Verifying Programs via Intermediate Interpretation

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
Uppsala, 2017

The Verification Problem

Given:

- a partial predicate p defined in a functional language:

  $$p : \mathbb{D} \to \{True, False, \perp\},$$

  i. e., p terminates for any $d \in \mathbb{D}$, but may fall into an abnormal deadlock state $\perp$;
- an input subset $Init \subseteq \mathbb{D}$.

Does there exist $d_0 \in Init$ such that $p(d_0)$ returns/reaches *False*?

That is a (un)reachability problem.

✓ This study focuses on capabilities of automatic recognizing the safety properties by the unfold-fold program transformation.

The Main Observation

Given: a computing system S, a safety property $\psi(\cdot)$ of S

- ▼ $f(n, \vec{x})$ produces n-th state of S, σ is a state of S.

$$\begin{cases} p(\sigma) \text{ returns } \mathit{False} & \text{iff } \psi(\sigma) \text{ does not hold;} \\ p(\sigma) = \mathit{True} & \text{otherwise} \end{cases}$$

- ▲ Since $p(\cdot)$ and $f(\cdot, \cdot)$ terminate,
  if $\forall n \,.\, (\mathbb{N} \ni n > 0)\ \forall \vec{x} \in \mathit{Init} \Longrightarrow p(f(n, \vec{x}))$ does not return *False*,
  then S is safe whenever it starts from *Init*.

✓ This study focuses on capabilities of automatic recognizing such (un)reachability properties by the unfold-fold program transformation methods via intermediate interpretation.

The Aim

- ✓ We explore potential capabilities of an unfold-fold program specialization method called Turchin's supercompilation, for verifying the safety properties of the functional programs modeling a class of **complicated** computing systems.

The Main Idea / LN 2007

Use supercompilation aiming at moving the safety property hidden in the program semantics to a simple syntactic property of the residual program:

- this syntactic property should be easily recognized;
- hope the corresponding residual programs will include no operator "*return False*;"

$f(n, \vec{x})$ produces n-th state of a computing system S, $\psi(\cdot)$ is its safety property over $Init \subseteq \mathbf{D}$: $Init \ni \sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$

$$p(f(n, \vec{x})) \text{ returns } \begin{cases} False & \text{iff } \psi(\sigma_n) \text{ does not hold, whenever} \\ & S \text{ starts from } Init; \\ True & \text{otherwise} \end{cases}$$

The Main Idea / LN 2007

Use supercompilation aiming at moving the safety property hidden in the program semantics to a simple syntactic property of the residual program:

- this syntactic property should be easily recognized;
- hope the corresponding residual programs will include no operator "*return False*;"

$f(n, \vec{x})$ produces n-th state of a computing system S, $\psi(\cdot)$ is its safety property over $Init \subseteq \mathbf{D}$: $Init \ni \sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$

$$p_{res}(n, \vec{x}) \text{ returns } \begin{cases} \sout{False \quad \text{iff } \psi(\sigma_n) \text{ does not hold whenever}} \\ \sout{\quad S \text{ starts from } Init;} \\ True \quad \text{otherwise} \end{cases}$$

A Class of Non-Deterministic Parameterized Cache Coherence Protocols

Synapse N+1, MSI, MOSI, MESI, MOESI, Illinois University, Berkley RISC, DEC Firefly, IEEE Futurebus+, Xerox PARC Dragon

Various methods for verification have been tried on the benchmark:

- J. Esparza, A. Finkel, and R. Mayr, 1999-. . . ;
- G. Delzanno et al., 2000-. . . ;
- E. Emerson and V. Kahlon, 2003;
- F. Fioravanti, A. Pettorossi, and M. Proietti, 2007-. . . ;
- A. Lisitsa and A. Nemytykh, 2007-. . . ;
- . . . .

Specifying Non-Deterministic Cache Coherence Protocols in Functional Programming Language: The Main Idea / LN 2007

$f(n, \vec{x})$ produces n-th state of a computing system S, $\psi(\cdot)$ is its safety property over $Init \subseteq \mathbf{D}$: $Init \ni \sigma_1 \to \sigma_2 \to \ldots \to \sigma_n$

$$p(f(n,\vec{x})) \text{ returns } \begin{cases} \mathit{False} & \text{iff } \psi(\sigma_n) \text{ does not hold, whenever } S \text{ starts from } \mathit{Init}; \\ \mathit{True} & \text{otherwise} \end{cases}$$

- Consider the n as a *time*.
- The *time* value is modeled by a finite stream of external events.
- The *time* ticks are labeled by the events.
- The protocol has to react to the external non-deterministic events by updating its states.

