] Department of Computer Science, The University of Liverpool
a.lisitsa@csc.liv.ac.uk
[2] Program Systems Institute of Russian Academy of Sciences
nemytykh@math.botik.ru

**Abstract**

The paper presents two examples of non-traditional uses of program specialization by Turchin's supercompilation method. In both cases we are interested in syntactical properties of residual programs produced by supercompilation. In the first example we apply supercompilation to a program encoding a word equation and as a result we obtain a program representing a graph describing the solution set of the word equation. The idea of the second example belongs to Alexandr V. Korlyukov. He considered an interpreter simulating the dynamics of the well known missionaries-cannibals puzzle. Supercompilation of the interpreter allows us to solve the puzzle. The interpreter may also be seen as an encoding of a non-deterministic protocol.
***Keywords*** program specialization, supercompilation, program analysis, program transformation, verification.

# 1 Introduction

One of the main concerns in the research in program specialization and other program transformation techniques is the efficiency of residual (or transformed) programs. It is widely known also that specialization of interpreters with respect to given programs can be used for effective changing of the semantics of the programs. For example, a number of researchers were interested in the generation of efficient programs implementing inverse functions [24], [25], [30], [23]. Still efficiency of the resulting programs is the major issue. But there is something more in program transformation techniques. They can be used for analysis of the programs and more specifically for their verification [11], [26], [4, 5], [6], [16], [8], [1], [22].

In this paper we take a step further and show that specialization can be used also not only for the analysis of the programs, but also for the solution of combinatorial and algebraic problems encoded in the programs. We consider a couple of examples illustrating use of a supercompiler (SCP4 [19],[20],[21]) for this purpose. We hope that these examples will be able to motivate further research in automated program specialization.

## 1.1 The Presentation Language

We present our program examples in a variant of a pseudocode for functional programs while real supercompilation experiments with the programs were done in a strict functional programming REFAL language [29], [31].

*The second author was partially supported by RFBR, research project No. 14-07-00133_a, and by Program No. 16 for Basic Research of Presidium of Russian Academy of Sciences.

The programs given below are written as *strict* term rewriting systems based on pattern matching. The sentences in the programs are ordered from the top to the bottom to be matched. To be more close to REFAL we use two kinds of variables: *s*.variables range over *symbols* (i.e. characters and identifiers, for example, `'a'` and `True`), while *e*.variables range over the whole set of the `S`-expressions. We also use a syntactical sugar for representation of words (finite sequences of characters), so, for example, the list `'b':'a':[]` is shortened as `'ba'` and `'aba':e.x` denotes `'a':'b':'a':e.x`. `<•,•,•>` denotes a triple. As usually the sign `++` stands for the standard operation `append`.

## 2 Word Equations

For a given alphabet $\Sigma$ we denote as $\Sigma^*$ the set of finite words over $\Sigma$. Let $X$ be a finite set of variables disjoint with $\Sigma$. A word expression is defined to be either an element of $\Sigma$, a variable, the empty word $\epsilon$ or a finite sequence of word expressions. For given two word expressions $L$ and $R$, an equation of the form $L = R$ is called a word equation.

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

A classical problem is to describe the set of all solutions of a given word equation. In a sense the equation itself describes the set. We are interested in a more constructive (transparent) description of the set. An important contribution in solving the problem was made in the late 1960's by Yu. I. Khmelevskiĭ [7]. In 1970's G. S. Makanin [18] suggested an algorithm deciding whether or not there exists a solution of any given word equation. There exists an algorithm based on Makanin's ideas, which for a given word equation generates a finite graph describing the corresponding solution set.

An example given below demonstrates both the statement above (i.e., a solution exists) and the fact that supercompilation is able to generate the *same* graph as Makanin's algorithm does for this example word equation. In this example the graph is a finite automaton.

```
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 function `equal` is a predicate checking whether two given words are equal. The function `main` is a predicate testing whether a given word is a solution of the following word equation:

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

We may specialize the program with respect the partial knowledge of the arguments of the call for the function `equal`. The residual program produced by the supercompiler SCP4 looks as follows:

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

This residual program `P` (as any program!) can be seen as a graph. Vertices of the graph correspond to the program function calls and the return expressions. Its edges are labeled with the case expressions or assignments. This graph describes the solution set of the word equation being considered. The edge corresponding to the last sentence of `P` is redundant for

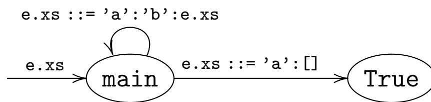


Figure 1: Graph describing the solution set of the word equation.

such description. This sentence will be absent, if we will remove the last sentence of the original function `equal`. After that the obtained graph *coincides* with the graph generated by Makanin's algorithm (see, for example, [3]). See Figure 1 for the graph. The solution set is `('ab')`$^*$`'a'`.

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;
```

