Finite Countermodel Based Verification for Program Transformation

(A Case Study)

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

[1]Department of Computer Science
The University of Liverpool

[2]Program Systems Institute
Russian Academy of Sciences

Workshop on Verification and Program Transformation
London, 2015

Unfold-Fold Program Transformation Methods

Unfold-Fold Program Transformation Methods:

- Include various types of (un)reachability tasks
- Solving the tasks may improve the semantics tree (of parameterized configurations) of the program being transformed.

That means some general-purpose verification methods may be used for strengthening program transformation techniques.

Unfold-Fold Program Transformation Methods

Unfold-Fold Program Transformation Methods:

- Include various types of (un)reachability tasks
- Solving the tasks may improve the semantics tree (of parameterized configurations) of the program being transformed.

That means some general-purpose verification methods may be used for strengthening program transformation techniques.

Safety Verification for Turchin's Supercompilation

- We consider the question how the particular safety verification method using finite countermodels (FCM) might be used in Turchin's supercompilation method.
- We extract a number of supercompilation sub-algorithms trying to solve (un)reachability problems
- and demonstrate the use of an external countermodel finder for solving some of the problems.

Safety Verification for Turchin's Supercompilation

- We consider the question how the particular safety verification method using finite countermodels (FCM) might be used in Turchin's supercompilation method.
- We extract a number of supercompilation sub-algorithms trying to solve (un)reachability problems
- and demonstrate the use of an external countermodel finder for solving some of the problems.

Safety Verification for Turchin's Supercompilation

- We consider the question how the particular safety verification method using finite countermodels (FCM) might be used in Turchin's supercompilation method.
- We extract a number of supercompilation sub-algorithms trying to solve (un)reachability problems
- and demonstrate the use of an external countermodel finder for solving some of the problems.

The Main Idea

The combination of supercompilation and the FCM is promising:

- the analysis exploited within supercompilation does not cover all regular properties of parameterized program configurations;
- supercompilation (and other specialization methods) sometimes is able to recognize non-regular program properties.
- The main idea 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.

The Main Idea

The combination of supercompilation and the FCM is promising:

- the analysis exploited within supercompilation does not cover all regular properties of parameterized program configurations;
- supercompilation (and other specialization methods) sometimes is able to recognize non-regular program properties.

- The main idea 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.

The Main Idea

The combination of supercompilation and the FCM is promising:

- the analysis exploited within supercompilation does not cover all regular properties of parameterized program configurations;
- supercompilation (and other specialization methods) sometimes is able to recognize non-regular program properties.

- The main idea 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.

The Main Idea behind a Supercompiler

Supervised compilation is a powerful semantic based unfold-fold program transformation method having a long history well back to the 1960-70s, when it was proposed by V. Turchin.

It observes 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 Main Idea behind a Supercompiler

A supercompiler

- unfolds a potentially infinite tree of all possible computations of a parameterized program P;
- reduces (in the process) the redundancy that could be present in P;
- folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of P;
- analyses global properties of the graph and specializes this graph w.r.t. these properties (without an additional unfolding).

Refal

- functional programming language (V. Turchin) used in work on Turchin's supercompilation
- in this work we use for presentation somewhat simplified/modified fragment of Refal

The Main Idea behind a Supercompiler

A supercompiler

- unfolds a potentially infinite tree of all possible computations of a parameterized program `P`;
- reduces (in the process) the redundancy that could be present in `P`;
- folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of `P`;
- analyses global properties of the graph and specializes this graph w.r.t. these properties (without an additional unfolding).

Refal

- functional programming language (V. Turchin) used in work on Turchin's supercompilation
- in this work we use for presentation somewhat simplified/modified fragment of Refal

The Main Idea behind a Supercompiler

A supercompiler

- unfolds a potentially infinite tree of all possible computations of a parameterized program `P`;
- reduces (in the process) the redundancy that could be present in `P`;
- folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of `P`;
- analyses global properties of the graph and specializes this graph w.r.t. these properties (without an additional unfolding).