The Benchmark

A Class of Non-deterministic Parameterized Cache Coherence Protocols

Synapse N+1, MSI, MOSI, MESI, MOESI, Illinois University, Berkley RISC, DEC Firefly, IEEE Futurebus+, Xerox PARC Dragon

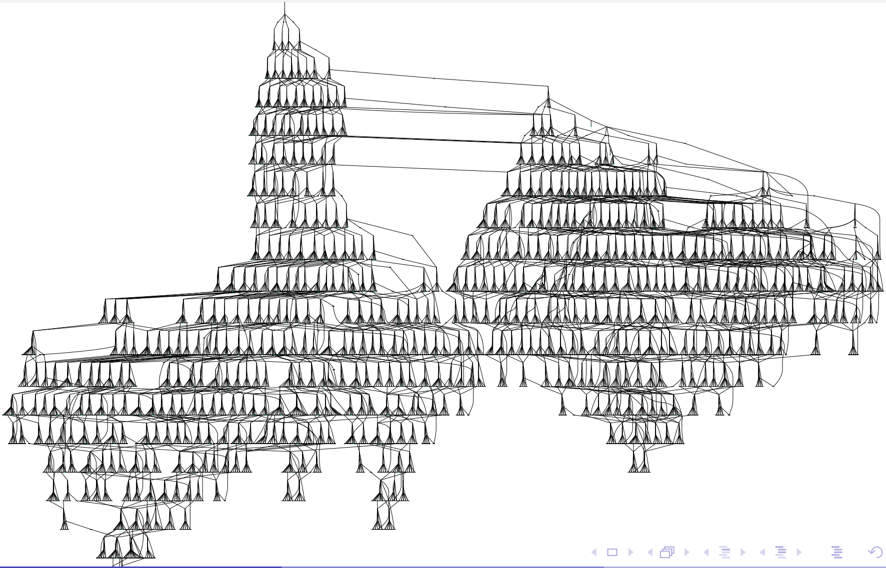
Early we have verified some safety properties of these protocols, as well as several other broadcast non-deterministic protocols (LN 2007)

- specified in terms of a strict functional programming language
- using the direct verification method described above
- by means of the supercompiler SCP4

Later this approach was extended by G. W. Hamilton for verifying a wider class of temporal properties of reactive systems. (2015)

A Proof Structure for Safety Verification by SCP4

Two Consumers - Two Producers protocol / Abstract Multithreaded Java Program



The Presentation Language $\mathcal{L}$

$prog$	$::=$	$def_1 \ \ldots \ def_n$	Program
def	$::=$	$f(\ ps_1\ ) \Rightarrow exp_1;\ \ldots;\ f(\ ps_n\ ) \Rightarrow exp_n;$	Function Definition
exp	$::=$	v	Variable
	$\mid$	$term : exp$	*Cons* Application
	$\mid$	$f\textbf{(}\ exp_1, ..., exp_n\ \textbf{)}$	Function Application
	$\mid$	exp_1 **++** exp_2	*Append* Application
	$\mid$	$[\,]$	*Nil*
$term$	$::=$	$s.name$	Symbol-Type Variable
	$\mid$	(exp)	Constructor Application
	$\mid$	σ	Symbol
ps	$::=$	$p_1, ..., p_n$	Patterns
p	$::=$	$v \mid s.name : p \mid (p_1) : p_2 \mid \sigma : p \mid [\,]$	
v	$::=$	$s.name \mid e.name$	Variable

The Presentation Language $\mathcal{L}$

- Programs in $\mathcal{L}$ are *strict* term rewriting systems based on pattern matching.
- The rules in the programs are ordered from the top to the bottom to be matched.
- Two kinds of variables:
  - *s.*variables range over *symbols*,
  - *e.*variables range over the whole set of the expressions.
- For any rule $l \Rightarrow r$, any variable of r should appear in l.
- The parenthesis constructor $(\bullet)$ is used without a name.
- *Cons* constructor is used in infix notation and may be omitted.

