Finite Countermodel Based Verification for Program Transformation

(A Case Study)

Alexei P. Lisitsa
Department of Computer Science,
The University of Liverpool
a.lisitsa@csc.liv.ac.uk

Andrei P. Nemytykh
Program Systems Institute,
Russian Academy of Sciences*
nemytykh@math.botik.ru

Both automatic program verification and program transformation are based on program analysis. In the past decade a number of approaches using various automatic general-purpose program transformation techniques (partial deduction, specialization, supercompilation) for verification of unreachability properties of computing systems were introduced and demonstrated [25, 8, 31]. On the other hand, the semantics based unfold-fold program transformation methods pose themselves diverse kind of (un)reachability tasks and try to solve them, aiming to improve the semantics tree of the program being transformed. That means some general-purpose verification methods may be used for strengthening program transformation techniques. This paper considers the question how finite countermodels for safety verification method [29] might be used in Turchin's supercompilation method. We extract a number of supercompilation sub-algorithms trying to solve (un)reachability problems and demonstrate use of an external countermodel finder for solving some of the problems.
Keywords: program specialization, supercompilation, program analysis, program transformation, safety verification, finite countermodels

1 Introduction

A variety of semantic based program transformation techniques commonly called specialization aim to improve some properties of the program `P` being transformed in a given context of using the program `P`. The transformation techniques save the partial function defined by `P` on the domain of `P`. Given a cost model the specialization aim may be optimization of `P`. Specialization also can be used for verification of some program properties [25, 8, 31]. The residual program (the specialization result) may have simple explicit *syntactic* properties, which are not evident in the original program `P`. I.e., in essence, they are semantic properties of `P` and specialization shifts them on the syntactic level. Such *syntactic* properties of the residual programs may be treated in different ways.

For example, let two programming language $\mathscr{L}_1$ and $\mathscr{L}_2$ be given. Let us suppose for simplicity that the data sets of the two languages coincide. Let `D` denote the common data set. Let `Int(p,d)` be an $\mathscr{L}_1$-interpreter written in $\mathscr{L}_2$ and let we be given a specializer transforming the programs written in $\mathscr{L}_2$ into the same language. Here `p` is an $\mathscr{L}_1$-program to be computed on the input data `d` $\in$ `D`. That is `Int(p,d)` $=$ `p(d)`. Given a fixed program $\mathtt{p}_0(\cdot)$ one may specialize the interpreter with respect to its first argument: $\mathtt{Int}(\mathtt{p}_0,\mathtt{d})$. Let $\mathtt{q}(\cdot)$ be the result of the specialization. Then $\mathtt{q}(\cdot) \in \mathscr{L}_2$ and, by definition of specialization, for any `d` $\in$ `D` the following equalities $\mathtt{q(d)} = \mathtt{Int}(\mathtt{p}_0,\mathtt{d}) = \mathtt{p}_0\mathtt{(d)}$ hold. Formally, those mean $\mathtt{q}(\cdot)$ can be seen as a result of compilation (from $\mathscr{L}_1$ into $\mathscr{L}_2$) of the program $\mathtt{p}_0(\cdot)$. Potential issue with the residual program is it could be not optimal. Speaking informally, to

*The second author was supported by RFBR, research project No. 14-07-00133_a.

To appear in EPTCS.

guarantee some optimality one may use the following syntactic property of the residual program: `q(·)` contains no the key-words of the $\mathscr{L}_1$-language. Those are all $\mathscr{L}_1$-terms excluding the terms defining data. The construction above is well-known and called the first Futamura projection [11, 7, 47].

It has been known for a while [48, 49, 24, 8, 10] that program transformation techniques, in particular program specialization, can be used to prove some properties of programs automatically. For example, if a program actually implements (in a given context of use) a constant function, sufficiently powerful and semantics preserving program transformation may reduce the program to a syntactically trivial "constant" program, pruning away unreachable branches and proving thereby the property. Viability of such an approach to verification has been demonstrated in previous works [30, 31] where it was applied to the verification of program models of parameterized cache coherence protocols and Petri Nets models [21, 32]. Later the program functional modeling was addressed to several cryptographic protocols and supercompilation succeeded to verify the protocol program models and even more a supercompiler was used for an interactive search for attacks on some cryptographic protocols [1, 42].

Specialization can be used for analysis of other program properties, basing on syntactic properties of the corresponding residual program. For example, any program can be seen as a graph and for some reasons one may try to apply specialization for improving some given properties of the graph (see [33]).

It is clear that any automated program analysis somehow may be applied to program verification. On the other hand, many program analysis tasks, in particular those appearing in the context of program transformation techniques can be considered as verification problems.

In this paper we address the question of applications of a particular verification technique, *finite countermodel based verification (FCM)* [44, 16, 28, 29] within the context of *supercompilation* particular program transformation method. Supercompilation is a semantic based specialization method introduced by V. F. Turchin [49] and which utilizes in particular unfold-fold based program transformation.

One of the main challenges in unfold-fold based program transformation techniques is to construct a good enough approximation of the *semantic* tree of the program `P` being transformed. That is to detect as many unreachable paths in the syntactic tree of `P` as possible and to prune away the dead paths. This syntactic tree is being stepwise unfolded from the program `P` definition.[1] Thus, in essence, this challenge is an (un)reachability problem. Supercompilation also poses itself other kind of (un)reachability tasks and tries to solve them (see Sec. 5 for examples).

Finite countermodels method (FCM) for safety verification utilizes the principle *reachability as derivability in the first-order logic* and reduces the task of safety verification, generally for infinite state and parameterized systems, to the tasks of *disproving* first-order formulae, which are tackled then by available finite model finders. Proposed initially for the verification of cryptographic protocols in [44, 16] it has been later expanded to the larger classes of infinite state and parameterised verification problems, including safety for general term-rewriting systems [28]. The method has been shown to be *relatively complete* with respect to more widely known methods of *regular model checking* [29] and *regular tree model checking* [26] and generally it works when the safety can be demonstrated using *regular* invariants.

What makes the combination of supercompilation and the FCM method interesting and promising it is their somewhat complimentary features. On the one side the analysis mechanisms exploited within supercompilation do not cover *all* regular properties of parameterized program configurations, so any help from the theoretically (relatively) complete the FCM method could potentially be useful. On the other hand, supercompilation (as well as other specialization methods) sometimes is able to recognize and use non-regular properties of the program `P` (see Sec. 6 for examples) going beyond of what is

[1] I.e. given a loop (or recursion definition), a one-step *syntactic* unfolding is a formal symbolic one-step unwinding the loop without any improvement of the copy of the loop body definition taken out of the loop iterations following the taken iteration.

possible by the FCM method.

So in short, the main idea of this paper is to explore the combination of FCM, theoretically complete for verification by regular invariants, with the mechanisms of supercompilation able to recognize and deal with non-regular properties.

It is known that the problem of constructing a good approximation of the semantics tree of a program is especially difficult if *strings* are among data sets involved in the program. Associativity of string concatenation causes that. Unfolding for such programs connects program transformation with solving word equations. The decidability of the corresponding satisfiability problem was demonstrated by G. S. Makanin [34] in 1977, but still solving word equations remains nontrivial task from a practical point of view. Strings form a fundamental data type used in most programming languages. Recently, a number of techniques producing string constraints have been suggested for automatic testing [15, 2] and program verification [45]. String-constraint solvers are used in many testing and analysis tools [5, 43]. This attracts interest in string manipulation in the context of program analysis and transformation and motivates our choice of the presentation language below.

