From big-step to small-step semantics and back with interpreter specialisation (invited paper)

John P. Gallagher*
Roskilde University, Denmark
IMDEA Software Institute, Spain

Manuel Hermenegildo
IMDEA Software Institute, Spain

Bishoksan Kafle
IMDEA Software Institute, Spain

Maximiliano Klemen
IMDEA Software Institute, Spain

Pedro López García
IMDEA Software Institute, Spain

José Morales
IMDEA Software Institute, Spain

Abstract. We investigate representations of imperative programs as constrained Horn clauses. Starting from operational semantics transition rules, we proceed by writing interpreters as constrained Horn clause programs directly encoding the rules. We then specialise an interpreter with respect to a given source program to achieve a compilation of the source language to Horn clauses (an instance of the first Futamura projection). The process is described in detail for an interpreter for a subset of C, directly encoding the rules of big-step operational semantics for C. A similar translation based on small-step semantics could be carried out, but we show an approach to obtaining a small-step representation using a linear interpreter for big-step Horn clauses. This interpreter is again specialised to achieve the translation from big-step to small-step style. The linear small-step program can be transformed back to a big-step non-linear program using a third interpreter. A regular path expression is computed for the linear program using Tarjan's algorithm, and this regular expression then guides an interpreter to compute a program path. The transformation is realised by specialisation of the path interpreter. In all of the transformation phases, we use an established partial evaluator and exploit standard logic program transformation to remove redundant data structures and arguments in predicates and rename predicates to make clear their link to statements in the original source program.

1 Operational semantics

The operational semantics of a program defines the execution of the program as a run in a transition system. The rules of the transition system usually follow one of two styles, which are called *natural semantics* and *structural operational semantics* [32]. Both styles have roots in the early history of programming languages but were formally presented later; natural semantics (NS) was explicitly proposed by Kahn in the 1980s [22] and used to specify programming languages and type systems [6, 7]; the name was chosen to indicate an analogy with natural deduction. Structural operational semantics (SOS) was formulated by Plotkin in 1981 [35, 37], who later wrote an account of the origins of SOS [36].

In both NS and SOS approaches an imperative program P defines a relation $\sigma\langle P\rangle\sigma'$ where σ, σ' stand for the program state before and after execution of P respectively. This relation, closely related to a Hoare triple [17], can be formally specified by transition systems in different ways. In NS, transitions are of the form $\langle s, \sigma\rangle \Rightarrow \sigma'$, where s is a program statement, σ, σ' are states, and the transition means that s is completely executed in state σ, terminating in final state σ'. Thus NS is often called *big-step* semantics since the transition for a statement goes from the initial to the final state. By contrast, in SOS, often called *small-step* semantics, a transition has the form $\langle s, \sigma\rangle \Rightarrow \langle s', \sigma'\rangle$, which defines a single step that

*Email. `jpg@ruc.dk`

Submitted to:
VPT 2020

$$\frac{\langle s_1,\sigma\rangle \Rightarrow \sigma' \quad \langle s_2,\sigma'\rangle \Rightarrow \sigma''}{\langle s_1;s_2,\sigma\rangle \Rightarrow \sigma''} \qquad \frac{\langle s,\sigma\rangle \Rightarrow \sigma' \quad \langle \mathsf{while}\,(b)\,s,\sigma'\rangle \Rightarrow \sigma''}{\langle \mathsf{while}\,(b)\,s,\sigma\rangle \Rightarrow \sigma''} \quad \text{if } b \text{ is true in } \sigma$$

$$\frac{}{\langle \mathsf{while}\,(b)\,s,\sigma\rangle \Rightarrow \sigma} \quad \text{if } b \text{ is false in } \sigma$$

Figure 1: Examples of big-step rules for $s_1;s_2$ (left) and $\mathsf{while}\,(b)\,s$ (right).