Parameterized Synapse Cache Coherence Protocol

The initial value of counter *invalid* is parameterized, the other two counters are initialized by zero.

$$\begin{aligned}
&\text{(rh)} & dirty + valid \geq 1 \;\rightarrow\; .\\
&\text{(rm)} & invalid \geq 1 \;\rightarrow\; dirty' = 0, valid' = valid + 1, invalid' = invalid + dirty - 1\\
&\text{(wh1)} & dirty \geq 1 \;\rightarrow\; .\\
&\text{(wh2)} & valid \geq 1 \;\rightarrow\; valid' = 0, dirty' = 1, invalid' = invalid + dirty + valid - 1\\
&\text{(wm)} & invalid \geq 1 \;\rightarrow\; valid' = 0, dirty' = 1, invalid' = invalid + dirty + valid - 1
\end{aligned}$$

Any state reached by the protocol should **not** satisfy any of the two following properties:

(1) $invalid \geq 0, dirty \geq 1, valid \geq 1;$

(2) $invalid \geq 0, dirty \geq 2, valid \geq 0.$

Model of the Synapse Cache Coherence Protocol

Written in the Pseudocode (I)

```
Main( (e.time) : (e.is) ) ⇒ Loop( (e.time) : (Invalid I e.is) : (Dirty ) : (Valid ) );

Loop( ( [ ] ) : (Invalid e.is) : (Dirty e.ds) : (Valid e.vs) )
  ⇒ Test( (Invalid e.is) : (Dirty e.ds) : (Valid e.vs) );
Loop( (s.t : e.time) : (Invalid e.is) : (Dirty e.ds) : (Valid e.vs) )
  ⇒ Loop( (e.time) : Event( s.t : (Invalid e.is) : (Dirty e.ds) : (Valid e.vs) ) );

Event( rm : (Invalid I e.is) : (Dirty e.ds) : (Valid e.vs) )
  ⇒ (Invalid Append( (e.ds) : (e.is) )) : (Dirty ) : (Valid I e.vs);
Event( wh2 : (Invalid e.is) : (Dirty e.ds) : (Valid I e.vs) )
  ⇒ (Invalid Append( (e.vs) : (Append( (e.ds) : (e.is) )) )) : (Dirty I) : (Valid );
Event( wm : (Invalid I e.is) : (Dirty e.ds) : (Valid e.vs) )
  ⇒ (Invalid Append( (e.vs) : (Append( (e.ds) : (e.is) )) )) : (Dirty I) : (Valid );
   Append( ( [ ] ) : (e.ys) ) ⇒ e.ys;
   Append( (s.x : e.xs) : (e.ys) ) ⇒ s.x : Append( (e.xs) : (e.ys) );
```

Model of the Synapse Cache Coherence Protocol

Written in a Pseudocode (II)

$$Test(\ (Invalid\ e.is) : (Dirty\ I\ e.ds) : (Valid\ I\ e.vs)\ ) \Rightarrow False;$$
$$Test(\ (Invalid\ e.is) : (Dirty\ I\ I\ e.ds) : (Valid\ e.vs)\ ) \Rightarrow False;$$
$$Test(\ (Invalid\ e.is) : (Dirty\ e.ds) : (Valid\ e.vs)\ ) \Rightarrow True;$$

- This predicate tests the safety property.

The Cache Coherence Protocols Executed by Intermediate Interpreters of Turing Complete Languages (IITCL)

Consider a specializer *Spec* transforming programs written in a language $\mathcal{L}$.

Given an interpreter $Int_{\mathcal{M}}$ of a language $\mathcal{M}$ and a cache coherence protocol model specified in $\mathcal{M}$. Let $Int_{\mathcal{M}}$ be written in $\mathcal{L}$.

- Specialization of the following initial configurations is an attempt to verify the safety property of the protocol model indirectly.

  $$Int(\ (Call\ Main\ e.d),\ (Prog\ \underline{Synapse})\ )$$

  where the value of variable *e.d* is unknown, and <u>*Synapse*</u>, e. g., stands for the Synapse program model encoded in the data of language $\mathcal{L}$.

The Cache Coherence Protocols Executed by IITCL-s

Two aims:

- new powerful methods for the specialization, in order to verify the safety properties of the indirect models that are much more complicated as compared with the corresponding direct models;
- using the specializers for verifying the indirect protocol models specified in languages, which have no implemented specializers.

This report is devoted to these issues in the context of using the supercompiler SCP4.

The Cache Coherence Protocols Executed by IITCL-s

Using the supercompiler SCP4:

Verified: safety properties of the indirect models using a self-interpreter *Int* of a Turing-complete fragment of the SCP4 object language.

Proved: in an uniform way, several properties of the *Int* configurations generated by specilization of *Int* w.r.t. the direct models; these properties are crucial for removing the interpretation overheads.

Verified: safety properties of the indirect models using an interpreter of the Jones language WHILE.