This paper assumes the reader has basic knowledge of concepts of functional programming, pattern matching, term rewriting systems. We also assume that the reader is familiar with the basics of the first-order logic. In particular, we use without definitions the following concepts: the first-order predicate logic, first-order models, interpretations of relational, functional and constant symbols, satisfaction $\models$ of a formula in a model, semantic consequence $\models$, deducibility (derivability) $\vdash$ in the first-order logic. We denote interpretations by square brackets, so, for example, $[f]$ denotes an interpretation of a functional symbol f in a model.

2 Preliminaries

2.1 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 [50].

The programs given below are written as *strict* (call by value) term rewriting systems based on pattern matching. The sentences in the programs are ordered from top to bottom to be matched.

The data set is a free monoid of concatenation (i.e. the concatenation is associative) with an additional unary constructor, which is denoted only with its parenthesis (i.e. without a name). The colon sign stands for the concatenation. The constant ε is the unit of the concatenation and may be omitted, other constants c are characters. The monoid $\mathscr{D}$ of the data may be defined with the following grammar:
$\mathscr{D} \ni d$ `::=` ε `|` c `|` $d_1 : d_2$ `|` `(`d`)`
Thus a datum is a finite sequence (including the empty sequence), which can be seen as a forest of *arbitrary* finite trees.

Let $\mathscr{F} = \cup_i \mathscr{F}_i$ be a finite set of functional symbols, where $\mathscr{F}_i$ is a set of functional symbols of arity i. Let v, f denote a variable and a function name correspondingly, then the monoid of the corresponding terms may be defined as follows:
t `::=` ε `|` c `|` v `|` f`(` *args* `)` `|` $t_1 : t_2$ `|` `(`t`)`
args `::=` t `|` t`,` *args* – where the number of the arguments of f equals its arity.

Let the denumerable variable set $\mathscr{V}$ be disjoined in three sets $\mathscr{V} = \mathscr{E} \cup \mathscr{S} \cup \mathscr{T}$, where the names from $\mathscr{E}$ are prefixed with `'e.'`, while the names from $\mathscr{S}$ – with `'s.'` and the names from $\mathscr{T}$ – with `'t.'`. `s.`variables range over characters, `e.`variables range over the whole data set, while a `t.`variable

can take as value any character or any data surrounded by parentheses. For a term t we denote the set of all e-variables (t,s-variables) in t by $\mathscr{E}(t)$ (correspondingly $\mathscr{T}(t)$, $\mathscr{S}(t)$). $\mathscr{V}(t) = \mathscr{E}(t) \cup \mathscr{S}(t) \cup \mathscr{T}(t)$.

Examples of the variables are `s.r, t.F1, e.cls, e.memory`. I.e. a variable name may be any identifier. We also use a syntactical sugar for representation of words (finite sequences of characters), so, for example, the list `'b':'a':`ε is shortened as `'ba'` and `'aba':e.x` denotes `'a':'b':'a':e.x`.

Let $\mathscr{C}$ denote a set of the characters. We denote the monoid of terms by $\mathscr{T}(\mathscr{C},\mathscr{V},\mathscr{F})$. A term without function names is passive. We denote the set of all passive terms by $\mathscr{P}(\mathscr{C},\mathscr{V})$. Let $\mathscr{G}(\mathscr{C},\mathscr{F}) \subset \mathscr{T}(\mathscr{C},\mathscr{V},\mathscr{F})$ be the set of ground terms, i.e. terms without variables. Let $\mathscr{O}(\mathscr{C}) \subset \mathscr{G}(\mathscr{C},\mathscr{F})$ be the set of object terms, i.e. ground passive terms. Given a subset of the variables $\mathscr{V}_1 = \mathscr{S}_1 \cup \mathscr{T}_1 \cup \mathscr{E}_1$ where $\mathscr{S}_1 \subseteq \mathscr{S}$, $\mathscr{T}_1 \subseteq \mathscr{T}$, $\mathscr{E}_1 \subseteq \mathscr{E}$, a substitution is a mapping $\theta : \mathscr{V}_1 \to \mathscr{T}(\mathscr{C},\mathscr{V},\mathscr{F})$ such that $\theta(\mathscr{S}_1) \subseteq \mathscr{C} \cup \mathscr{S}$ and $\theta(\mathscr{T}_1) \subseteq \mathscr{C} \cup \mathscr{S} \cup \mathscr{T} \cup \{(t) \mid t \in \mathscr{T}(\mathscr{C},\mathscr{V},\mathscr{F})\}$. A substitution can be extended to act on all terms homomorphically. A substitution is called *object* iff its range is a subset of $\mathscr{O}(\mathscr{T})$. We use notation $s = t\theta$ for $s = \theta(t)$, call s an *instance* of t and denote this fact by $s \ll t$.

A program P is a pair $\langle \tau, R \rangle$, where τ is a term called *initial* and R is a finite set of rules of the form $f(p_1,\ldots,p_k) = r$, where $f \in \mathscr{F}_k$, for each $(1 \le i \le k)$, p_i is a passive term, r is a term containing the function names only from R, $\mathscr{V}(r) \subseteq \mathscr{V}(f(p_1,\ldots,p_k))$.

A program $P = \langle \tau, R \rangle$ with $R = \{l_i = r_i \mid i = 1 \ldots n\}$ gives rise to a reachability relation $\to_P$ on ground terms $\to_P$ as follows. For ground t_1 and t_2 the term t_2 is one-step P-reachable from t_1 if and only if there exists a substitution $\theta : \mathscr{V}(t) \to \mathscr{D}$ such that 1) there exists $i : 1 \le i \le n$ such that for all $j \in \mathbb{N}$, $1 \le j < i$, $t_1\theta$ does not match against l_j and it matches against l_i, and 2) $t_2 = r_i\theta$. In words, t_2 is obtained from t_1 by application of *the first, in a given order* applicable rewriting rule from R.

We denote by $\Rightarrow_P$ a one-step reachability defined similarly to $\to_P$ above, but omitting the clause "for all $j \in \mathbb{N}$, $1 \le j < i$, $t_1\theta$ does not match against l_j". Thus $t_1 \Rightarrow_P t_2$ iff t_2 is obtained from t_1 by application of any rule from R. It is obvious that $\Rightarrow_P$ is an *overapproximation* of $\to_P$, that is $\to_P \subseteq \Rightarrow_P$. We denote *reflexive and transitive closure* of $\to_P$ and $\Rightarrow_P$ by $\to_P^*$ and $\Rightarrow_P^*$, respectively.

2.2 Examples

The infinite sequence `Fib` of Fibonacci words is defined recursively as

$$w_0 = b; w_1 = a; w_{i+2} = w_i w_{i+1};$$

and consists of the words: $b, a, ba, aba, baaba, ababaaba, baabaababaaba, \ldots$

Example 1 *The program $\langle \tau, R \rangle$,where τ is `Fib(e.n)` and R given below, computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument (the `e.n` variable) in the unary notation. The result words are separated by the parenthesis constructors rather than the comma sign. For example, `Fib('III') = ('aba') : ('baaba')`. Note that the right side of the last rule uses associativity of the concatenation: the length value of `e.xs` is unknown, e.g. it may be grater than* 1.

```
Fib(e.n) = F(e.n, 'b', 'a');
F(ε, e.xs, e.ys) = (e.xs) : (e.ys);
F('I' : e.ns, e.xs, e.ys) = F(e.ns, e.ys, e.xs : e.ys);
```

The following example demonstrates using the associative constructor in the patterns (left-sides) of the first and second rules. The example program defines a predicate testing: (1) whether or not a given Fibonacci word postfix is `'bb'`; (2) given two consecutive Fibonacci words, can the first of them end with `'b'`, while the second start with `'b'`? In the positive case the predicate value is `'F'`, otherwise – `'T'`.