$$\frac{\langle s_1,\sigma\rangle \Rightarrow \langle s_1',\sigma'\rangle}{\langle s_1;s_2,\sigma\rangle \Rightarrow \langle s_1';s_2,\sigma'\rangle} \qquad \frac{}{\langle \mathsf{while}\,(b)\,s,\sigma\rangle \Rightarrow \langle s;\mathsf{while}\,(b)\,s,\sigma\rangle} \quad \text{if } b \text{ is true in } \sigma$$

$$\frac{\langle s_1,\sigma\rangle \Rightarrow \sigma'}{\langle s_1;s_2,\sigma\rangle \Rightarrow \langle s_2,\sigma'\rangle} \qquad \frac{}{\langle \mathsf{while}\,(b)\,s,\sigma\rangle \Rightarrow \sigma} \quad \text{if } b \text{ is false in } \sigma$$

Figure 2: Examples of small-step rules for $s_1;s_2$ (left) and $\mathsf{while}\,(b)\,s$ (right).

moves from s in state σ to the *next* statement s' and next state σ'. We also have a transition $\langle s,\sigma\rangle \Rightarrow \sigma'$ for the case that s terminates in one step. A computation is defined as a chain of small steps.

We will use the nicknames *big-step* and *small-step* for the rest of the paper, since they capture the essential difference between NS and SOS.

The rules for big-step transitions follow the syntactic structure of statements, and the transition for a statement is defined in terms of transitions of its immediate components. Consider for example rules for transitions for the statements $s_1;s_2$ and $\mathsf{while}\,(b)\,s$ shown in Figure 1. (Somewhat more elaborate and realistic rules are used in Section 3). A rule is read as stating that the transition below the line holds if the conditions above the line, along with any side-conditions, hold.

Transitions in small-step semantics define individual computation steps from one statement-state pair to the next. Consider again the statements $s_1;s_2$ and $\mathsf{while}\,(b)\,s$, the small-step rules for which are shown in Figure 2.

A complete computation for the execution of a program P with initial state σ_0 is constructed using big-step semantics by finding a derivation, using big-step rules, of the transition $\langle P,\sigma_0\rangle \Rightarrow \sigma_n$, where σ_n is the final state. On the other hand, in the small-step semantics, execution of P is modelled by constructing a run of the form (where $s_0 = P$):

$$\langle s_0,\sigma_0\rangle \Rightarrow \langle s_1,\sigma_1\rangle \Rightarrow \langle s_2,\sigma_2\rangle \Rightarrow \cdots \Rightarrow \sigma_n$$

where each step $\langle s_i,\sigma_i\rangle \Rightarrow \langle s_{i+1},\sigma_{i+1}\rangle$ and the final step $\langle s_{n-1},\sigma_{n-1}\rangle \Rightarrow \sigma_n$ is derivable from the small-step transition rules. This run is also denoted $\langle s_0,\sigma_0\rangle \Rightarrow^* \sigma_n$.

2 Interpreters constructed from semantic rules

The transition rules of operational semantics, both big- and small-step, such as those exemplified above, have the following form.

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

where $\alpha_0,\alpha_1,\ldots,\alpha_k$ are atomic statements about transitions, and θ is a side-condition, which is assumed to consist of simple guards or subsidiary operations evaluated as part of the rule. Such a rule is read as an implication $\forall(\theta \wedge \alpha_1 \wedge \ldots \wedge \alpha_k \rightarrow \alpha_0)$, which is a Horn clause. The close correspondence between big-step rules and Horn clauses was noted by Kahn [22]. For instance, the first big-step rule for $s;\mathsf{while}\,(b)\,s$ shown in Figure 1, is written as the following Horn clause.

$$\forall b, s, \sigma, \sigma', \sigma''.(b \text{ is true in } \sigma \wedge \langle s, \sigma\rangle \Rightarrow \sigma' \wedge \langle \mathsf{while}\,(b)\,s, \sigma'\rangle \Rightarrow \sigma'' \rightarrow \langle \mathsf{while}\,(b)\,s, \sigma\rangle \Rightarrow \sigma'')$$