Other Approaches

- In 1998 J. P. Gallagher et al. reported on a language-independent method for analyzing the imperative programs via intermediate interpretation by a logic programming language.
- Our interest in this task is in part inspired by a work done by De E. Angelis et al. (2014-2015) where this task was studied in the context of specialization of constraint logic programs.
  - They use external satisfiability modulo theories (SMT) solvers.

Comparison with the De E. Angelis et al. Approach

De E. Angelis et al.

- in terms of constraint logic programming;
- using external satisfiability modulo theories solvers;
- the presented transformation examples deal with neither function nor constructor application stack in the interpreted programs;

LN

- in terms of functional programming;
- self-sufficient methods for specialization of functional programs;
- the models include both the function call and constructor application stacks, the size of the first one is uniformly bounded on the input parameter while the second one is not;

Comparison with the De E. Angelis et al. Approach

The verification system VeriMAP

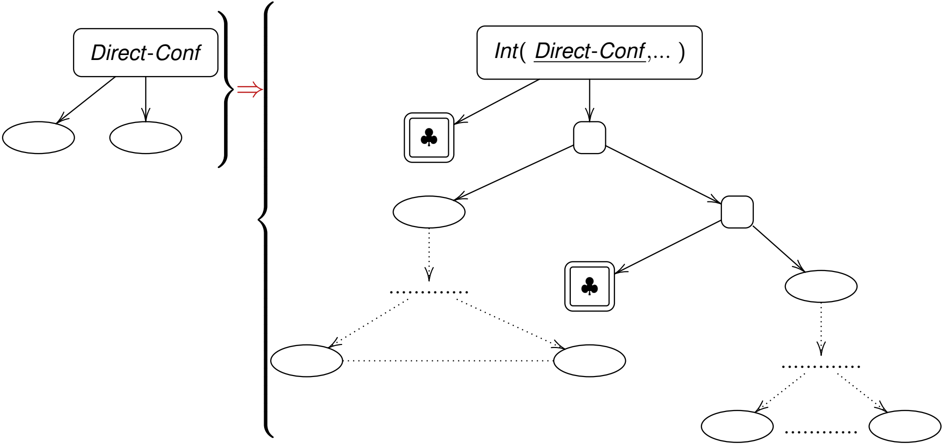
- uses nontrivial properties of integers recognized by both CLP built-in predicates and external SMT solvers.

The supercompiler SCP4

- uses a nontrivial property of the program configurations
- the property is the associativity of the built-in append function $++$ supported by the supercompiler SCP4 itself, rather than by an external solver

Interpreter Big-Step vs. Direct Model Small-Step

The overhead:



The configurations are tried to be generalized or folded inside this big-step.

The Main Problem

The Interpreting Pattern Matching

It produces many branching vertices.

The configurations in the vertices are subjects **to be**

- **generalized**
- **folded one by another**

If generalization of C_1 and C_2 or folding of C_2 by C_1 will happen when C_1 and C_2 are from the same big-step, then a recursive interpretation overhead appears in the residual program.

There is almost no hope of succeeding the indirect verification.

The Main Problem

The Interpreting Pattern Matching / Fragment

$$EvalCall(\, s.f, e.d, (Prog\ s.n)\,) \Rightarrow Matching(F, \texttt{[]}, LookFor(\ s.f, Prog(\ s.n\ )\ ), e.d\ );$$

$$\begin{array}{l}
Matching(\ F,\ e.old,\ ((e.p) : \texttt{'='} : (e.exp)) : e.def,\ e.d\ ) \\
\quad \Rightarrow Matching(\ Match(\ e.p,\ e.d,\ (\texttt{[]})\ ),\ e.exp,\ e.def,\ e.d\ ); \\
Matching(\ (e.env),\ e.exp,\ e.def,\ e.d\ ) \Rightarrow (e.env) : e.exp; \\
\quad Match(\ (Var\ \texttt{'e'}\ s.n),\ e.d,\ (e.env)\ ) \Rightarrow PutVar((Var\ \texttt{'e'}\ s.n) : e.d, (e.env)\ ); \\
\quad Match(\ (Var\ \texttt{'s'}\ s.n) : e.p,\ s.x\ :\ e.d,\ (e.env)\ ) \\
\qquad\qquad \Rightarrow Match(\ e.p,\ e.d,\ PutVar(\ (Var\ \texttt{'s'}\ s.n)\ :\ s.x,\ (e.env)\ )\ ); \\
\quad Match(\ (\texttt{'}\star\texttt{'}\ e.q) : e.p,\ (\texttt{'}\star\texttt{'}\ e.x) : e.d,\ (e.env)\ ) \\
\qquad\qquad \Rightarrow Match(\ e.p,\ e.d,\ Match(\ e.q,\ e.x,\ (e.env)\ )\ ); \\
\quad Match(\ s.x : e.p,\ s.x : e.d,\ (e.env)\ ) \Rightarrow Match(\ e.p,\ e.d,\ (e.env)\ ); \\
\quad Match(\ \texttt{[]},\ \texttt{[]},\ (e.env)\ ) \Rightarrow (e.env); \\
\quad Match(\ e.p,\ e.d,\ e.fail\ ) \Rightarrow F;
\end{array}$$