Refal

- functional programming language (V. Turchin) used in work on Turchin's supercompilation
- in this work we use for presentation somewhat simplified/modified fragment of Refal

The Main Idea behind a Supercompiler

A supercompiler

- unfolds a potentially infinite tree of all possible computations of a parameterized program `P`;
- reduces (in the process) the redundancy that could be present in `P`;
- folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of `P`;
- analyses global properties of the graph and specializes this graph w.r.t. these properties (without an additional unfolding).

Refal

- functional programming language (V. Turchin) used in work on Turchin's supercompilation
- in this work we use for presentation somewhat simplified/modified fragment of Refal

The Main Idea behind a Supercompiler

A supercompiler

- unfolds a potentially infinite tree of all possible computations of a parameterized program `P`;
- reduces (in the process) the redundancy that could be present in `P`;
- folds the tree into a finite graph of states and transitions between possible (parameterized) configurations of `P`;
- analyses global properties of the graph and specializes this graph w.r.t. these properties (without an additional unfolding).

Refal

- functional programming language (V. Turchin) used in work on Turchin's supercompilation
- in this work we use for presentation somewhat simplified/modified fragment of Refal

The Rooted Binary Tree

LISP Data

The Forest

REFAL Data

Arbitrary trees. Associative concatenation.

Concatenation is Associative

The data set is a free monoid $\mathcal{D}$ of concatenation with an additional unary constructor, which is denoted only with its parenthesis (i.e. without a name).

Definition

$$\mathcal{D} \ni d \quad ::= \quad \epsilon \quad | \quad c \quad | \quad d_1 : d_2 \quad | \quad (d)$$

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.

Thus a datum is a finite sequence (including the empty sequence), which can be seen as a forest of arbitrary finite trees.

Concatenation is Associative

The data set is a free monoid $\mathcal{D}$ of concatenation with an additional unary constructor, which is denoted only with its parenthesis (i.e. without a name).

Definition

$$\mathcal{D} \ni d \quad ::= \quad \epsilon \quad | \quad c \quad | \quad d_1 : d_2 \quad | \quad (d)$$

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.

Thus a datum is a finite sequence (including the empty sequence), which can be seen as a forest of arbitrary finite trees.

Term Rewriting Systems Based on Pattern Matching

Call by Value

A program P is a pair $\langle \tau, R \rangle$ with $R = \{l_i = r_i \mid i = 1 \ldots n\}$.

- τ is the entry point;
- the sentences $l_i = r_i$ are ordered from top to bottom to be matched;
- $\forall_i.\ \texttt{Vars}(r_i) \subseteq \texttt{Vars}(l_i)$ are ordered from top to bottom to be matched.

Fibonacci words

Definition

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*, . . .

Term Rewriting Systems Based on Pattern Matching

Example Program

$\langle \tau, R \rangle$ computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument in the unary notation.

τ is `Fib(e.n)` and R is:

```
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);
```

- Variables:
  - s-variables: range over characters
  - e-variables: range over the whole domain

```
'b' : 'a' ≡ 'ba';  'aba' : e.x ≡ 'a' : 'b' : 'a' : e.x
Fib('III') = ('aba') : ('baaba')
           ≡ ('a' : 'b' : 'a') : ('b' : 'a' : 'a' : 'b' : 'a')
```

Term Rewriting Systems Based on Pattern Matching

Example Program

$\langle \tau, R \rangle$ computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument in the unary notation.

τ is `Fib(e.n)` and R is:

```
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);
```

- Variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole data set D;

```
'b' : 'a' ≡ 'ba';  'aba' : e.x ≡ 'a' : 'b' : 'a' : e.x
Fib('III') = ('aba') : ('baaba')
           ≡ ('a' : 'b' : 'a') : ('b' : 'a' : 'a' : 'b' : 'a')
```

Term Rewriting Systems Based on Pattern Matching

Example Program

$\langle \tau, R \rangle$ computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument in the unary notation.