Naturally, the correspondence between transition rules and Horn clauses assumes that the statements and states are represented as first-order terms. A front end parser generates a suitable term representation for the source program (an abstract syntax tree) while program states are represented (for example) by a list or tree data structures in which the values of program variables are recorded. We will see a detailed example of program and state representation in Section 3. We will also write atomic statements using typical predicate-argument form, when writing rules as Horn clauses. For instance the rule above might be written as follows (omitting the universal quantifier, writing the implication $\theta \wedge \alpha_1 \wedge \ldots \wedge \alpha_k \rightarrow \alpha_0$ as $\alpha_0 :- \theta \wedge \alpha_1 \wedge \ldots \wedge \alpha_k$, and using comma in place of the conjunction symbol).

```
transition(while(B, S), St, St2) :-
    eval(B, St, true),
    transition(S, St, St1),
    transition(while(B, S), St1, St2).
```

The correspondence between natural semantics and Horn clauses was identified and used in the Typol system [6] to implement executable versions of semantic specifications as logic programs. The set of Horn clauses derived from the transition rules, together with definitions for the subsidiary predicates, constitutes an executable interpreter in a logic programming system. In our setting, given a program `p` and an initial state `s0` (in some suitable representation as variable-free terms), the query `transition(p,s0,X)` can be run to compute the state `X` resulting from executing `p` in state `s0`. Executability is not our main goal, however; the interpreter is a step in the procedure to transform C programs to Horn clauses, but it is useful to be able to test the interpreter by executing programs.

Using such an interpreter, for either big-step or small-step transitions, a translation from the imperative source code to Horn clauses is achieved using *partial evaluation*. This is an instance of the first Futamura projection [8]; an interpreter specialised (by partial evaluation) with respect to a source program can be seen as a compilation of the source program into the language of the interpreter, which in our case is Horn clauses.

In Section 3 we will describe an interpreter for a subset of C, based directly on big-step semantics. An interpreter could be written based on small-semantics in a similar way.

3 A big-step interpreter for a subset of C

Consider a semantics for a subset of C, with the abstract syntax shown in Figure 3. We use a C parser to read a source program and produce an abstract syntax tree (AST) (see example in Figure 4). A program consists of a list of function declarations and global variable declarations. There is exactly one function called `main`, a call to which is the entry to the program.

The complete interpreter can be found in [9](bigstep.pl). It can be run in Ciao Prolog [15] and in other Prolog systems with minor modifications. The most important predicates are the following.

```
eval(Expr, St0, St1, V, Env)
solve(S, St0, St1, Ret, Env)
```

In both predicates, the argument `Env` is the function environment, which is the list of global variable and function declarations. This remains constant during program execution. The arguments `St0` and `St1`

program	$\rightarrow$	*decl*$*$
decl	$\rightarrow$	*vardecl* $\mid$ *fdecl*
fdecl	$\rightarrow$	$\mathsf{function}(id, vardecl*, stm)$
var	$\rightarrow$	$\mathsf{var}(id)$
value	$\rightarrow$	$\mathsf{null} \mid n$
expr	$\rightarrow$	$\mathsf{var}(expr) \mid \mathsf{cns}(num) \mid \mathsf{add}(expr, expr) \mid \mathsf{call}(id, expr*) \mid \mathsf{div}(expr, expr) \mid$ $\mathsf{logicaland}(expr, expr) \mid \mathsf{mul}(expr, expr) \mid \mathsf{sub}(expr, expr) \mid \mathsf{not}(expr) \mid$ $expr < expr \mid expr \leq expr \mid expr == expr \mid expr > expr \mid expr \geq expr$
stm	$\rightarrow$	$\mathsf{skip} \mid \mathsf{ret}(expr) \mid \mathsf{ret} \mid \mathsf{asg}(\mathsf{var}(id), expr) \mid \mathsf{block}(localdecl*, stm) \mid \mathsf{call}(id, expr*) \mid$ $\mathsf{seq}(stm, stm) \mid \mathsf{while}(expr, stm) \mid \mathsf{ifthenelse}(expr, stm, stm) \mid$ $\mathsf{let}(var, expr, stm) \mid \mathsf{for}(stm, expr, stm, stm)$
vardecl	$\rightarrow$	$\mathsf{vardecl}(var, type, expr)$
localdecl	$\rightarrow$	$\mathsf{decl}(var, expr)$
type	$\rightarrow$	int
num	$\rightarrow$	$\mathsf{nat}(n)$

Figure 3: Abstract syntax of source programs.

represent memory states. They are lists of elements of the form `(V,Val)` where `V` is a program source variable name, and `Val` is its value in the state (an integer).