The graph encoded by the residual program consists of the root and a leaf labeled with `False`. The only edge outcoming from the root leads in the leaf. That means the word equation

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

has no solutions.

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. It is certain, that for arbitrary word equations the supercompiler SCP4, in general, is not able to reproduce the *same* graph as Makanin's algorithm does. Weakness of the supercompiler as well as the supercompilation method *per se* does not allow to achieve such a result. In our opinion, the ideas underlying Makanin's algorithm may be borrowed for further development of the supercompilation method.

# 3 Supercompilation of an Absurd Program (A. V. Korlyukov's Example)

The idea of the example given in this section belongs to A. V. Korlyukov. Some of the comments to the example belong to A. V. Korlyukov as well, while another part is given by the authors of this paper.

Unfortunately, A. V. Korylukov passed away prematurely and published the idea only in Russian Internet pages [10],[9].

Following V. F. Turchin let us consider the well-known problem “Missionaries and Cannibals” [29].

*Three missionaries and three cannibals come to the bank of a river and see a boat. They want to cross the river. The boat, however, can carry no more than two people. There is a further restriction. At no time should the number of cannibals on either bank of the river (including the moored boat) exceed the number of missionaries (because the missionaries would then be overpowered and eaten). How (if at all) is it possible to cross the river?*

Now we generalize the problem. 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.

We intend to investigate the problem in details. We will use the supercompiler SCP4 as a "calculator" in an interactive mode (as usually any calculator being used).

We consider an interpreter of the dynamic system moving the crowd from the left bank to the right one. The interpreter is given in Figure 2. It takes (as an input) the pair of the numbers $n, k$ describing the initial crowd on the left bank and a finite sequence of the boat states (we call such a sequence a path). It returns information: does a prefix of the path bring the crowd to the right bank (i.e., `True` or `False`)? Additionally, if the answer is `True`, then it returns the rest of the path, which did not take a part in moving the crowd; if the answer is `False`, then it returns the part of the crowd brought to the right bank.

There are a number of tricks in the encoding. We encode the pair of the numbers $n, k$ as a triple of nonnegative integers $m, p, c$ such that

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

and we use unary notation to represent the integers. Thus $m$ is is the overweight of missionaries compared to cannibals, $c$ is the same vice versa; while $p$ is the number of those who are outnumbered, i.e. if there are more missionaries than cannibals, $p$ is the number of cannibals, and vice versa. In other words, p is the number of the pairs of the missionary-cannibal balancing one another. For example, five missionaries and three cannibals will be represented as `[['mm'],['ppp'],[]]`, while two missionaries and three cannibals will be represented as `[[],['pp'],['c']]`. A state of a given bank is encoded with such a triple. A state of the boat is encoded as one identifier, the name of which consists of the first capital letters of its passengers. For instance, `MM` means the state with two missionaries on the boat, while `C` is the state with one cannibal. The function `Move` has four arguments: the first one is a state of the boat, the second is an active bank (`L` or `R`), `e.l` and `e.r` are variables taking (as their values) the states of the left and right banks correspondingly. The `Move` modifies the banks' states and returns the new active bank and the two modified states. The functions `Minus` and `Plus` define unary arithmetic according to our encoding. The function `Int` iterates the crossing process. Here we skip over the functions `CutFalse, BlockRepetition` and will consider them later. As an example we give the following call for the interpreter and its result.

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

Notice that the two `MC` ending the path is not a misprint. The last `MC` does matter and, by the definition of `mainInt`, it is included in the result. It is a part of the input path, which is formally redundant. It is not used to solve the puzzle. This property of the interpreter is one of the tricks incidental to the supercompiler SCP4. The reason behind this property will be explained below (see Sect. 3.3).