τ is `Fib(e.n)` and R is:

```
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);
```

- Variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole data set D;

```
'b' : 'a' ≡ 'ba';  'aba' : e.x ≡ 'a' : 'b' : 'a' : e.x
Fib('III') = ('aba') : ('baaba')
           ≡ ('a' : 'b' : 'a') : ('b' : 'a' : 'a' : 'b' : 'a')
```

Term Rewriting Systems Based on Pattern Matching

Example Program

$\langle \tau, R \rangle$ computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument in the unary notation.

τ is `Fib(e.n)` and R is:

```
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);
```

- Variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole data set D;

```
'b' :'a' ≡'ba'; 'aba' :e.x ≡'a' :'b' :'a' :e.x
Fib('III') = ('aba') : ('baaba')
           ≡ ('a' :'b' :'a') : ('b' :'a' :'a' :'b' :'a')
```

Term Rewriting Systems Based on Pattern Matching

Example Program

$\langle \tau, R \rangle$ computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument in the unary notation.

τ is `Fib(e.n)` and R is:

```
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);
```

- Variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole data set $\mathcal{D}$;

```
'b' : 'a' ≡ 'ba';  'aba' : e.x ≡ 'a' : 'b' : 'a' : e.x
Fib('III') = ('aba') : ('baaba')
           ≡ ('a' : 'b' : 'a') : ('b' : 'a' : 'a' : 'b' : 'a')
```

Term Rewriting Systems Based on Pattern Matching

Example Program

$\langle \tau, R \rangle$ computes the n-th pair of consecutive Fibonacci words, where n is given in the input argument in the unary notation.

τ is `Fib(e.n)` and R is:

```
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);
```

- Variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole data set $\mathcal{D}$;

```
'b' : 'a' ≡ 'ba';  'aba' : e.x ≡ 'a' : 'b' : 'a' : e.x
Fib('III') = ('aba') : ('baaba')
           ≡ ('a' : 'b' : 'a') : ('b' : 'a' : 'a' : 'b' : 'a')
```

Computig Fibonacci words and testing their properties

Example Programs

$\langle \tau, R \rangle$ computes the `e.n`-th pair of consecutive Fibonacci words.

τ is `Fib(e.n)` and R is:

```
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);
```

$\langle \tau_2, R_2 \rangle$ is a predicate testing two given consecutive Fibonacci words.

τ_2 is `B(Fib(e.n))` and R_2 is the union of R and:

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

On Semantics of the Pattern Matching

Markov's Rule

$$\texttt{'AB'} : \texttt{e.X} : \texttt{'CD'} : \texttt{e.Y} : \texttt{'D'} = \texttt{'ABCDCDCDCDDD'}$$

Deterministic choice:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- $\sigma_4(\texttt{e.X}) = \epsilon$, $\sigma_4(\texttt{e.Y}) = \texttt{'CDCDCDD'}$

On Semantics of the Pattern Matching

n-Ary Functions

Given data $d_1, \ldots, d_n$, matching $f(t_1, \ldots, t_n)$ against $f(d_1, \ldots, d_n)$ leads to solving the following system of equations in the free monoid:

$$(t_1) \ldots (t_n) = (d_1) \ldots (d_n) \Leftrightarrow \begin{cases} t_1 = d_1 \\ \ldots \\ t_n = d_n \end{cases}$$

FCM: Safety Properties Verification by Refutation

$P = \langle \tau, R \rangle$

Let S_0 be the initial states of P/τ, $\mathcal{B}$ be a theory defined in a lang. $\mathcal{M}$ and $\phi(\cdot) \in \mathcal{M}$ specify a subset of S_0, and $\psi(\cdot)$ – a set of bad st-s of P.

Assume that R satisfies the following:

Given two states s_0, s of P, if s is reachable from s_0 then $\mathcal{B}, \phi(s_0) \vdash \phi(s)$. Where $s_0 \in S_0$.

Refutation of $\psi(s)$ (in the theory $\mathcal{B} \wedge \phi(s_0)$) means:

The fact of safety of P – unreachability of the states satisfying $\psi(\cdot)$.

One may refute the hypothesis $\psi(s)$ by a countermodel of $\mathcal{B} \wedge \phi(s) \rightarrow \psi(s)$.