Example 2 *τ is `B(Fib(e.n))` and R is from the previous example together with:*

```
B( (e.xs :'bb'):(e.ys) ) = 'F';
B( (e.xs :'b'):('b' : e.ys) ) = 'F';
B( (e.xs):(e.ys) ) = 'T';
```

2.2.1 On Semantics of the Pattern Matching

Associativity of the concatenation creates an issue with the pattern matching, namely, given a term τ and a rule $(l = r) \in R$, then there can be several substitutions matching τ against l. Thus we here have a kind of non-determinism. An example is as follows:

Example 3 *$\tau =$ `f('abcabc', 'bc')` and $l =$ `f(e.x : e.w : e.y, e.w)`. There exist two substitutions matching these terms: the first one is $\theta_1(e.x) =$ `'a'`, $\theta_1(e.w) =$ `'bc'`, $\theta_1(e.y) =$ `'abc'`, the second is $\theta_2(e.x) =$ `'abca'`, $\theta_2(e.w) =$ `'bc'`, $\theta_2(e.y) = \varepsilon$.*

To make the pattern matching unambiguous in the language $\mathscr{L}$, we take the following decision arising from Markov's normal algorithms [35] and used in Refal [50]: (1) if there is more than one way of assigning values to the variables in the left-side of a rule in order to achieve matching, then the one is chosen in which the leftmost e-variable takes the shortest value; (2) if such a choice still gives more than one substitution, then the chosen e-variable shortest value is fixed and the case (1) is applied to the leftmost e-variable from the e-variables excluding considered ones, and so on while the whole list of the e-variables in the left-side of the rule is not exhausted.

In the sequel we refer to this rule as the Markov's rule and such a substitution as the Markov's substitution on l, matching τ.

Example 4 *$\tau =$ `f('abacad')` and $l =$ `f(e.x : 'a' : e.y : 'a' : e.z)`. There exist three substitutions matching the terms: the first is $\theta_1(e.x) = \varepsilon, \theta_1(e.y) =$ `'b'`, $\theta_1(e.z) =$ `'cad'`; the second is $\theta_2(e.x) = \varepsilon, \theta_2(e.y) =$ `'bac'`, $\theta_2(e.z) =$ `'d'`; the third is $\theta_3(e.x) =$ `'ab'`, $\theta_3(e.y) =$ `'c'`, $\theta_3(e.z) =$ `'d'`.*

The leftmost e-variable is e.x. Both in the first and the second substitutions the length of the e.x's values is zero. The next leftmost e-variable is e.y and $ln($`'b'`$) < ln($`'bac'`$)$. The first substitution meets Markov's rule.

Given a term of the form $\mathtt{f}(t_1,\ldots,t_n)$ where for all $(1 \le i \le n)$, $t_i \in \mathscr{O}(\mathscr{C})$ and a term $\mathtt{f}(p_1,\ldots,p_n)$ where all p_i are passive terms, the matching $\mathtt{f}(t_1,\ldots,t_n)$ against $\mathtt{f}(p_1,\ldots,p_n)$ can be viewed as solving the following system of equations in the free monoid of the object terms $\mathscr{O}(\mathscr{C})$.

$$\begin{cases} p_1 &= t_1 \\ & \ldots \\ p_n &= t_n \end{cases}$$

We look for all values of the variables (i.e. substitutions θ_i) from $\mathscr{V}(\mathtt{f}(p_1,\ldots,p_n))$ such that for each i and each $(1 \le j \le n)$, $\theta_i(p_j) = t_j$ and if the values' set is not empty we choose the only Markov's substitution, where Markov's rule acts on the pattern $(p_1)\ldots(p_n)$. Note the patterns p_j may share variables and this equation system is equivalent to the following only equation $(p_1)\ldots(p_n) = (t_1)\ldots(t_n)$. This system has an important property: for all $(1 \le i \le n)$ $t_i \in \mathscr{O}(\mathscr{C})$. There is a simple algorithm solving the equation systems meeting this property.

2.3 On Supercompilation

In this paper we are interested in one particular approach in program transformation and specialization, known as supercompilation[2]. Supercompilation is a powerful semantic based program transformation technique [49, 46] having a long history well back to the 1960-70s, when it was proposed by V. Turchin. The main idea behind a supercompiler is to observe the behavior of a functional program p running on *partially* defined input with the aim to define a program, which would be equivalent to the original one (on the domain of latter), but having improved properties. The supercompiler unfolds a potentially infinite tree of all possible computations of a parameterized program. In the process, it reduces the redundancy that could be present in the original program. It folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of the computing system. And, finally, it analyses global properties of the graph and specializes this graph with respect to these properties (without an additional unfolding). The resulting program definition is constructed solely based on the meta-interpretation of the source program rather than by a (step-by-step) transformation of the program.

The result of supercompilation may be a specialized version of the original program, taking into account the properties of partially known arguments, or just a re-formulated, and sometimes more efficient, equivalent program (on the domain of the original).

Turchin's ideas have been studied by a number of authors for a long time and have, to some extent, been brought to the algorithmic and implementation stage [41]. From the very beginning the development of supercompilation has been conducted mainly in the context of the Refal programming language [50, 40, 38, 39] basing on syntax and semantics being similar to our presentation language syntax and semantics. A number of the simplest model supercompilers for subsets of functional languages based on Lisp data were implemented as well with an aim to formalize some aspects of the supercompilation algorithms [46, 37, 20, 22]. The most advanced supercompiler for Refal is SCP4 [38, 39, 41].

3 Finite Countermodels and Program-State Reachability

Given a term $t \in \mathscr{T}(\mathscr{C}, \mathscr{V}, \mathscr{F})$, the set of instances of $t\theta$ such that the substitution θ is an object on $\mathscr{V}(t)$ is called the state set of t. Given a program $P = \langle \tau, R \rangle$ in $\mathscr{L}$, the state set S_0 of τ is called the initial states of P. The computing system R is able to evolve, starting from a fixed state $s_0 \in S_0$ and producing some states of P. Suppose that S_0 is described by a predicate (characteristic function) $\Sigma_0(\cdot)$ written in a logical language $\mathscr{M}$. Let $\mathscr{B}$ be a formal theory defined in $\mathscr{M}$ and $\phi(\cdot)$ be a formula (predicate) in $\mathscr{M}$, describing some state set of P. Assume that R satisfies the following: given two states s_0, s of P, if s is reachable from s_0 then $\mathscr{B}, \phi(s_0) \vdash \phi(s)$. Suppose that a formula $\psi(\cdot)$ (hypothesis in $\mathscr{B}$) specifies a set of bad states, reachability of which contradicts a safety property of the system P. Then refutation of $\psi(s)$ (in the theory $\mathscr{B} \wedge \phi(s_0)$) will mean the fact of safety of P – unreachability of the states satisfying the formula $\psi(\cdot)$. One may refute the hypothesis $\psi(s)$ by producing a counterexample; i.e. a countermodel of the formula/theory $\mathscr{B} \wedge \phi(s) \rightarrow \psi(s)$.