At first glance the program looks a bit strange. It needs the whole history of its computations as the second argument to start transformations of the first argument (such a program is a kind of a predicate). But we are not going to execute the program, we will supercompile it. And we will show that does matter.

We now turn ourselves to the reasons motivating such an encoding, which allowed us to simplify the specialization task. We assume that the reader has the basic understanding of supercompilation concepts and some experience of using a supercompiler.

Speaking informally, the supercompiler trying to prove a statement encoded as a program unfolds a potentially infinite semantics tree of this program, improves/optimizes some properties of the tree, and folds the optimized tree back to a finite graph representing the result program. The nodes of the semantics tree are parameterized (i.e. partially unknown) states of the program `P` being transformed.

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

/* The boat on the left bank.  */
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)>;

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);

CutFalse( Deadlock ) = [];
BlockRepetition( Deadlock ) = [];

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

Figure 2: The interpreter.

In the simplest cases the unfolded semantics tree contains at most finite numbers of the parameterized states (let us call them basic states) such that any evaluation path of the tree, starting from the root – a parameterized entry state of P, can be folded to one the basic states. I.e. each evaluation path is either finite or includes a parameterized node $st_c$ such that there are a basic node $st_b$ and a substitution $\theta$ defined over the basic-node parameters and reducing $st_b$ to $st_c$: $\theta(st_b) = st_c$. Such folding can be seen as a proof, by induction, of the statement encoded by P, in which the basic nodes are the induction assumptions and each basic node is an auxiliary statement (a lemma). (See, for example, [13], [16] or recall Prolog.)

In a general case to create at most finite numbers of the basics nodes, as a rule, the supercompiler is forced to generalize some parameterized states being the nodes of the semantics tree; otherwise it will run unbounded time. Given two parameterized states $st_1$ and $st_2$, the generalization decides reasonable or not to generalize these states, and, if the corresponding decision is positive, it generates a new state $st_g$ and two substitutions $\theta_1$, $\theta_2$ such that $\theta_1(st_g) = st_1$, $\theta_2(st_g) = st_2$. The generalization, based on syntactical structures, replaces some of known data syntactically included in $st_1$ and/or $st_2$ with unknown data (parameters). As a consequence it generalizes the statements encoded by $st_1$, $st_2$ (i.e. it forget some information about these states). Then the supercompiler considers the generalized hypothesis encoded by $st_g$ as a possible basic node. The following problem can be encountered here: the hypothesis $st_g$ may be wrong and hence it cannot be proved, in spite of the fact that both $st_1$ and $st_2$ hold.

The more subtle generalization, the more chances to generate a correct generalized hypothesis; as well as the more structured syntax of $st_1$ and $st_2$, the more chances to generate a correct generalized hypothesis. Thus our encoding described above aims to separate the intentional notations explicitly. For example, the tricks above, encoding the pair $n, k$ as a triple of nonnegative integers $m, p, c$, makes the parts of the missionaries and cannibals balancing one another syntactically transparent. These parts encoded by $p$ are, in a sense, weakly relevant to the question formulated in the puzzle being considered, while $m$ and $c$ also separated one from the other do matter. Thus we give an helpful hint which may be used by the supercompiler.

Now let us return to the concrete encoding. First of all, let us formulate the problem in program terms. For the following 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 will answer the given question interactively using the supercompiler SCP4 and analyzing the residual programs produced by the supercompiler. By the precondition of the problem we can narrow the initial left state's set

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

to one of the following two forms

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

otherwise the missionaries will immediately be eaten. Thus we have the two tasks:

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

and

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

to be solved.

With the goal to make the residual programs more compact (so easier analyzable) we add an additional trick. Among all the paths, moving the boat from one bank to the other, there exist meaningless paths. The simplest of them contains at least two identical states staying side by side in the path: such two states mean the boat has just moved passengers to a bank immediately and moves the same kind of passengers back to the other bank. If there exists such a path successfully moving the crowd to the right bank, then there exits a shorter successful path, where the conjugated states are removed. This note allows us to exclude the indicated kind of paths (henceforth everywhere in this paper). We do that by calling the function `BlockRepetition` with an argument which never matches the function definition. So the call leads to an abnormal

stop at run time and to cut the corresponding evaluation path of the program at supercompile time.The first argument `s.pa` of the function `Int` serves to recognize two identical states staying side by side in a given path.