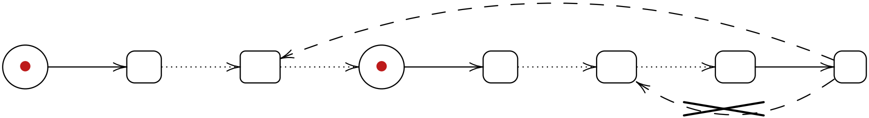
The Main Problem

The Interpreting Pattern Matching

Composition of the Turchin and the Higman-Kruskal relations

- forces **completely unfolding** the configurations with *Match* at the top of the application stack
  - **no generalization**
  - **no folding**

  inside any single big-step
  - – the big-steps' entry points



The Main Result

The Intermediate Self-Interpreter / Pattern Matching

Key Proposition

$\forall$ pattern p_0 s.t. $\forall$ variable v $\mu_v(p_0) < 2$ and $\forall$ passive expression d, the unfold-fold loop,

- starting off from configuration *Match*($\underline{p_0}$, d, ([ ]))
- and controlled by the Turchin-Higman-Kruskal composition,

results in a tree program s.t. $\forall$ non-transitive vertex in the tree is labeled by a config. of the form *Match*($\underline{p_i}$, d_i, (env_i)), ...

Where d, d_i are partial known data; ([ ]), (env_i) – environments.
$\mu_v(exp)$ denotes the multiplicity of variable v in *exp*.

- no generalization
- no folding

The Intermediate Self-Interpreter

Internal Big-Step Analysis

Given a big-step of the self-interpreter being specialized w.r.t. any given cache coherence protocol from the series of interest.

If no generalization was done before this big-step,
then simple corollaries of Key Proposition imply:

- no generalization
- no folding

may happen inside this single big-step of the self-interpreter.

That and several additional observations allow us to verify the safety properties of these indirect protocol models.

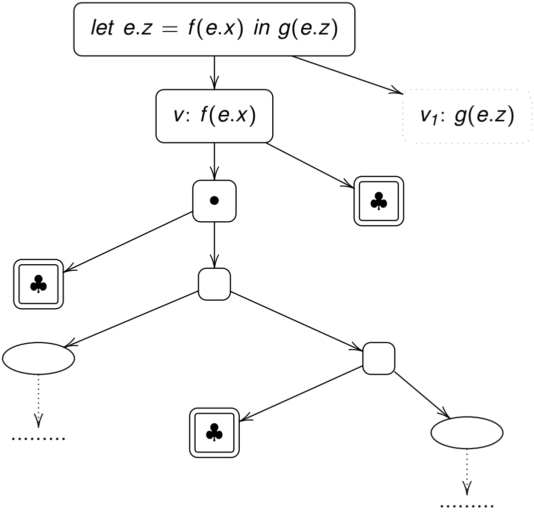
The Main Result

Internal Big-Step Analysis

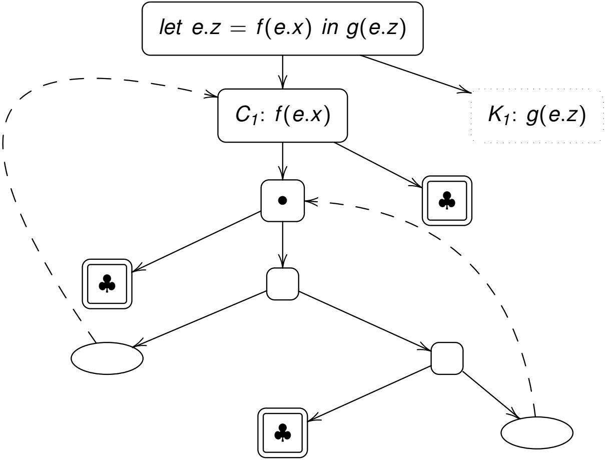
- Key Proposition
- **the proof of Key Proposition, given in an uniform way**

Key Proposition is applied only to the Synapse N+1 protocol, but can be applied for any protocol from the series of interest.

Unfolding



Folding



An intermediate state of an unfold-fold graph: the configuration K_1 still is not unfolded.

