# From big-step to small-step semantics and back with interpreter specialisation

John P. Gallagher[1,2] Manuel Hermenegildo[2,3] Bishoksan Kafle[2]
Maximiliano Klemen[2,3] Pedro López-García[2,4] José Morales[2,3]

[1]Roskilde University, Denmark

[2]IMDEA Software Institute, Spain

[3]U. Politécnica de Madrid (UPM)

[4]Spanish Council for Sci. Research (CSIC)

VPT 2020, presented at VPT2021@ETAPS, 27 March 2021

**A Horn clause approach to verification of imperative programs**

Imperative programs

$\Downarrow$

Horn clauses expressing operational semantics

$\Downarrow$

Specialisation

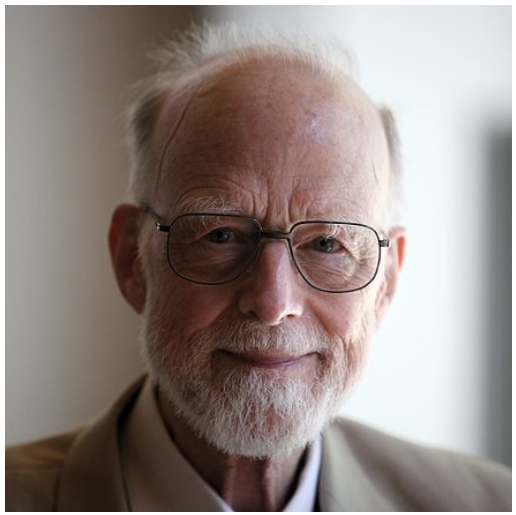
$\Downarrow$

Horn clause satisfiability checking

# The Hoare triple verification problem

To motivate the talk, consider verifying **a Hoare triple** $\{Pre\}\ s\ \{Post\}$.

This states that if *Pre* holds in the state before $s$ is executed then *Post* holds in the state after execution of $s$ (assuming $s$ terminates).



C.A.R. Hoare

# The program execution relation

To express the Hoare triple in Horn clauses, we define the execution of the program $s$ as a relation

$$\langle s, \sigma_0 \rangle \Longrightarrow \sigma_1$$

This means that if statement $s$ is executed in initial state $\sigma_0$, then $\sigma_1$ is the state after execution of $s$, assuming that the execution halts.

# The Hoare triple as a Horn clause

Then, if $\{Pre\}$ and $\{Post\}$ are represented as predicates *pre* and *post* on states, the Hoare triple is captured by the clause

$$pre(\sigma_0) \wedge \langle s, \sigma_0 \rangle \Longrightarrow \sigma_1 \rightarrow post(\sigma_1)$$

which is equivalent to

$$false \leftarrow pre(\sigma_0) \wedge \langle s, \sigma_0 \rangle \Longrightarrow \sigma_1 \wedge \neg post(\sigma_1)$$

This is satisfiable if and only if the Hoare triple holds.

# Operational semantics: big-step and small-step

The relation $\langle s, \sigma_0 \rangle \Longrightarrow \sigma_1$ can be formalised in different ways. The most widespread methods use operational semantics, following one of two styles.

- Natural semantics, often called big-step operational semantics [Kahn 1987].
- Structural operational semantics, often called small-step operational semantics [Plotkin 1981].

# Big-Step Semantics (Natural semantics)

In the big-step style, developed by Gilles Kahn in the 1980s, the execution relation is defined by structural decomposition of statements.

E.g. statement composition $s_1; s_2$

$$\frac{\langle s_1, \sigma_0 \rangle \Longrightarrow \sigma_1 \qquad \langle s_2, \sigma_1 \rangle \Longrightarrow \sigma_2}{\langle s_1 \,;\, s_2, \sigma_0 \rangle \Longrightarrow \sigma_2}$$

Gilles Kahn

# Small-Step Semantics (Structural operational semantics)

In the small-step style, formalised by Gordon Plotkin in 1981, the execution relation is defined as a chain or run of small steps $\langle s_0, \sigma_0 \rangle \Rightarrow \langle s_1, \sigma_1 \rangle$.

This means that $s_0$ is executed in state $\sigma_0$, leaving the remaining statement $s_1$ to be executed in state $\sigma_1$.