3.1 Formal Theory of $\mathscr{D}$

Recall that the data set $\mathscr{D}$ of our presentation language (see Sec. 2.1) is an algebraic structure – the free monoid of concatenation with an additional free unary operation. Characters are the constants of the monoid. In this paper we assume that the character alphabet $\mathscr{A}$ is finite and $\beta, \gamma \notin \mathscr{A}$, $: \notin \mathscr{A}$. Let β

[2] from *super*vised *compilation*

stand for the unary operation. In this encoding the data set $\mathscr{D}$ is redefined as follows:
$\mathscr{D} \ni d$ `::=` ε | γ | $d_1 : d_2$ | $\beta(d)$, where γ ranges over $\mathscr{A} \setminus \{\varepsilon\}$.

Let $\mathscr{A}$ be $\{\varepsilon, \texttt{'a'}, \texttt{'b'}\}$, consider the following theory $T_{\mathscr{D}}$ in the first-order predicate logic:

$\forall x,y,z.(x:y):z = x:(y:z)$
$\forall x.x:\varepsilon = x$
$\forall x.\varepsilon:x = x$
$(\neg(\varepsilon = \texttt{'a'})) \wedge (\neg(\varepsilon = \texttt{'b'})) \wedge (\neg(\texttt{'a'} = \texttt{'b'}))$
$R(\varepsilon) \wedge R(\texttt{'a'}) \wedge R(\texttt{'b'})$
$\forall x.R(x) \to R(\beta(x))$
$\forall x,y.R(x) \wedge R(y) \to R(x:y)$

The first three axioms are the free monoid axioms: the first one expresses associativity of concatenation, the second and third axioms say the constant ε is the identity element. The last three axioms axiomatize the unary predicate $R(\cdot)$ reflecting the inductive definition of the data set.

The theory $T_{\mathscr{D}}$[3] represents the data set $\mathscr{D}$ as stated in the following proposition

Proposition 1 $d \in \mathscr{D} \Leftrightarrow T_{\mathscr{D}} \vdash R(d)$

4 Unfolding and $\mathscr{L}$-Program-State Reachability

Given a function call $\texttt{st}_0 = \texttt{f}_k\texttt{(d}_0\texttt{)}$, where $k \in \mathbb{N}, \texttt{f}_k \in \mathscr{F}_k, \texttt{d}_0 \in \mathscr{D}^k$, the abstract $\mathscr{L}$-machine `Int(p,·)`, starting from the state $\texttt{f}_k\texttt{(d}_0\texttt{)}$ by stepwise matching $\texttt{d}_0$ against the left-sides l_i of the rules defining $\texttt{f}_k$, chooses a particular rule ρ_{i_0} of $\texttt{f}_k$ and constructs a next state based on the right-side of ρ_{i_0}. This matching algorithm can be seen as an algorithm stepwise solving the following equations $l_i = \texttt{d}_0$. That is to say, the matching chooses a mapping (Markov's substitution) $\sigma(\cdot) : \mathscr{V}(l_i) \to \mathscr{D}$ such that $\sigma(l_i) = \texttt{d}_0$, if such a mapping exists. Let $\texttt{Step(st}_0\texttt{)}$ denote an algorithm including the successful pattern matching and the replacement of $\texttt{st}_0$ with $\sigma(r_i)$.

In general, `Int(p,·)` state-wise iterates `Step(st)` when the state `st` is a configuration from the function stack top and input data of the state `st` are completely static (known). The unfolding algorithm approximates the semantic tree of `p` by means of iterating a meta-extension `MStep(·)` of `Step(·)` in the case when its input data `st` may be parameterized (partially unknown/dynamic). A launch of `MStep(st)` results in a tree being branched with the corresponding input parameters. The parameters' branchings are produced by a meta-extension of the matching, i.e. by an algorithm solving the equations $l_i = \texttt{st}$ of general form in the term monoid $\mathscr{T}(\mathscr{C}, \mathscr{V}, \mathscr{F})$. In particular, the algorithm has to solve word equations: in general, this task is very nontrivial as mentioned above (see Sec. 1), although there exist several simple algorithms solving such equations of some restricted forms.

4.1 One-Step Unreachability

Taking into account the hardness of Makanin's algorithm one may approximate the semantic tree as follows. Let a program rule $l = r$ and a parameterized state `st` be given. Before launching a main algorithm `F` trying to solve the equation $l = \texttt{st}$, `MStep` tries to prove that this equation has no solutions, using a specialized algorithm `UnSat`. `UnSat` may be incomplete, i.e. sometimes it may fail in proving unsatisfiability of the equation or even work in unbounded time on some its input. But `UnSat` may

[3] Note that if $T_{\mathscr{D}}$ is used only for constructing a countermodel $\mathscr{M}$ in $T_{\mathscr{D}}$ of a formula F, then the fourth axiom may be varied, since existence of such an $\mathscr{M}$ in $T_{\mathscr{D}}$ implies that $\mathscr{M}$ is a countermodel of F in a wider (weaker) theory as compared with $T_{\mathscr{D}}$.

be simple enough to be implemented for an existing external computing system with respect to the specializer unfolding the tree. One may limit the time being taken by `UnSat`. If `UnSat` does not finish its work in the given time limit, we may think `UnSat` fails in proving the unsatisfiability. In the fail case we just call `F`. If `UnSat` succeeds, then the program rule $l = r$ being considered is unreachable from the parameterized state `st` and the tree branch corresponding to this rule should be pruned away. If `F` succeeds, then it, like a Prolog interpreter, may return a simple narrowing of the parameters of `st` and Markov's substitution depending on the narrowed parameters and satisfying the equation $l =$ `st`. This substitution allows us to proceed with the unfolding. Unfortunately the narrowing of the parameters may not be expressed in the language $\mathscr{U}$ describing the parameterized configurations of the program being transformed, even if the algorithm `F` takes a reasonable time. For example, the Markov decision set may be recursive, while the language $\mathscr{U}$ is not. We now exemplify the situation.

Example 5 *Consider the following program* $p = \langle \tau, R \rangle$, `τ = f('a' : e.q, e.q : 'a')` *and R is*

```
f(e.x, e.x) = 'T';
f(e.x, e.y) = 'F' : (e.x) : (e.y);
```

The extended pattern matching has to solve the following system of the parameterized equations:

$$\begin{cases} \texttt{e.x} = \texttt{'a' : e.q} \\ \texttt{e.x} = \texttt{e.q : 'a'} \end{cases} \qquad (\star)$$

It is equivalent to the only relation $\Phi(\texttt{e.q})$ *(equation) –* `'a' : e.q = e.q : 'a'` *imposed on the parameter* `e.q`.[4]

At a naïve glance, $\Phi(\texttt{e.q})$ *must be the predicate labeling the first branch from the semantics tree root, while the second branch must be labeled by the negation* $\neg\Phi(\texttt{e.q})$. $\Phi(\texttt{e.q})$ *narrows the range of* `e.q`. *But the problem is* $\Phi(\texttt{e.q})$ *cannot be represented in the pattern language, using at most finitely many of the patterns to define a one-step program (a result of the unfolding). Recursion should be used to check whether a given input data belongs to the truth set of* $\Phi(\texttt{e.q})$. *This recursion is unfolded as an infinite tree.*

The `e.x`*-variable multiplicity* $\mu_{\texttt{e.x}}(\texttt{f(e.x, e.x)}) > 1$ *causes this problem: the system* $(\star)$ *implies an equation, where the parameter* `e.q` *plays the role of a variable and* both sides of the equation *contain* `e.q`.