Whenever we hope for a positive answer to the problem question we will replace the following sentence of the program to be supercompiled

```
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([]);
```

Once again we do that for the sake of simplicity of the corresponding residual programs. The call `CutFalse([])` like the call for the function `BlockRepetition` cuts the branch leading to the negative answer.

## 3.1 Diagonal Cases

We start with the case when the number of the missionaries equals to the number of the cannibals. That is to say, we have to supercompile the following start configuration:

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

Our first experiment is for two missionaries and two cannibals.

**Example 1.** *The start configuration:*

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

*The program contains the sentence with the call CutFalse([]). 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 length of the shortest path moving the crowd to the right bank is 5. There exist four such paths:*

```
[CC, C, MM, M, MC]
[CC, C, MM, C, CC]
[MC, M, MM, M, MC]
[MC, M, MM, C, CC]
```

*Moreover, the residual program specifies the whole set of the successful paths.*

We leave to the reader to analyze the residual programs for one missionary and one cannibal (Example 2) and for three missionaries and three cannibals (Example 3). The source programs contain the sentence with the call `CutFalse([])`. The answers to the question are positive in both cases.

**Example 2.**

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

**Example 3.**

```
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 following experiment is for an equal number of missionaries and cannibals, and the number is greater than three.

**Example 4.** *The start configuration:*

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

*The residual program for the source program containing the sentence with `False` is given in the attachment A. The residual program never returns `True` and that is a* syntactical *property of the program. On the other hand, the source program terminates for* any given *arguments. That allows us to conclude that there exist no paths moving the crowd to the right bank.*

*Note that if we supercompile the source program containing the sentence with the call `CutFalse([])`, then the residual program is* empty. *In other words, it is a trivial program with the empty domain and this property also is syntactical rather than semantic. We might infer the negative answer to the problem question from the emptiness of the residual program.*

Thus we have considered all the diagonal cases. If the number of both missionaries and cannibals is greater than three, then there exist no paths moving the crowd to the right bank, otherwise such a path exists.

## 3.2 The Number of Missionaries is Greater Than the Number of Cannibals

Now we consider the case when the number of the missionaries greater than the number of the cannibals. The corresponding start configuration is:

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

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

**Example 5.** *The start configuration is*

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

*The supercompiler SCP4 produces quite a large residual program. The program may be found in [17]. Unfortunately we have to analyze the residual program, because the numbers of both missionaries and cannibals are not given in advance, and for each pair may exists a needed path and such paths may have (and do have) very different structures. Moreover, 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 (a deadlock) of the program).*

On the other hand, such 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 try to narrow the paths' set, where we are looking for the successful path's witnesses. With such an aim we restrict the boat states as follows.

**Example 6.** *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 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. We do that by changing 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)>;

/* 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)>;
```

*This function filters out the forbidden states of the boat: they never match with the sentences of the redefined function `Move` and they lead to an abnormal stop (recognition impossible, deadlock).*