The complete execution is a chain ending with the `halt` statement.

$$\langle s_0, \sigma_0 \rangle \Longrightarrow \sigma_n \equiv \langle s_0, \sigma_0 \rangle \Rightarrow^* \langle \texttt{halt}, \sigma_n \rangle$$

Gordon Plotkin

# Small-Step Semantics (continued)

Small-step rules for statement composition are as follows.

$$\frac{\langle s_1, \sigma_0 \rangle \Rightarrow \langle \texttt{halt}, \sigma_1 \rangle}{\langle s_1 \,;\, s_2, \sigma_0 \rangle \Rightarrow \langle s_2, \sigma_1 \rangle} \qquad \frac{\langle s_1, \sigma_0 \rangle \Rightarrow \langle s_1', \sigma_1 \rangle}{\langle s_1 \,;\, s_2, \sigma_0 \rangle \Rightarrow \langle s_1' \,;\, s_2, \sigma_1 \rangle}$$

The first rule applies if $s_1$ completes execution in one step. The second rule applies if one step of the execution of $s_1$ leaves $s_1'$ remaining to be executed.

Note that small-step executions are linear, while big-step executions are tree-structured.

# Horn clause interpreters from semantic rules

The rules for both big-step and small-step semantics correspond closely to Horn clauses. A rule of the form

$$\frac{\alpha_1 \ \ldots \ \alpha_k}{\alpha_0} \qquad \text{if } \theta$$

is written as the Horn clause

$$\alpha_0 \leftarrow \theta \wedge \alpha_1 \wedge \ldots \wedge \alpha_k$$

where we assume that statements and states are encoded as first-order predicate logic terms.

The correspondence between natural semantics and Horn clauses was exploited by Kahn et al. in the Typol system.

# A big-step interpreter

The big-step interpreter for a subset of C including blocks and procedures is available at https://github.com/jpgallagher/Semantics4PE.

solve$(s, \sigma_0, \sigma_1, r, \Gamma)$ denotes $\Gamma \vdash \langle s, \sigma_0 \rangle \Longrightarrow \sigma_1, r$
where $\Gamma$ is the environment of procedure definitions and $r$ is a value returned by the statement $s$.

Clauses for assignment and statement composition

```
solve(asg(var(X),E),St,St2,Ret,Env) :-
   eval(E,St,St1,V,Env),
   save(X,V,St1,St2).
solve(seq(S1,S2),St,St2,Ret,Env) :-
   solve(S1,St,St1,Ret1,Env),
   solve(S2,St1,St2,Ret,Env).
```

# Interpreter specialisation

An interpreter specialised w.r.t. a given source program yields a translation of the source program into the language of the interpreter.

The 1st Futamura projection [Futamura 1971].

In our case, specialisation of the big-step Horn clause interpreter yields a translation from C into Horn clauses.

Yoshihiko Futamura

# Example translation by specialisation of big-step semantics

```
int n;

void f(int n){
  int x,a;
  x=n;
  a=1;
  while(x>0){
    x--;
    {
      int y;
      y=a;
      while(y>0){
        y--;a++;
      }
    }
  }
}

void main() {
  f(n);
}
```

```
main__1(A) :-
     f__2(A).
f__2(A) :-
     C is A, D is C, E is 1,
     while__9(D,E,F,G).
while__9(A,B,D,E) :-
     A>0, G is A-1, H is B,
     while__18(B,H,I,J),
     while__9(G,I,D,E).
while__9(A,B,A,B) :-
     A=<0.
while__18(A,B,E,F) :-
     B>0, H is B-1, I is A+1,
     while__18(I,H,E,F).
while__18(A,B,A,B) :-
     B=<0.
```

# Renaming and filtering interpreter predicates

We used LOGEN [Leuschel et al. 2006] to specialise the interpreter. It incorporates filtering to rename predicates and remove redundant structure. E.g. the initial call to the interpreter is

solve(call(main,[]),[(n,B)],[(n,C)],A, [[vardecl(var(n),int,null)],
function(f,[[vardecl(var(n),int,null)]],let(var(x),null,let(var(a),null,
seq(asg(var(x),var(n)),seq(asg(var(a),cns(nat(1))),
while(var(x)>cns(nat(0)),seq(asg(var(x),sub(var(x),cns(nat(1)))),
let(var(y),null,seq(asg(var(y),var(a)),while(var(y)>cns(nat(0)),
seq(asg(var(y),sub(var(y),cns(nat(1)))),
asg(var(a),add(var(a),cns(nat(1))))))))))))),
function(main,[],call(f,[var(n)]))])

which is renamed to solve__1(C,B,A), retaining only the variables in the call.

This is further renamed to main__1(A,B,C).

# Eliminating redundant state variables

A potential disadvantage: each big-step predicate has many arguments.

`solve_n`$(x_1, \ldots, x_n, x'_1, \ldots, x'_n, r)$, where $x_1, \ldots, x_n$ are the values of the variables in the input state, $x'_1, \ldots, x'_n$ are the values of the variables in the output state and $r$ is the statement outcome value.