Let us replace the initial parameterized configuration in Example 5:

$$\tau = \texttt{f('a' : e.q : 'a' : e.q : 'b', e.q : 'a' : e.q : 'b' : e.q)}$$

The extended pattern matching now has to solve the following equation

$$\texttt{'a' : e.q : 'a' : e.q : 'b'} = \texttt{e.q : 'a' : e.q : 'b' : e.q}$$

It is easy to see that this equation having variable `e.q` both in the left and right sides is inconsistent in $\mathscr{D}$. Mace4 automated finite model finder by W. McCune [36] quickly recognizes this fact in the context of the formal first-order theory $T_{\mathscr{D}}$ (see Sec. 3.1). I.e. Mace4 finds a finite countermodel of the following formula $\exists$`e.q.` `'a' : e.q : 'a' : e.q : 'b' = e.q : 'a' : e.q : 'b' : e.q` in the theory $\mathscr{D}$. That is unfolding

[4] It is easy to see that its solution set is $\{\theta_i(\texttt{e.q}) = \texttt{'a'}^i \mid i \in \mathbb{N}\}$.

the program being considered in the given context can prune away the first rule of R and result in the following program (tree): $p_1 = \langle \tau_1, R_1 \rangle$, $\tau_1 =$ `f1('a':e.q:'a':e.q:'b', e.q:'a':e.q:'b':e.q)` and R_1 is the only rule: `f1(e.x, e.y) = 'F':(e.x):(e.y);`

Note that we did not construct any narrowing of the parameter `e.q` and the information on the property of `e.q` (i.e. the equation above is inconsistent) is lost. The constructed tree is an approximation of the semantic tree of $\langle \tau, R \rangle$, rather than the exact semantics tree. If the right-side of the remaining rule include a function call then, in general, the lost information might be useful for further specialization. For example, it might be `'F' : g((e.x) : (e.y))`. The configuration τ_1 might be encountered in a middle vertex of the unfolding tree.

The example above demonstrates a potential feature of using a finite countermodel finder for improving the approximation of the semantics tree generated by the unfolding. An interface bundling the supercompiler SCP4 [38, 39, 41] with Mace4 has been implemented. It allows formulating in Mace4 such a kind of unreachability problem and returning to SCP4 the result produced by Mace4 during a time indicated by a user in a pseudo-comment of the program being transformed.

At first glance, the construction given in Example 5 can be generalized as follows. Let a program $P = \langle t, R \rangle$, $t = f(u_1, \ldots, u_k)$ and R – a function f definition below, where $k \in \mathbb{N}, f \in \mathscr{F}_k$ and for all $(1 \leq i \leq k)$ $u_i \in \mathscr{P}(\mathscr{C}, \mathscr{V})$, be given. Let $\#\mathscr{V}(t) = m \in \mathbb{N}$ and $\#\mathscr{V}(l_i) = s_i \in \mathbb{N}$. Let the sets $\mathscr{V}(t), \mathscr{V}(l_1), \ldots, \mathscr{V}(l_n)$ be ordered. Let q_j stand for the j-th element of $\mathscr{V}(t)$, while w_{ij} stand for the j-th element of $\mathscr{V}(l_i)$.

$$\begin{cases} f(p_{11}, \ldots, p_{1k}) & = & r_1 \\ & \ldots & \\ f(p_{n1}, \ldots, p_{nk}) & = & r_n \end{cases}$$

Definition 1 *Given $i \in \mathbb{N}$, $1 \leq i \leq n$, the rule $l_i = r_i$ of the function f is said to be one-step reachable from the term t if there exists a substitution $\theta : \mathscr{V}(t) \to \mathscr{D}$ such that for all $j \in \mathbb{N}$, $1 \leq j < i$, $t\theta$ does not match against l_j and it matches against l_i. A rule is said to be one-step unreachable if it is not one-step reachable.*

Let $i \in \mathbb{N}$, $1 \leq i \leq n$, be given. One may suspect that refuting the following formula, expressing reachability of a rule, leads to proving that the rule $l_i = r_i$ of the function f above is one-step unreachable from the term t.

$$\exists \texttt{e.v} \exists q_1, \ldots, q_m . ((\texttt{e.v} = \texttt{(}u_1\texttt{)} \ldots \texttt{(}u_k\texttt{)}) \wedge (\forall w_{11}, \ldots, w_{1s_1} \neg (\texttt{e.v} = \texttt{(}p_{11}\texttt{)} \ldots \texttt{(}p_{1k}\texttt{)})) \wedge$$
$$\ldots$$
$$(\forall w_{(i-1)1}, \ldots, w_{(i-1)s_{(i-1)}} \neg (\texttt{e.v} = \texttt{(}p_{(i-1)1}\texttt{)} \ldots \texttt{(}p_{(i-1)k}\texttt{)})) \wedge$$
$$(\exists w_{i1}, \ldots, w_{is_i} (\texttt{e.v} = \texttt{(}p_{i1}\texttt{)} \ldots \texttt{(}p_{ik}\texttt{)}))$$
$$)$$

But the variables here are assumed to range over the data set $\mathscr{D}$ only and so (naïve) application of a generic finite model finder may lead to vacuous countremodels, refuting the formula on the domain elements unrelated to $\mathscr{D}$. One may still to try to analyse such conditions automatically, possibly using alternating applications of the first-order model finder and a theorem prover. We will address this issue elsewhere.

Nevertheless we can overapproximate the reachability condition. Namely, we do not take care of any rule excluding the current rule being explored. That is to say, we approximate the $\mathscr{L}$ pattern matching with the pattern matching used in non-deterministic term rewriting. The corresponding formula to be refuted is as follows: $\exists q_1, \ldots, q_m . \exists w_{i1}, \ldots, w_{is_i} . \texttt{(}u_1\texttt{)} \ldots \texttt{(}u_k\texttt{)} = \texttt{(}p_{i1}\texttt{)} \ldots \texttt{(}p_{ik}\texttt{)}$.

So the refutation of this formula proves non-reachability by non-deterministic rewriting and therefore original non-reachability.

Concluding this section we emphasize that the worst-case time complexity of the program resulting in Example 5 is $O(1)$, while the worst-case time complexity of the original program is $O(n)$, where n is the input data size. Improving this complexity was done due to a launch of Mace4. There is a hidden loop over the input data in the first rule of the original program.

5 Global Unreachability

Unlike the most known specialization techniques, by definition, supercompilation is allowed to extend the domain of the partial function defined by the program being transformed. That makes supercompilation more flexible as compared with those methods. For example, supercompilation is able to improve the worst-case time complexity of some input programs, while partial evaluation cannot [19]. Other transformation techniques such as distillation [17] can also improve the worst-case time complexity of some programs. This section considers one of the supercompilation tools aiming to obtain such a kind of transformation in connection with finite countermodels.

5.1 Online Generated Program Output Formats

Given a program $P = \langle t, R \rangle$ and a substitution $\theta : \mathscr{V}(t) \to \mathscr{D}$, $[\![t\theta]\!]$ below denotes the result of (finite) computation of $t\theta$ according to the program P.