*Now, supercompiling the start configuration*

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

*we obtain the following 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];
```

*Analyzing this result program we see that if the boat is on the left bank, then the path `[MC,C,MC,M]` adds a pair (missionary-cannibal) on the right bank, provided that on the left bank the number of the missionaries greater than the number of the cannibals. That means a repeated iteration of such a path leads to success. The path `[MC,C,MC]` (a prefix of that considered above) decides the problem with two missionaries and one cannibal.*

*Note that such a narrowing of the possible paths might be unsuccessful for other start configurations. For example, in the case of the start configuration*

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

*there exists no successful path from the narrowed set, while we have found a successful path in the whole paths' set (see Example 3).*

*Nevertheless supercompilation of the start configuration*

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

*with the original definition of the function* `Move` *produces a residual program* `P` *(see the attachment [17]), which is more complicated than the residual program above, but the key information of the path* `[MC,C,MC,M]` *still may be retrieved from the* `P` *(by applying more efforts).*

The other cases (when in the start configurations the number of the cannibals is greater than the number of the missionaries, no cannibals, no missionaries) are trivial. The following table summarizes our investigation. It gives a left upper angle of the result matrix. The empty cells correspond to the `False` answer.

| 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 | | | | | | | |

### 3.3 On a Property of the Supercompiler SCP4

The reader may note that these authors do not used an evident simplified encoding of the interpreter used as a predicate. I.e. a syntactically pure predicate, which returns either `True` or `False`. The interpreter `Int` given in Figure 2 and used in our experiments returns some additional information concatenated to these logical constants (see the first and second sentences of the definition of `Int`). From logical point of view, this additional information is meaningless. The reason why we use such a trick is as follows.

The supercompiler SCP4 is an optimizer. Given an input program `P(x,y)`, by definition, SCP4 is allowed to extend domain of `P(x,y)`. Assume that `P(x,y)` may return only `True` or `False`. Suppose $\mathtt{P(x_0,y)}$ is a specialization task to be optimized by SCP4. I.e. the input data are partially known: $\mathtt{x_0}$ is fixed. Assume for each `y` the call $\mathtt{P(x_0,y)}$ returns `True`. I.e. $\mathtt{P(x_0,y)}$ is a partial constant function. SCP4 uses a tool, which is sometimes able to recognize partial constant functions, even if their definitions include some recursions. Assume SCP4 is able to recognize that $\mathtt{P(x_0,y)}$ may return the only constant `True`. In such a case the result program `Q(y)` generated by SCP4 contains the only following sentence `Q(y) = True`. Notice that for each `y` from the domain of $\mathtt{P(x_0,y)}$, $\mathtt{Q(y) = P(x_0,y)}$ holds and the residual program `Q(y)` defines an extension of the original function defined by $\mathtt{P(x_0,y)}$. Thus, by definition, *the supercompiler SCP4 generated a legal result program being the most efficient program*, which can be constructed in the given context of the definition.

That is exactly what the supercompiler SCP4 will produce if the definition of `Int` will be simplified as discussed above. But such a trivial residual program does say nothing about the structure of its unknown input `path`, which we are trying to investigate!

# 4 Conclusion

What did happen in the experiments described above? Program specialization (supercompilation) originally developed as an optimization method was used for studying some *syntactical* properties of the residual programs rather than for executing the programs. In a sense we used the supercompiler SCP4 as a tool helping to solve the mathematical tasks. We used the tool in an interactive fashion. In such a way we use any calculator.

In the section 2 we were interested in the graph encoded within the residual program. The graph describes the solution set of a word equation.

Using the supercompiler SCP4 to solve the missionaries-cannibals problem was more complicated. In this case, using a rough analogy, the supercompiler was exploited as a kind of a "PROLOG interpreter" (see also [27]). We were interested in narrowing a parameter of the start configurations (the *goals* to be supercompiled) (`mainInt([[e.m],[e.p],[e.c]], e.path)`). The narrowed parameter `e.path` (its *substitution set*) has to satisfy a property imposed by the source program (actually a kind of a predicate). Unlike PROLOG, when the substitution set cannot be described as a finite union of parameterized lists (S-expressions), the supercompiler produces a finite residual program describing the substitution set more explicitly (transparently) as compared with the source program. (The same note is valid for the word equation example.) The function calls `CutFalse([]), BlockRepetition([])` are analogues of the CUT mechanism in PROLOG.

We have to note that first examples of theorem proving by supercompilation were given by V. F. Turchin [28]. There are ROLOG-programs that solve this missionaries-cannibals puzzle puzzle too. For example S. Antoy and M. Hanus [2] propose to use a constraint logic program, where both the complexity of the operation move and the creation of invalid states are avoided by a constrained constructor pattern.

The function `mainInt(e.l,e.path)` can be seen as an interpreter transforming data $\texttt{e.l}_0$ according to a given program $\texttt{e.path}_0$. In such a case our experiments can be seen as specialization of the interpreter with respect to partial information on the data. The authors used such specialization of interpreters for verification of *safety* properties of nondeterministic cache coherence protocols [13], [16], [12], [14] and a cryptographic protocol [1]. The dynamics system "missionaries-cannibals" also may be considered as a nondeterministic protocol, where the boat states are actions controlling the global state of the computing system. The actions are being chosen in a nondeterministic way governed by a number of guides. The examples above of the start crowd configurations, for which we proved the negative answer on the question given in the problem, are the conditions on the start protocol configurations satisfying the safety property defined by the positive answer on the question. That is to say, the computer system starting with such a configuration never reaches the global state, when the whole crowd is on the right bank. Our treatment of the start configurations corresponding to the positive answer is a search for a witness violating the safety property. Acting in a similar way we have found bugs in a number of cache coherence protocols (see [15], [14]).

Finally we would like to note that actually all our experiments were done in a strict functional programming language REFAL [29], [31], using the supercompiler SCP4 under the following strategy:

```
$MATCHING ForRepeatedSpecialization;
$STRATEGY Applicative;
```

## Acknowlegements

We would like to give our deepest thanks to the reviewers who were generous in their constructive comments. We have tried to follow some of their recommendations. The remaining recommendations stimulate us to develope the ideas demonstrated in this paper and to describe them in more formal terms in the future. We accept responsibility for all remaining errors.


## References


[1] A. Ahmed, A. P. Lisitsa, and A. P. Nemytykh. Cryptographic protocol verification via supercompilation (a case study). In *VPT 2013. First International Workshop on Verification and Program Transformation*, volume 16 of *EPiC Series*, pages 16–29, 2013.

[2] S. Antoy and M. Hanus. Functional logic design patterns. In *the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, volume 2441 of *LNCS*, pages 67–87. Springer, 2002.

[3] V. Diekert. Makanin's algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, Chapter 12, pages 387–442. Cambridge University Press, 2002.

[4] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying ctl properties of infinite state systems by specializing constraint logic programs. In *the Proc. of VCL01*, volume DSSE-TR-2001-3 of *Tech. Rep.*, pages 85–96, UK, 2001. University of Southampton.

[5] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems: An experimental evaluation. In M. Alpuente, editor, *the Proc. of LOPSTR 2010*, volume 6564 of *LNCS*, pages 164–183. Springer, 2011.

[6] G. W. Hamilton. Distilling programs for verification. *Electronic Notes in Theoretical Computer Science*, 190(4):17–32, 2007. The Proc. of the International Conference on Compiler Optimization Meets Compiler Verification.

[7] Yu. I. Khmelevskiĭ. Equations in free semigroups. (in Russian). In I. G. Petrovskiĭ, editor, *Trudy Math. Inst. Steklov*, volume 107, 1971. English translation in: Proc. of Steklov Inst. Math., 107, Amer. Math. Soc., 1976.

[8] A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In *the Proc. of PSI'11*, volume 7162 of *LNCS*, pages 193–209, 2012.

[9] A. V. Korlyukov. User manual on the supercompiler SCP4. (in Russian). [online], 1999. `http://www.refal.net/supercom.htm`.

[10] A. V. Korlyukov. Missionaries and cannibals. (in Russian). [online], 2001. URL `http://www.refal.org/~korlukov/pearls/misscann/misscann.htm`.

[11] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *LNCS*, pages 63–82, Venice, Italy, 2000.

[12] A. P. Lisitsa and A. P. Nemytykh. A note on specialization of interpreters. In *The 2-nd International Symposium on Computer Science in Russia (CSR-2007)*, volume 4649 of *LNCS*, pages 237–248, 2007.

[13] A. P. Lisitsa and A. P. Nemytykh. Verification as parameterized testing (Experiments with the SCP4 supercompiler). *Programmirovanie, (In Russian)*, 1:22–34, 2007. English translation in *J. Programming and Computer Software*, Vol. **33**, No.1, pp: 14–23, 2007.

[14] A. P. Lisitsa and A. P. Nemytykh. Experiments on verification via supercompilation. [online], 2007–2009. `http://refal.botik.ru/protocols/`.

[15] A. P. Lisitsa and A. P. Nemytykh. Extracting bugs from the failed proofs in verification via supercompilation. In Bernhard Beckert and Reiner Hahnle, editors, *Tests and Proofs: Papers*

*Presented at the Second International Conference TAP 2008*, number 5/2008 in Reports of the Faculty of Informatics, Univesitat Koblenz-Landau, pages 49–65, April 2008.

[16] A. P. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *International Journal of Foundations of Computer Science*, 19(4):953–970, August 2008.