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)

A Variant of the Higman-Kruskal Relation

The term set $\mathcal{T}$ is a subset of $\mathbb{E}$ such that *term* $\in \mathcal{T}$ iff
term $::= (exp) \mid s.name \mid symbol$.

Definition

The homeomorphic embedding relation $\underline{\propto}$ is the smallest transitive relation on $\mathbb{E}$ satisfying the following properties, where f is an n-arity function, $\alpha, \beta, \tau, s, t, t_1, \ldots, t_n \in \mathbb{E}$ and $\alpha, \beta, \tau \in \mathcal{T}$.

- if x, y are variables of the same type, then $x \,\underline{\propto}\, y$
- $[\,] \,\underline{\propto}\, t,\ t \,\underline{\propto}\, t,\ t \,\underline{\propto}\, f(\ t_1,\ \ldots,\ t, \ldots, t_n\ ),\ t \,\underline{\propto}\, (t),\ t \,\underline{\propto}\, \alpha : t;$
- if $s \,\underline{\propto}\, t$ and $\alpha \,\underline{\propto}\, \beta$, then $(s) \,\underline{\propto}\, (t), \alpha : s \,\underline{\propto}\, \beta : t;$
- if $s \,\underline{\propto}\, t$, then $f(\ t_1,\ \ldots,\ s, \ldots, t_n\ ) \,\underline{\propto}\, f(\ t_1,\ \ldots,\ t, \ldots, t_n\ )$.

A Variant of the Higman-Kruskal Relation

- We use relation $\underline{\propto}$ modulo associativity of $++$ and the following equalities: $term : exp_1 = term \;++\; exp_1$, $exp \;++\; [\,] = exp$, $[\,] \;++\; exp = exp$.
- An additional restriction separating the basic cases of the induction from the regular ones:

for any symbol $\sigma.([\,]) \;\underline{\not\propto}\; (\sigma)$ and
for any symbol-variable $v.([\,]) \;\underline{\not\propto}\; (v)$

We impose this restriction on the relation $\underline{\propto}$ modulo the equalities above and denote the obtained relation as $\preccurlyeq$.

For any infinite sequence of expressions $t_1, \ldots, t_n, \ldots$ there exist two relation expressions t_i, t_j such that $t_i \preccurlyeq t_j$.

Configurations

$\mu_v(exp)$ denotes the multiplicity of variable *v* in *exp*.

Definition

A configuration is a finite sequence of the form

$\texttt{let}\ e.h = f_1(\ exp_{11},\ \ldots,\ exp_{1m}\ )\ \texttt{in}\ \ldots$
$\ldots\ \ \texttt{let}\ e.h = f_k(\ exp_{k1},\ \ldots,\ exp_{kj}\ )\ \texttt{in}\ \ exp_{n+1}$

where exp_{n+1} is passive, for all $i > 1$ $\mu_{e.h}(f_i(\ \ldots\ )) = \mu_{e.h}(exp_{n+1}) = 1$, and $\mu_{e.h}(f_1(\ \ldots\ )) = 0$; for all *i* and all *j* variable *e.h* does not occur in any function application being a sub-expression of exp_{ij}.

Configurations

$\texttt{let}\ e.h = f_1(\ exp_{11},\ \ldots,\ exp_{1m}\ )\ \texttt{in}\ \ldots$
$\ldots\ \texttt{let}\ e.h = f_k(\ exp_{k1},\ \ldots,\ exp_{kj}\ )\ \texttt{in}\ exp_{n+1}$

Since the value of $e.h$ is reassigned in each `let` in the stack we use the following presentation:

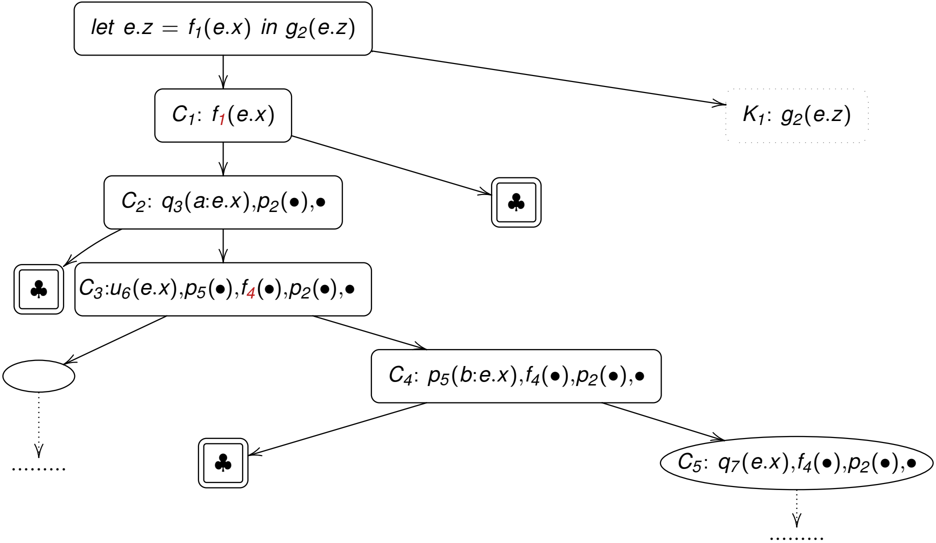
$$f_1(\ exp_{11},\ \ldots,\ exp_{1m}\ ),\ \ldots,\ f_k(\ exp_{k1},\ \ldots,\ exp_{kj}\ ),\ exp_{n+1}$$

- $e.h$ is replaced with placeholder $\bullet$
- the last expression may be omitted if it equals $\bullet$

Example

$$f(\ a : e.xs +\!\!+\ e.ys\ ),\ g(\ \bullet +\!\!+\ e.ys,\ (Var\ b\ c),\ \texttt{[]}\ ),$$
$$f(\ s.x : \bullet\ ),\ s.x : \bullet +\!\!+\ t(\ s.x : e.zs\ ),\ \bullet$$

Timed Configurations



Turchin's Relation

prefix$_i$ | context

C_i: $f^1_{t_1}(\dots), \dots, f^{l-1}_{t_{l-1}}(\dots),$ $f^l_{t_l}(\dots), \dots, f^k_{t_k}(\dots), exp_1$

prefix$_j$ | context

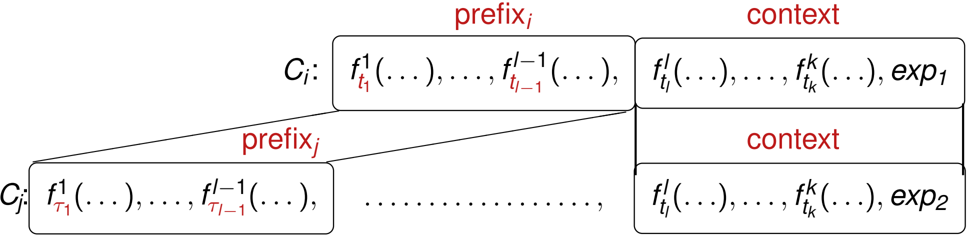
C_j: $g^1_{\tau_1}(\dots), \dots, g^{l-1}_{\tau_{l-1}}(\dots),$ $\dots\dots\dots\dots\dots\dots,$ $g^n_{\tau_n}(\dots), \dots, g^m_{\tau_m}(\dots), exp_2$

- $\forall s \,.\, (0 < s < l)\ f^s_{t_s} \simeq g^s_{\tau_s}$ (i.e., $f^s = g^s$);
- $t_{l-1} \neq \tau_{l-1}$;
- $f^l_{t_l} = g^n_{\tau_n}, f^{l+1}_{t_{l+1}} = g^{n+1}_{\tau_{n+1}}, \dots, f^k_{t_k} = g^m_{\tau_m}$
  (i.e., $f^l = g^n, \dots$ and $t_l = \tau_n, \dots$), where $k - l = m - n$.

We say that configurations C_i, C_j are in Turchin's relation $C_i \vartriangleleft C_j$.

For any infinite path $C_1, C_2, \dots, C_n, \dots$ there exist two timed configurations C_i, C_j such that $i < j$ and $C_i \vartriangleleft C_j$.