Definition 2 *Let a program $P = \langle t, R \rangle$ be given. A term $u \in \mathscr{P}(\mathscr{C}, \mathscr{V})$ is said to be an output format of the program P if for any substitution $\theta : \mathscr{V}(t) \to \mathscr{D}$ there exists a substitution $\eta : \mathscr{V}(u) \to \mathscr{D}$ such that $u\eta = [\![t\theta]\!]$. Let u_1 and u_2 be two output formats of P. If $u_1 \ll u_2$, then we say u_1 lesser than u_2. If u_1 lesser than any other output format of P, then u_1 is said to be a minimal output format of P.*

The minimal output format of a given program is not unique. Examples of the output formats are: `e.x` is an output format of any program; `s.y` is the only (modulo variable renaming) minimal output format of the program defined in Example 2; both `s.z : e.x` and `s.z : e.x : e.y` are minimal output formats of the program given in Example 5.

The tree T being stepwise generated by unfolding is potentially infinite. As a consequence it is an object to be somehow folded back into a finite graph representing the residual program Q. The folding algorithm works stepwise online, i.e. given an intermediate state of T the algorithm tries to fold a potentially infinite path in this intermediate state into a loop, using generalization of parameterized configurations/states of the original program P. We omit the detials of the generalization algorithm. Edges folding such paths are called references. Thus the intermediate state of T actually is a graph G rather than a tree (see Fig. 1). Given a vertex (parameterized state of P) v of G, if all references from the vertices on the paths originating in v are ingoing in the vertices from the same path set, then the part of G is a self-sufficient (closed) subgraph. Such a vertex v is called the root of the subgraph. The root is a potential entry point into one of the residual functions (sub-programs), i.e. the root is an input format of a residual function H. Let such a root just be created by a step of the folding algorithm. The supercompiler SCP4 analyses the subgraph H and constructs its output format. The calls of the folding algorithm are stepwise interleaved with the calls of the unfolding algorithm. Hence G may include both some completely folded subgraphs and still non-unfolded parameterized configurations of P. Such configurations may include some calls of the residual function H. A non-trivial output format of H restricts H's image set. Therefore the information brought beyond H's recursions (loops) might be used for specializing the function call (`g(e.z)` in Fig. 1) using the call of H as an argument. That immediately

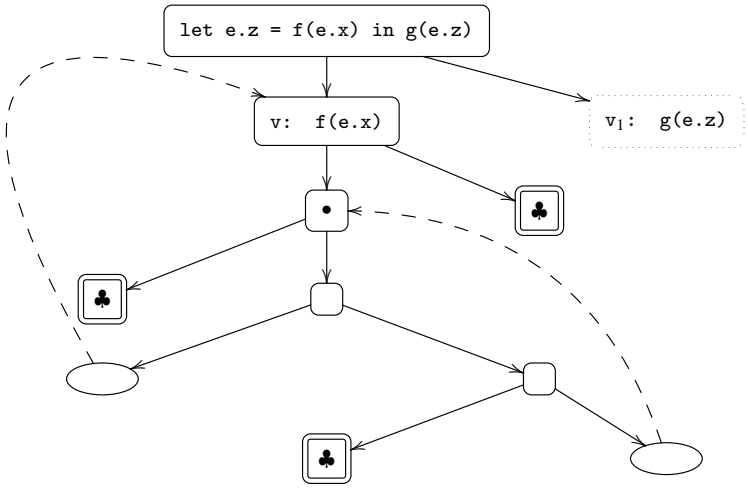

Figure 1: An intermediate state of an unfold-fold graph G: (1) the subgraph H rooted in v is self-sufficient, while (2) the subgraph rooted in • is not. The configuration v_1 still is not unfolded.

implies the worst-case time complexity of the residual program Q may be reduced as compared with the worst-case time complexity of P.

Note that the non-trivial output format may be a property of the original program P in the context of specialization (the initial configuration). In general, for example, P or its configuration being considered *per se* may have only the trivial output format. Thus the *online* generating of the output formats does matter.

The output format of the subgraph H is constructed by generalizing the formal exits from the recursions defining H (the double-boxing ♣ leaves in Fig. 1). Hence the lesser number of such exits the subgraph H includes the more specific output format of H may be constructed (see Def. 2 above). The recursion exits are presented as H's branches (edges) belonging to the paths outgoing from the root v. Thus a problem arises as follows. Given a subgraph H representing a potential residual sub-program one has to prove unreachability of as many of the syntactic recursion exits as possible. It is exactly the task we propose to delegate to a finite countermodel finder. Concrete launches of the finder may take too long times and even may not terminate. We suggest using the finder as explained in Sec. 4.1. A formal theory for the image set (or its superset) of the partial function H needed to launch the finder might be constructed by a compiler from the term rewriting language $\mathscr{L}$ into the first order logic language. The theory and a goal hypothesis to be refuted by the finder should somehow be based on the syntax of the subgraph corresponding to H. For the reasons explained in the previous Sec. 4.1 we can overapproximate the $\mathscr{L}$-unreachability with the non-deterministic term rewriting unreachability. The simplest hypothesis is the assumption that several given syntactic exit-branches from the recursion defined by H are unreachable from the subgraph root parameterized state. Below we consider simple examples of such formal theories and goals.

5.2 Two Simplest Output Formats

The simplest natural hypothesis is the partial (sub)function being analyzed is not the empty partial function. I.e. at least one of its syntactic exits is reachable from the subgraph root state. Refuting this hypothesis means the subgraph root itself is unreachable from the root of T and the branch leading in

this subgraph root is dead. Any term in $\mathscr{P}(\mathscr{C},\mathscr{V})$ is an output format of the empty partial function.

If the analysis of the hypothesis above does not lead to the empty partial function, the following hypothesis could be raised. The second simplest output format is a datum $d \in \mathscr{D}$. I.e. the (super)image set of (sub)function being analyzed includes the only datum. We might prove this fact if we were able to refute reachability of all syntactic exits excluding one, which surely returns the datum d.

Let us consider the program P given in Example 2. If the first two rules of the function `B` are unreachable from the initial configuration $\tau =$ `B(Fib(e.n))`, then the minimal output format of P is `'T'`. The supercompiler SCP4 is able to prove this fact by itself – without any call of a countermodel finder. This fact directly implies that none of the Fibonacci words contains `'bb'` as a subword. Let us slightly change the predicate from Example 2 as follows.

Example 6 *τ is `A(Fib(e.n))` and R is from Example 1 together with:*

```
A( (e.xs :'aaa') : (e.ys) ) = 'F';
A( (e.xs :'aa') : ('a' : e.ys) ) = 'F';
A( (e.xs) : (e.ys) ) = 'T';
```

SCP4 proves unreachability of the first two of `A`'s rules from the configuration `A(Fib(e.n))` and generates the minimal output format `'T'`. That means none of the Fibonacci words contains `'aaa'` as a subword. Notice that a more natural encoding of the same problem is presented in Example 7. For that encoding SCP4 fails to prove both of the properties of Fibonacci words. It cannot recognize that the first rules of both `A` and `B` are unreachable from the corresponding initial configurations.

Example 7

```
A( (e.xs) : (e.ys :'aaa': e.zs) ) = 'F';
A( (e.xs) : (e.ys) ) = 'T';
B( (e.xs) : (e.ys :'bb': e.zs) ) = 'F';
B( (e.xs) : (e.ys) ) = 'T';
```

One may try to prove that those first two rules are unreachable from τ, using the finite coutermodel method (see Sec. 3). For the reasons explained in the previous Sec. 4.1 we have to overapproximate the $\mathscr{L}$-reachability $\rightarrow^*$ with the non-deterministic term rewriting reachability $\Rightarrow^*$. A first order theory has to be created, in which derivability is compatible with the overapproximated reachability condition in the program being considered.