We can apply standard logic programming transformation and analysis techniques to remove variables that are not affected in the transition.

Constraint strengthening [Kafle & Gallagher 2016] and redundant argument filtering [Leuschel & Sørensen 1996].

# Translation using a small-step interpreter

A similar translation based on a small-step semantic interpreter has been performed. [Peralta et al. 1998], [Henriksen et al. 2006, De Angelis et al. 2015].
The result consists of linear Horn clauses, due to the form of small-step rules.

`solve(S0,St0,S2,St2) :-` $\phi$`, solve(S1,St1,S2,St2).`

However, here we are going to take a different approach, namely to linearize the result of a big-step translation.

# From big-step to small-step with a linear interpreter

A linear interpreter executes one step at a time. It is related to a continuation-passing interpreter. It maintains a stack of goals to be executed.

```
% Interpretation of a user atom
solve([A|As]) :-
      clpClause(A,B),
      solveConstraints(B,B1),
      append(B1,As,As1),
      solve(As1).
```

```
% Interpretation of a constraint
solve([A|As]) :-
      constraint(A),
      call(A),
      solve(As).

% Base case
solve([]).
```

# Example big-step to small-step

Result of specialising the linear interpreter w.r.t. the previous example (using LOGEN) followed by redundant argument elimination.

```
f__2__3 :-
     B is A, C is B, D is 1,
     while__9__4(C,D).
while__9__4(A,B) :-
     A>0, E is A-1, F is B,
     while__18__5(B,F,E).
while__9__4(A,B) :-
     A=<0.
while__18__5(A,B,C) :-
     B>0, H is B-1, I is A+1,
     while__18__5(I,H,C).
while__18__5(A,B,C) :-
     B=<0,
     while__9__4(C,A).
```

Note that each clause is linear (unlike the big-step program).

We also rely on the fact that the stack is bounded, i.e. no recursive procedures in the program.

# Mixing big-step and small-step

When the program contains recursion, we can modify the linear interpreter, by adding the following clause.

```
solve([A|As]) :-
      recursiveCall(A),
      solve([A]),
      solve(As).
```

A recursive call is regarded as a single step. This is equivalent to forming a procedure summary for the recursive call.

# Comparison of big-step and small-step translations

| Big-step | Small-step |
| --- | --- |
| • Program structure is retained, nested loops etc. | • Linear clauses, program structure is not retained |
| • Clauses may be non-linear | • Each predicate has only inputs (and possible final output) |
| • Each predicate has both inputs and outputs | • Computation has linear path structure |
| • Computation has tree structure | |

For some analysis and verification problems (e.g. complexity analysis) retaining a compositional, nested program structure is important. For others (model checking, reachability analysis), a linear form is more appropriate.

# From small-step to big-step

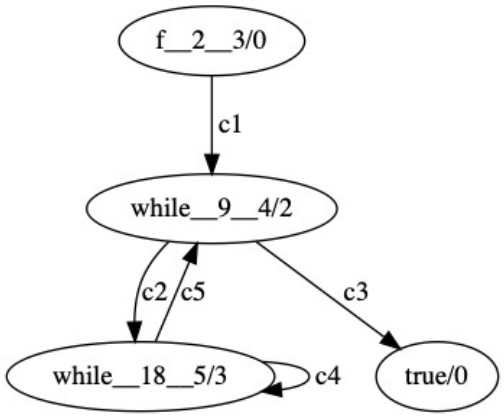
Given a linear program (e.g. a control-flow graph) it is sometimes desirable to reconstruct the nested loop structure of the program (related to the decompilation problem).

We now write another interpreter that performs this task, and again apply specialisation.

# Regular path expressions for linear programs

A linear program is a directed graph. The set of paths through the graph from entry to exit can be described by a regular expression.

R.E. Tarjan defined an algorithm that constructs a regular path expression from a given graph.