Formal Theory of $\mathcal{D}$ in the First-Order Predicate Logic

Let the character alphabet $\mathcal{A} = \{\epsilon, \texttt{'a'}, \texttt{'b'}\}$ and $\beta, \gamma \notin \mathcal{A}$, $: \notin \mathcal{A}$.
The data set $\mathcal{D}$ is redefined as follows:

$$\mathcal{D} \ni d \ ::= \ \epsilon \ \mid \ \gamma \ \mid \ d_1 : d_2 \ \mid \ \beta(d) \text{ , where } \gamma \in \mathcal{A} \setminus \{\epsilon\}$$

Theory $T_{\mathcal{D}}$

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

Proposition

$d \in \mathcal{D} \Leftrightarrow T_{\mathcal{D}} \vdash R(d)$

Formal Theory of $\mathcal{D}$

Can we used for:

- Pattern matching and word equations solving
- One-step reachability analysis
- Global reachability analysis
- . . .

One-Step Unreachability

Example Program $\langle \tau, R \rangle$

τ = `f('a':e.q:'a':e.q:'b',e.q:'a':e.q:'b':e.q)`
and R is:

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

We have the word equation:

`'a':e.q:'a':e.q:'b' = e.q:'a':e.q:'b':e.q`

This equation is inconsistent in $\mathcal{D}$.

One-Step Unreachability

Mace4 Automated First-Order Finite Model Finder

We have the word equation:

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

- This equation is inconsistent in $\mathcal{D}$.
- Mace4 by W. McCune quickly recognizes this fact in the context of the formal first-order theory $T_{\mathcal{D}}$.
- Mace4 finds a finite countermodel of the following formula

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

  in the theory $\mathcal{D}$.
- The first rule of the input program $\langle \tau, R \rangle$ may be pruned away.

One-Step Unreachability

Unfolding

Input Program: τ = `f('a':e.q:'a':e.q:'b',e.q:'a':e.q:'b':e.q)` and R is:

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

- The first rule of the input program may be pruned away.

Residual Program: $\langle \tau_1, R_1 \rangle$, τ_1 = `f1('a':e.q:'a':e.q:'b',e.q:'a':e.q:'b':e.q)` and R_1 is:

```
f1(e.x, e.y) = 'F':(e.x):(e.y);
```

Global Reachability and Program Output Formats

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

Definition

A passive term u is said to be an output format of the program P if for any substitution $\theta : \mathtt{Vars}(t) \to \mathcal{D}$ there exists a substitution $\eta : \mathtt{Vars}(u) \to \mathcal{D}$ such that $u\eta = [\![t\theta]\!]$.

Program Output Formats

Examples

- `e.x` is an output format of any program;
- both `s.z : e.u` and `s.z : e.u : e.v` are minimal output formats of the program $\langle \tau, R \rangle$:

$\langle \tau, R \rangle$, 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);
```

Program Output Formats

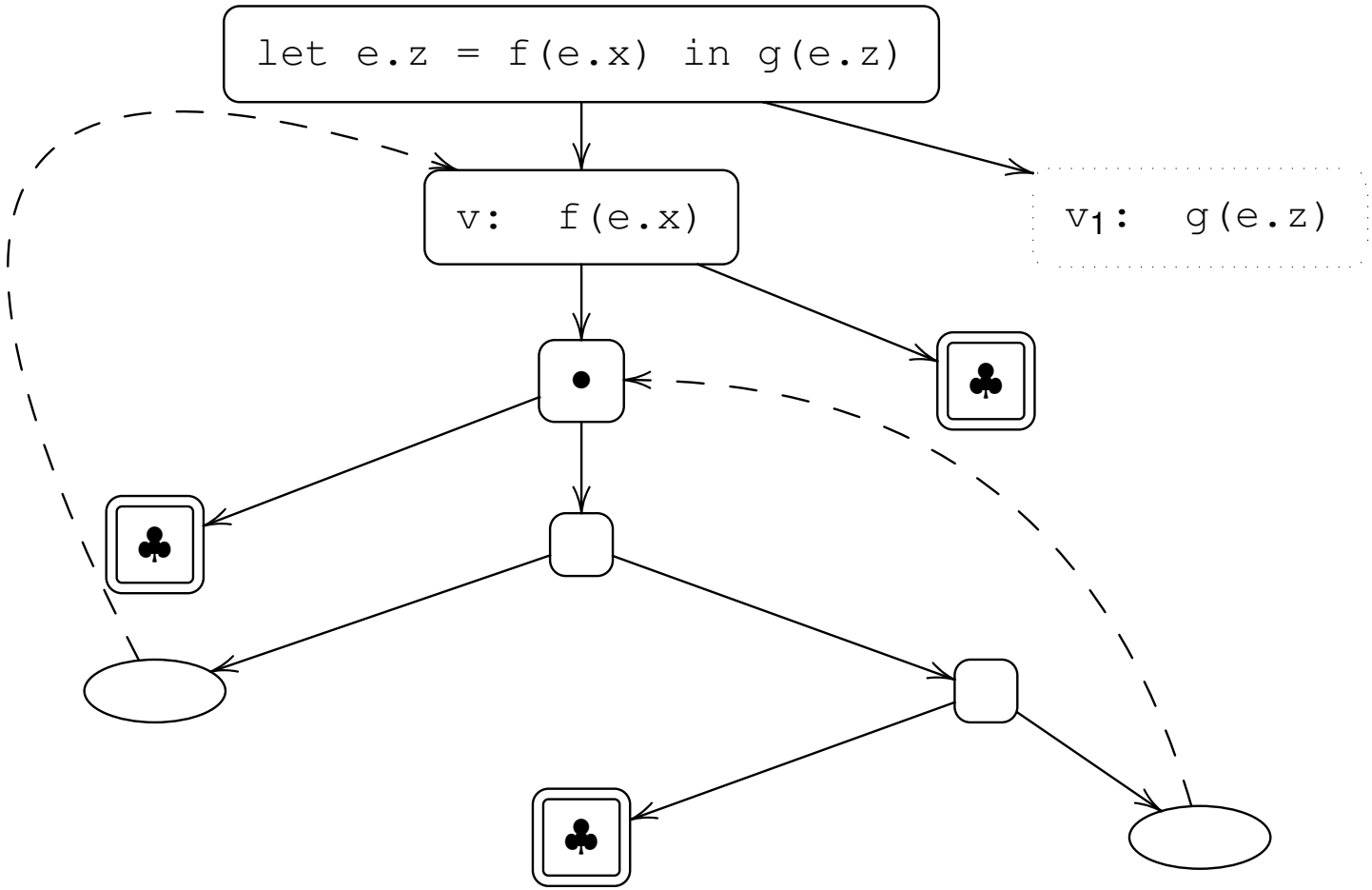
Examples

- `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 the program $\langle \tau_2, R_2 \rangle$:

τ_2 is `B( (e.xs):(e.ys) )` and R_2 is:

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

Online Generated Program Output Formats



An intermediate state of an unfold-fold graph: (1) the subgraph rooted in $\mathtt{v}$ is self-sufficient, while (2) the subgraph rooted in $\bullet$ is not.

The configuration $\mathtt{v}_1$ still is not unfolded.

Online Generated Program Output Formats

A Fibonacci Word Property

τ_2 is `B(Fib(e.n))` and R_2 is:

```
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);

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

- The supercompiler SCP4 is able to prove that the first two rules of the function `B` are unreachable from τ_2.
- None of the Fibonacci words contains `'bb'` as a subword.

Online Generated Program Output Formats

A Fibonacci Word Property

τ_2 is `B(Fib(e.n))` and R_2 is:

```
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);

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

- The supercompiler SCP4 is able to prove that the first two rules of the function `B` are unreachable from τ_2.
- None of the Fibonacci words contains `'bb'` as a subword.

Online Generated Program Output Formats

A Fibonacci Word Property

τ_2 is `B(Fib(e.n))` and R_2 is:

```
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);

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

- The supercompiler SCP4 is able to prove that the first two rules of the function `B` are unreachable from τ_2.
- None of the Fibonacci words contains `'bb'` as a subword.