Following [27] we here consider a simpler example of a first-order theory Fib_0 demonstrating how to establish similar properties automatically using first-order theorem disproving by finite countermodels finding. The theory Fib_0 is as follows:

$T_{\mathscr{D}}$

.....

K(`'b'`, `'a'`).

K(`e.xs`, `e.ys`) $\rightarrow$ K(`e.ys`, `e.xs : e.ys`).

A(`e.ys : 'aaa' : e.zs`).

B(`e.ys : 'bb' : e.zs`).

Here the meaning of the predicate K(`e.xs`, `e.ys`) is `e.xs` and `e.ys` are two consecutive Fibonacci words. Negation of the last two axioms correspond to the properties defined by the predicates `A`, `B` given in Example 7. Stepwise computation of a given Fibonacci word $\texttt{e.xs}_0$ corresponds to stepwise derivability of $\exists$`e.ys`(K(`e.xs`$_0$, `e.ys`)). Mace4 is able to refute $\exists$`e.xs`$\exists$`e.ys`(K(`e.xs`, `e.ys`) $\wedge$ B(`e.xs`)) and $\exists$`e.xs`$\exists$`e.ys`(K(`e.xs`, `e.ys`) $\wedge$ A(`e.xs`)), by finding countermodels M_1 and M_2 of sizes 5 and 11, respectively. Description of these models can be found in [27].

6 Regular Invariants and Beyond

The finite models produced above can be seen as compact representations of the *regular* invariants (separators) sufficient to prove the safety, i.e non-reachability properties [28]. As the example above shows enhancing the supercompilation by the "regular verifying power" of FCM may be beneficial for producing non-trivial program transformations. What is interesting here is that the mechanisms for program analysis and transformation deployed within supercompilation do not cover all the regular power of FCM but may go beyond that.

Given a program rule $l = r$, obviously, using an `e/t`-variable v such that $\mu_v(r) > 1$ may lead to one-step computing a non-regular formal language $\mathscr{H} \subset \mathscr{D}$. Such a language $\mathscr{H}$ may also be generated by recursion. The following two examples deal with such a kind of recursion. The examples (being variations of the rules borrowed from [3]) define the empty partial function. The programs $\langle \tau, R \rangle$ below never reach exits from recursions. The exits are defined in the two first rules (in both of the programs). The first recursion given in the program `F` constructs two equal strings in the second and third arguments, using the associative concatenation. Correspondingly the second recursion given in the program `G` constructs two equal binary trees, using the parenthesis constructor. The first arguments of the programs are the recursion depths. Evaluation of the programs generates the following formal languages of the terms correspondingly:

$H_f = \texttt{f(}K\texttt{,'h}^n\texttt{':'A', 'h}^n\texttt{':'A')}$, $H_g = \texttt{g(}K\texttt{,}\overbrace{\texttt{('h':}}^{n}\texttt{'A'}\overbrace{\texttt{)}}^{n}\texttt{,}\ \overbrace{\texttt{('h':}}^{n}\texttt{'A'}\overbrace{\texttt{)}}^{n}\texttt{)}$, where $K = (\texttt{'b'} \mid \texttt{'c'})^m$ and $m \in \mathbb{N}$.

Example 8 *The program F is $\langle \tau, R \rangle$, where τ is `f(e.ps, 'A', 'A')` and R is*

```
f(ε, 'h':e.xs, 'A') = 'A';
f(ε, 'A', 'h':e.ys) = 'A';
f('b':e.ps, e.xs, e.ys) = f(e.ps, 'h':e.xs, 'h':e.ys);
f('c':e.ps, 'h':e.xs, 'h':e.ys) = f(e.ps, e.xs, e.ys);
```

Example 9 *The program G is $\langle \tau, R \rangle$, where τ is `g(e.ps, 'A', 'A')` and R is*

```
g(ε, ('h':e.xs), 'A') = 'A';
g(ε, 'A', ('h':e.ys)) = 'A';
g('b':e.ps, e.xs, e.ys) = g(e.ps, ('h':e.xs), ('h':e.ys));
g('c':e.ps, ('h':e.xs), ('h':e.ys)) = g(e.ps, e.xs, e.ys);
```

Proving the emptiness of the partial functions defined by F and G can be seen as safety verification, that is non-reachability of the first two rules of F and G (i.e. the exits from the recursions). In [3] Y. Boichut and P.-C. Heam showed that this safety property cannot be proved by safety verification techniques using *regular invariants*. It means in particular that FCM won't help in proving that. On the other hand well-known specialization methods can prune away the recursion exits from the program G. Indeed, the configuration sequence on the recursion path produced by the unfolding algorithm and outgoing from the initial configuration `g(e.ps,'A','A')` is: `g(e.ps,'A','A')`, $\texttt{g(e.ps}_1\texttt{,('h':'A'),('h':'A'))}$, $\texttt{g(e.ps}_2\texttt{,('h':('h':'A')),('h':('h':'A')))}$, ... Generalization algorithms based on the Higman-Kruskal relation [23] will construct one of the following configurations: `g(e.ps,t.x,t.x)`, $\texttt{g(e.ps}_1\texttt{,('h':t.x),('h':t.x))}$, $\texttt{g(e.ps}_2\texttt{,('h':('h':t.x)),('h':('h':t.x)))}$, ... Note that `t.x` here means exactly the same as a standard variable in languages based on Lisp data. Now obviously, the exit branches from the recursion will be pruned away from the unfolding tree.

The associative case used in F is more difficult. The supercompiler SCP4 recognizes the emptiness of both of the partial functions: supercompiling the program G results in a message on the empty partial function, while supercompiling the program F results in a program without *syntactic* exits from recursions.

Thus a finite countermodel finder and a specializer have incomparable verifying and analysing power and their joint use may strengthen both.

7 Conclusions and Future Work

From a practical point of view, use of external program tools by a program transformer lies on the stream of software development. Indeed, that belongs to modular programming technology, where each program module is considered as a unit of compilation. External provers were used in automated program transformers including specializers for a long time. For instance, in 1988 Y. Futamura used such an external tool for proving some properties of parameterized configurations/states of program being specialized [13, 12]. On the other hand, given a program specializer written in a language $\mathscr{U}$, $\mathscr{U}$ may include non-trivial semantics mechanisms in itself. The mechanisms may allow to implement non-trivial basic program transformation/analysis directly by means of $\mathscr{U}$-semantics, i.e. through *local* syntactical construction without intricate programming. Such a kind of mechanisms may be considered as external tools with respect to the specializer. Examples of such programming languages are Prolog and Refal [50]. For example, F. Fioravanti, A. Pettorossi and M. Proietti [8, 9, 10] as well as several other authors develop an unfold-fold based transformation technique of constraint logic programs with negation, implementing their transformers by constraint logic programming.

The use of countermodels for the execution and analysis of logic programs has been considered in the paper [4]. It has been noticed that the failure of a query Q for a logic program P can be established by finding a countermodel for $P \rightarrow Q$. Furthermore a particular strategy using pre-interpretations (i.e. interpretations of the predicate symbols only, ignoring constructors and data) combined with the use of an abduction mechanism is proposed and compared with unrestricted search of countermodels. Such a technique can be adapted for term rewriting systems and functional languages. Given a program P, the explicit syntax composition in P does not allow to ignore all constructors used by P, but some of the constructors can still be ignored. It is not quite clear how similar transformations affect the countermodel search method, especially whenever one uses an external countermodel finder.