Path $c1(c2\ c4^*c5)^*c3$

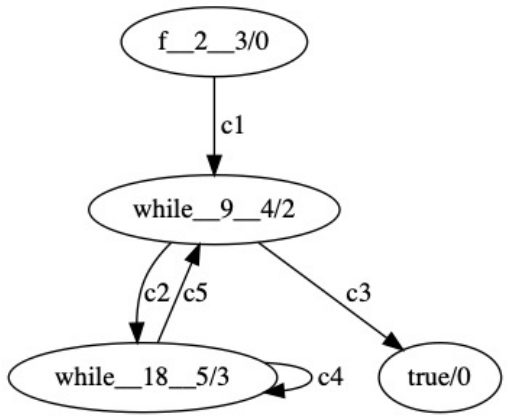
Robert E. Tarjan

# A linear path interpreter

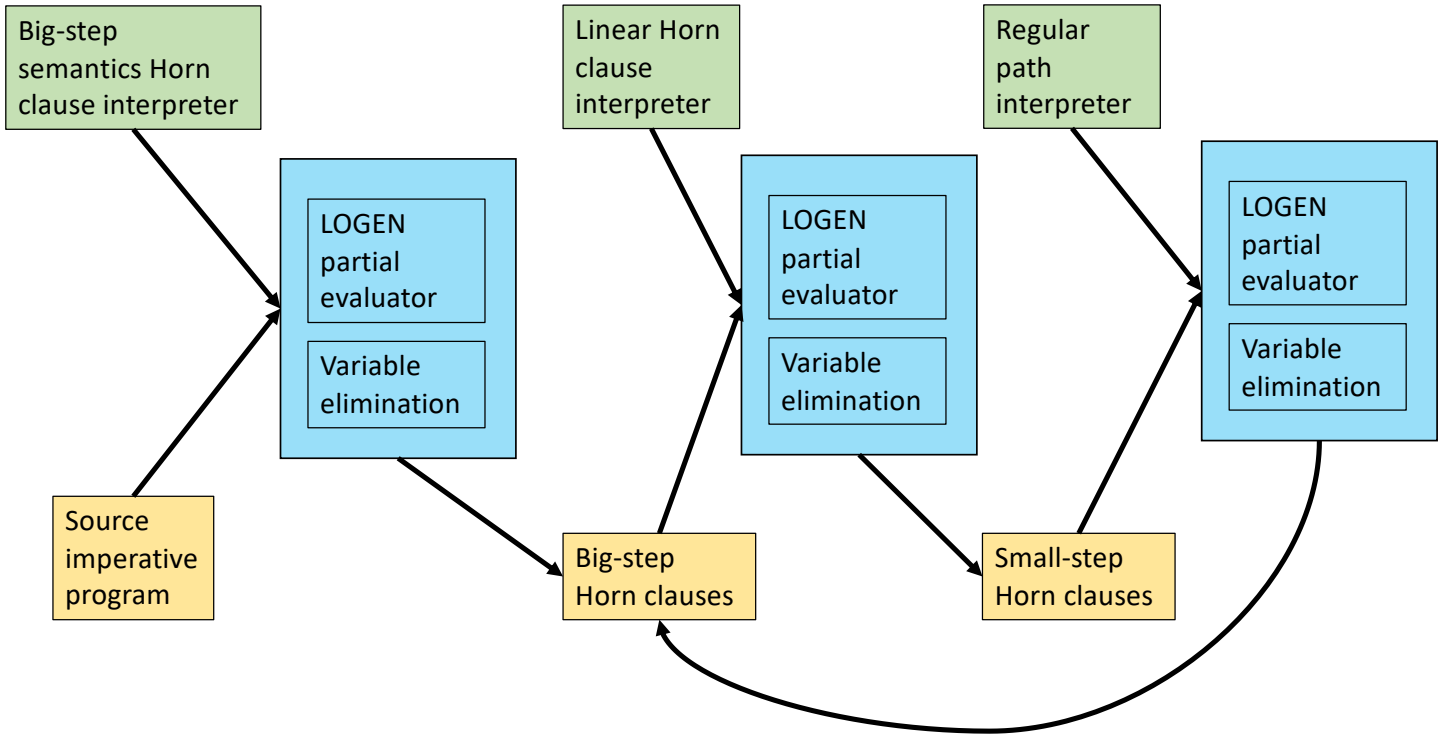
An interpreter for linear clauses guided by a regular expression. Each subexpression corresponds to a path from a start state to an end state.

The structure of the resulting specialised interpreter follows the nested structure of the regular expression.

```
% Example continued.
go__0(A) :-
   D=1,
   while__9__4__4(A,D,B,C),
   B=<0.
while__9__4__4(A,B,A,B) :-
   true.
while__9__4__4(A,B,C,D) :-
   A>0, A1 is A-1,
   while__18__5__9(B,B,A1,E,F,G),
   F=<0,
   while__9__4__4(G,E,C,D).
while__18__5__9(A,B,C,A,B,C) :-
   true.
while__18__5__9(A,B,C,D,E,F) :-
   B>0, A1 is A+1, B1 is B-1,
   while__18__5__9(A1,B1,C,D,E,F).
```



# Summary: big-step to small-step and back again



# Current work and next steps

- Specialisation of big-step semantics for Clight [Blazy & Leroy], with model of memory, pointers, arrays, ...
- Derive small-step from big-step semantics by specialising the big-step interpreter.
- Investigate the uses of path programs (see HCVS'21 talk).

Thank you!