Online Generated Program Output Formats

The Fibonacci Word Property: A More Natural Encoding

τ_3 is `B(Fib(e.n))` and R_3 is:

```
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);

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

- The supercompiler SCP4 fails to prove that the first rule of the function `B` is unreachable from τ_3.
- The output format of the corresponding residual program is `s.z`.

Online Generated Program Output Formats

The Fibonacci Word Property: A More Natural Encoding

τ_3 is `B(Fib(e.n))` and R_3 is:

```
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);

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

- The supercompiler SCP4 fails to prove that the first rule of the function `B` is unreachable from τ_3.
- The output format of the corresponding residual program is `s.z`.

Online Generated Program Output Formats

The Fibonacci Word Property: A More Natural Encoding

τ_3 is `B(Fib(e.n))` and R_3 is:

```
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);

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

- The supercompiler SCP4 fails to prove that the first rule of the function `B` is unreachable from τ_3.
- The output format of the corresponding residual program is `s.z`.

The Finite Coutermodel Method

The Fibonacci Word Property by Refutation

A first-order theory Fib_0

$T_{\mathcal{D}}$

.....

$K(\texttt{'b'},\texttt{'a'})$.

$K(\texttt{e.xs},\texttt{e.ys}) \rightarrow K(\texttt{e.ys},\texttt{e.xs} : \texttt{e.ys})$.

$B(\texttt{e.ys} : \texttt{'bb'} : \texttt{e.zs})$.

- $K(\texttt{e.xs},\texttt{e.ys})$ means $\texttt{e.xs}$ and $\texttt{e.ys}$ are two consecutive Fibonacci words.
- Stepwise computation of a given Fibonacci word $\texttt{e.xs}_0$ corresponds to stepwise derivability of $\exists \texttt{e.ys}(K(\texttt{e.xs}_0,\texttt{e.ys}))$.
- Mace4 is able to refute $\exists \texttt{e.xs} \exists \texttt{e.ys}(K(\texttt{e.xs},\texttt{e.ys}) \wedge B(\texttt{e.xs}))$ by finding a countermodel M_1 of sizes 5.

Regular Invariants

by the Finite Countermodels Method (FCM)

The finite model M_1 produced by Mace4 can be seen as compact representations of the regular invariants (separators) sufficient to prove the safety, i.e. non-reachability properties (A. Lisitsa, 2012).

- FCM may be beneficial for producing non-trivial program transformations.
- The mechanisms for program analysis and transformation deployed within supercompilation do not cover all the regular power of FCM.

Non-Regular Properties

The mechanisms for program analysis and transformation deployed within supercompilation do not cover all the regular power of FCM but may go beyond that.

Non-Regular Properties

String Example

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

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);
```

- The supercompiler SCP4 prunes away the two first sentences.

Non-Regular Properties

Binary Tree Example

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

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);
```

- The supercompiler SCP4 prunes away the two first sentences.

Final Remarks

The combination of supercompilation and the FCM is promising:

- The main idea 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.

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Related Works

- P. Cousot, R. Cousot, *Abstract interpretations* (1977 . . .)
- M. Leuschel et al., *Infinite state model checking by abstract interpretation* (1999 . . .)
- M. Bruynooghe, H. Vandecasteele, A. de Waal, M. Denecker, *Detecting unsolvable queries for definite prolog programs* (2000)
- F. Fioravanti, A. Pettorossi, M. Proietti, *A non-regular predicate derived from a constrained regular language specification* (2004)
- J. P. Gallagher, K. S. Henriksen, *Abstract domains based on regular types* (2004)
- E. K. Jackson, N. Bjørner, W. Schulte, *Canonical regular types* (2011)
- A. Lisitsa, *Finite countermodels for safety verification of parameterized tree systems* (2011)

Future Work

The examples given above 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 being compiled as possible.
- The overapproximated/non-deterministic reachability of the program $\langle \tau, R \rangle$ states from τ should correspond to derivability in the corresponding compiled program.

Thank You