The approach we presented in this paper is related also to the work on *abstract interpretations* [25, 6] and on *regular types* [18]. The work [14] explicitly connects both areas and demonstrates the transformations of the set of regular type definitions corresponding to the finite tree automata, into a finite pre-interpretation for a logic program, which then is shown can be used for the program analysis and verification. The core of the transformation is a determinization procedure for non-deterministic tree automata. The difference with our approach, apart of obvious differences between logic and functional programming languages considered, is that [14] deals with specific approach for pre-interpretation building, while we abstract away the details of implementation of the model building procedure, which is used as an oracle. Still after appropriate translations the approach of [14] can be used for the tasks considered in this paper and we plan to explore it in the future work.

In this paper we have shown that integrating a finite countermodel finder in a supercompiler may provide new features for non-trivial program transformations, which in turn may be used for non-trivial verification of safety properties of computing systems. In particular, global unreachability of some new kind of regular formal languages over the systems' state sometimes may be recognized. On the other

hand, Examples 8, 9 demonstrate that FCM may be strengthened by supercompilation tools. As regards this matter we would like to refer to an interesting example given by the researchers mentioned above working in the context of Prolog [9]. They derive a one-counter machine from a constrained regular language specification. The corresponding residual program tests that a string of a given length n does not belong to the language $\{\texttt{'a}^m\texttt{' : 'b}^n\texttt{'} \mid m = n \geq 0\}$.

Above we have described just the first steps and experiments in integrating FCM in a supercompiler. The examples given in Sec. 6 motivate future development of a compiler from a functional language (in our case, Refal) to the first-order logic language. The compiler should protect as many syntax properties of the program $\langle \tau, R \rangle$ being compiled as possible. The *overapproximated reachability* of the $\langle \tau, R \rangle$ states from τ should correspond to *derivability* in the corresponding compiled program. We conclude with the following note: FCM may be used for recognizing unreachable intermediate subgraphs generated by supercompilation even if the subgraphs are not self-sufficient (see Sec. 5). In such a case we have to treat the call names of the external function as free – to be entirely interpreted by FCM.

Acknowlegements

We are grateful to the reviewers of the paper for their generous and constructive comments, which both improved the presentation in this paper and will influence our future work.

References

[1] A. Ahmed, A. P. Lisitsa, and A. P. Nemytykh. Cryptographic protocol verification via supercompilation (A case study). In *VPT 2013*, volume 16 of *EPiC Series*, pages 16–29. EasyChair, 2013.

[2] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS 2009*, volume 5505 of *LNCS*, pages 307–321, 2009.

[3] Y. Boichut and P.-C. Heam. A theoretical limit for safety verification techniques with regular fix-point computations. *Information Processing Letters*, 108(1):1–2, September 2008.

[4] M. Bruynooghe, H. Vandecasteele, Andre de Waal, and Marc Denecker. Detecting unsolvable queries for definite prolog programs. arXiv:cs/0003067, 2000.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *10th Int. Static Analysis Symposium (SAS 2003)*, volume 2694 of *LNCS*, pages 1–18, 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of 4th ACM-SIGPLAN Symp. on Principles of Programming Languages (POPL'08)*, pages 281–292. ACM, 1977.

[7] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.

[8] 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.

[9] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 292–340. Springer, 2004.

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

[11] Y. Futamura. Partial evaluation of computing process an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[12] Y. Futamura, Z. Konishi, and R. Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 90:75–99, 2002.

[13] Y. Futamura and K. Nogi. Generalized partial computation. In *the IFIP TC2 Workshop*, pages 133–151, Amsterdam, 1988. North-Holland Publishing Co.

[14] John P. Gallagher and Kim S. Henriksen. Abstract domains based on regular types. In B. Demoen and V. Lifschitz, editors, *ICLP 2004*, volume 3132 of *LNCS*, pages 27–42. Springer, 2004.

[15] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI'08*, pages 206–215. ACM, 2008.

[16] J. Goubault-Larrecq. Finite models for formal security proofs. *Journal of Computer Security*, 6:1247–1299, 2010.

[17] G. W. Hamilton. Extracting the essence of distillation. In *The Proc. of the 7-th International Andrei Ershov Memorial Conference: Perspectives of System Informatics*, volume 5947 of *LNCS*, pages 151–164, 2009.

[18] E. K. Jackson, N. Bjørner, and W. Schulte. Canonical regular types. In *ICLP (Technical Communications*, pages 73–83, 2011.

[19] N. D. Jones. *Computability and Complexity from a Programming Perspective*. The MIT Press, 2000.

[20] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher order call-by-value language. *ACM SIGPLAN Notices*, 44(1):277–288, 2009.

[21] 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.

[22] I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Technical Report 62, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[23] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Trans. Amer. Math. Society*, 95:210–225, March 1960.

[24] H. Lehmann and M. Leuschel. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce. In *Proceedings of LOPSTR03*, volume 3018 of *LNCS*, pages 1–19, 2004.

[25] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *LOPSTR'99*, volume 1817 of *LNCS*, pages 63–82, 2000.

[26] A. Lisitsa. Finite countermodels for safety verification of parameterized tree systems. CoRR, abs/1107.5142, 2011.

[27] A. Lisitsa. Finite models for verification. a talk given in ENS Cachan, LSV, June 2012. `http://www.csc.liv.ac.uk/~alexei/countermodel/`.

[28] A. Lisitsa. Finite models vs tree automata in safety verification. In *23rd International Conference on Rewriting Techniques and Applications RTA'2012*, pages 225–239, 2012.

[29] A. Lisitsa. Finite reasons for safety. *Journal of Automated Reasoning*, 51(4):431–451, December 2013.

[30] 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.

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

[32] A. P. Lisitsa and A. P. Nemytykh. Solving coverability problems by supercompilation. Presentation on the Workshop on Reachability Problems - RP'08, 2008.

[33] A. P. Lisitsa and A. P. Nemytykh. A note on program specialization. what can syntactical properties of residual programs reveal? In *VPT 2014*, volume 28 of *EPiC Series*, pages 52–65. EasyChair, 2014.

[34] 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.

[35] A. A. Markov. The theory of algorithms. *AMS Translations*, 2(15):1–14, 1960.

[36] W. McCune. Prover9 and Mace4. [online]. `http://www.cs.unm.edu/~mccune/mace4/`.

[37] N. Mitchell and C. Runciman. A supercompiler for core haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *LNCS*, pages 147–164. Springer-Verlag, 2008.

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

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

[40] A. P. Nemytykh, V. A. Pinchuk, and V. F. Turchin. A self-applicable supercompiler. In *PEPM'96*, volume 1110 of *LNCS*, pages 322–337. Springer-Verlag, 1996.

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

[42] Antonina Nepeivoda. Ping-pong protocols as prefix grammars and Turchin relation. In *VPT 2013*, volume 16 of *EPiC Series*, pages 74–87. EasyChair, 2013.

[43] H. Ruan, J. Zhang, and J. Yan. Test data generation for c programs with string-handling functions. In *TASE'08*, pages 219–226. IEEE, 2008.

[44] P. Selinger. Models for an adversary-centric protocol logic. *Electr. Notes Theor. Comput. Sci.*, 55(1), 2001.

[45] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pages 13–22. IEEE, Sept. 2007.

[46] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[47] V. F. Turchin. The language Refal – the theory of compilation and metasystem analysis. Technical Report 20, Courant Institute of Mathematical Sciences, New York University, February 1980.

[48] 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, 1980.

[49] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[50] 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.