[17] A. P. Lisitsa and A. P. Nemytykh. Residual programs in the experiments with the "Missionaries and Cannibals" puzzle. [online], 2009. `ftp://www.botik.ru/pub/local/scp/refal5/experiments_with_missionaries-cannibals_puzzle.zip`.

[18] G. S. Makanin. The problem of solvability of equations in a free semigroup. (in Russian). *Matematicheskii Sbornik*, 103(2):147–236, 1977. English translation in: Math. USSR-Sb., 32, pp: 129–198, 1977.

[19] A. P. Nemytykh. The supercompiler Scp4: General structure. (extended abstract). In *the Proc. of PSI'03*, volume 2890 of *LNCS*, pages 162–170, 2003.

[20] A. P. Nemytykh. *The Supercompiler SCP4: General Structure*. URSS, Moscow, 2007. (Book in Russian).

[21] A. P. Nemytykh and V. F. Turchin. The supercompiler Scp4: Sources, on-line demonstration. [online], 2000. `http://www.botik.ru/pub/local/scp/refal5/`.

[22] Antonina Nepeivoda. Ping-pong protocols as prefix grammars and Turchin relation. In *VPT 2013. First International Workshop on Verification and Program Transformation*, volume 16 of *EPiC Series*, pages 74–87, 2013.

[23] Yo. Kawada R. Glück and T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. *Proc. of the ACM SIGPLAN symposium on principles and practice of parallel programming and workshop on partial evaluation and semantics-based program manipulation, ACM SIGPLAN Notices*, 38(10):10–19, 2003.

[24] A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Proc. of Partial Evaluation and Mixed Computation*, pages 486–479, North-Holland, 1988.

[25] A. Y. Romanenko. The generation of inverse functions in Refal. volume 26 of *ACM SIGPLAN Notices*, pages 12–22, Sept. 1991.

[26] Abhik Roychoudhury and C. R. Ramakrishnan. Unfold/fold transformations for automated verification of parameterized concurrent systems. In *Program Development in Computational Logic*, pages 261–290, 2004.

[27] M. H. Sørensen and R. Glück. Partial deduction and driving are equivalent. In *Programming Language Implementation and Logic Programming*, volume 844 of *LNCS*, pages 165–181, 1994.

[28] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, volume 85 of *LNCS*, pages 645–657. Springer-Verlag, 1980.

[29] V. F. Turchin. *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts, 1989. Electronic version: `http://www.botik.ru/pub/local/scp/refal5/`, 2000.

[30] V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[31] V. F. Turchin, D. V. Turchin, A. P. Konyshev, and A. P. Nemytykh. Refal-5: Sources, executable modules. [online], 2000. `http://www.botik.ru/pub/local/scp/refal5/`.

# A The Residual Program Obtained in Example 4

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

f(CC, [], e.p) = <[], ['cc']> ;
f(CC, [C], e.p ) = <[],['c']> ;
f(CC, [C, CC], e.p) = <[],['ccc']> ;
f(CC, [C, CC, C], e.p) = <[],['cc']> ;
f(CC, [C, CC, C, MM], e.p) = <['pp'],[]> ;
f(CC, [C, CC, C, MM ,MC], e.p) = <['p'],[]> ;
f(CC, [C, CC, C, CC, e.path], e.p) = <[], ['ccc', g(e.p, [], e.path)]>;
f(CC, [C, M], e.p) = <['p'],[]> ;
f(CC, [C, M, MC], e.p) = <[],[]> ;
f(CC, [C, M, MC, s.x, e.path], e.p) = f(s.x,e.path,e.p);

f(MC, [], e.p) = <['p'],[]> ;
f(MC, [M], e.p) = <[], ['c']> ;
f(MC, [M, CC], e.p) = <[], ['ccc']> ;
f(MC, [M, CC, C], e.p) = <[], ['cc']> ;
f(MC, [M, CC, C, MM], e.p) = <['pp'], []> ;
f(MC, [M, CC, C, MM, MC], e.p) = <['p'], []> ;
f(MC, [M, CC, C, CC, e.path], e.p) = <[], ['ccc', g(e.p, [], e.path)]>;
f(MC, [M, C], e.p) = <[], ['cc']> ;
f(MC, [M, C, CC], e.p) = <[], []> ;
f(MC, [M, C, CC, s.x, e.path], e.p) = f(s.x,e.path,e.p);

f(C, [], e.p) = <[], ['c']> ;


g(e.p, e.c, []) = 'c':e.c;
g(e.p, e.c, C) = e.c;
g('p':e.p, e.c, [C, CC, e.path]) = g(e.p, 'c':e.c, e.path);
```