Turchin's Relation is not Transitive



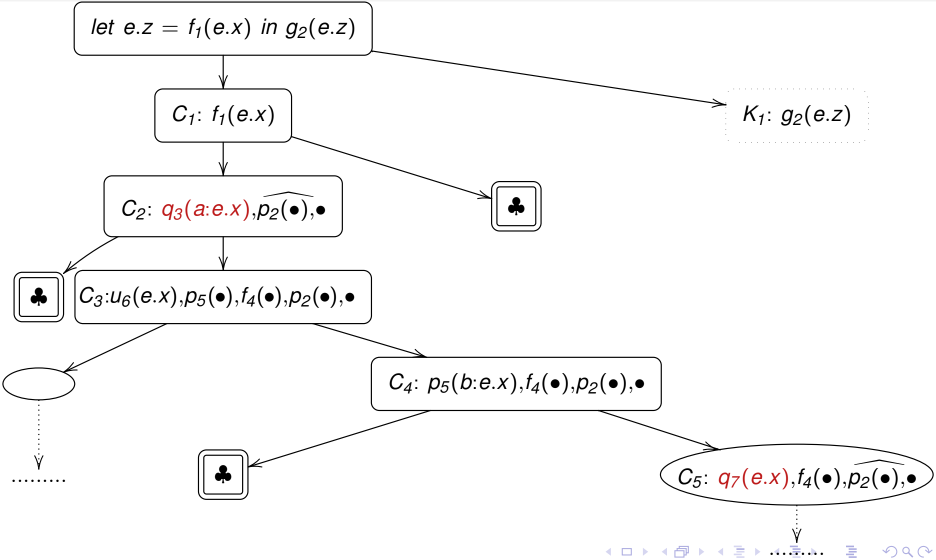
We say that configurations C_i, C_j are in Turchin's relation $C_i \triangleleft C_j$.

The idea:

- the function applications in the context never took a part in computing the configuration C_j, in this segment of the path;
- any function applications in the prefix of C_i took a part in computing the configuration C_j;

Turchin's Relation

Example: $C_2 \lhd C_5$



A Composition of the Turchin and the Variant of the Higman-Kruskal Relations

Given two timed configurations C_1, C_2 in a path.

- If $C_1 \triangleleft C_2$ does not hold, then the unfold-fold loop unfolds the current configuration C_2 and goes on.

A Composition of Relations $\lhd$ and $\preccurlyeq$

The Composition $\lhd \circ \preccurlyeq$ is a Well-Disordering

If $C_1 \lhd C_2$ holds, C_1, C_2 configurations are of the forms:

	prefix₁		*context₁*
C_1:	$f^1_{t_1}(\ldots), \ldots, f^{l-1}_{t_{l-1}}(\ldots),$		$f^l_{t_l}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_1$
	prefix₂		*context₂*
C_2:	$f^1_{\tau_1}(\ldots), \ldots, f^{l-1}_{\tau_{l-1}}(\ldots),$	$\ldots\ldots\ldots\ldots\ldots\ldots\ldots,$	$f^l_{t_l}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_2$

Compare the prefixes:

- if $\exists i.(1 \le i < l) \& f^i_{t_i}(\ \ldots\ ) \not\preccurlyeq f^i_{\tau_i}(\ \ldots\ )$, then C_2 is unfolded and the unfold-fold loop goes on,
- else C_1 is decomposed into *prefix₁* and *context₁*.

The Composition $\lhd \circ \preccurlyeq$

Folding and Generalization

If $C_1 \lhd C_2$ holds, C_1, C_2 configurations are of the forms:

$$C_1: \underbrace{f^1_{t_1}(\ldots), \ldots, f^{l-1}_{t_{l-1}}(\ldots),}_{prefix_1} \; \underbrace{f^l_{t_l}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_1}_{context_1}$$

$$C_2: \underbrace{f^1_{\tau_1}(\ldots), \ldots, f^{l-1}_{\tau_{l-1}}(\ldots),}_{prefix_2} \; \ldots\ldots\ldots\ldots\ldots, \; \underbrace{f^l_{t_l}(\ldots), \ldots, f^k_{t_k}(\ldots), exp_2}_{context_2}$$

C_1 is decomposed into *prefix*$_1$ and *context*$_1$:

- try to fold *prefix*$_2$ by *prefix*$_1$ and *context*$_2$ by *context*$_1$;
- if some of these attempts fail, then generalize the corresponding configurations.

The $\lhd \circ \preccurlyeq$-Strategy

Definition

A configuration is said to be a transitive configuration if one-step unfolding of the configuration results in a tree containing only the vertices with at most one outgoing edge.

For the sake of simplicity, the unfold-fold loop skips all transitive configurations encountered and removes them from the tree being unfolded.

- The unfold-fold loop is controlled by the $\lhd \circ \preccurlyeq$-strategy.

Conclusion

Using the supercompiler SCP4:

Verified: safety properties of the indirect models using a self-interpreter *Int* of a Turing-complete fragment of the SCP4 object language.

Proved: in an uniform way, several properties of the *Int* configurations generated by specilization of *Int* w.r.t. the direct models; these properties are crucial for removing the interpretation overheads.

Verified: safety properties of the indirect models using an interpreter of the Jones language WHILE.

Thank You

Some problems to investigate:

- description of suitable properties of interpreters to which our uniform reasoning demonstrated in this paper might be applied
- run time analysis