The predicate `eval(Expr,St0,St1,V,Env)` means that expression `Expr` is evaluated in state `St0` and yields value `V` and updated state `St1`. (Evaluation of an expression can cause a state change, if a function is called within the expression). The predicate `solve(S,St0,St1,Ret,Env)` means that the statement `S` is executed in initial state `St0`, and terminates with final state `St1` and *statement outcome* `Ret` (this construct is inspired by the big-step semantics for the Clight subset of the C language by Blazy and Leroy [2], and can be used in future extensions of the interpreter for handling break, continue and return statements).

Utility predicates in the interpreter include those for retrieving and updating values in the state, for extending the state when local variables are introduced in blocks, and for matching function parameters.

3.1 Specialising the big-step interpreter

Given the representation of a program, namely a set of declarations of functions and global variables *env*, a *partial evaluator* is used to specialise the interpreter. The call to the interpreter is

`solve(call(main,`*args*`),` *st*0`, St1, Ret,` *env*`)`

where *args* are the arguments of the `main` function in *env*, and *st*0 is the initial state consisting of the values (possibly undefined) of the global variables declared in *env*. We use the partial evaluator Logen [25, 26]. This is an offline partial evaluator, which means that the program to be specialised (the big-step interpreter) is first annotated to indicate which parts will be evaluated and which parts will be retained in the specialised program. For the interpreter, we annotate all calls to `solve` as being retained in the specialised program, and all other calls except operations on undefined values from the state are evaluated.

Logen also requires a *filter* to be defined for each retained predicate (in our case `solve`). The filter declares which parts of a call to the predicate are known and unknown during partial evaluation. This

```
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);
}
```

```
[[
  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)]) )]
```

Figure 4: Source program (left) and its abstract syntax tree (right).

information is used both to rename specialised predicates, and to abstract away parts of calls declared to be unknown. The filter declaration for `solve` is as follows; it uses a *binding type* called `store` and the standard binding types `static` (known) and `dynamic` (unknown).

```
:- type store--->(type list(struct(',',[static,dynamic]))).
:- filter solve(static,(type store),(type store),dynamic, static).
```

The binding type `store` declares lists of pairs of binding type `(static,dynamic)`. This describes program states containing variable-value pairs in which the variable name is known but the value is unknown. The filter declaration for `solve` thus states that the statement and environment arguments are known, while the state arguments consist of lists describing the unknown values of a set of known variables. The statement outcome argument is unknown. More information on binding types and filters can be found in the references to Logen.

3.1.1 Renaming specialised predicates

Logen requires the program to be annotated such that calls that arise during partial evaluation generate only a finite number of different values of the known parts of the filter declaration. This guarantees that only a finite number of different calls arise and partial evaluation terminates. For each call, a renamed version is generated, whose arguments consist only of the `dynamic` parts of the call. In the case of `solve`, the arguments of specialised versions are thus the values of variables in the state, and the statement outcome.

For example, the initial call to `solve` for the program in Figure 4 is renamed as follows.

```
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)]))]) ⟹ solve__1(C,B,A).
```

The renamed predicate `solve__1(C,B,A)` has three arguments corresponding to the dynamic values in the call.

3.1.2 Further renaming and unfolding

The specialised interpreter produced by Logen contains predicates of the form `solve__n(...)`. In order to generate more informative predicate names, a post-processing step is performed on the output of Logen. Using the renaming table (which can optionally be output by Logen), it is possible to retrieve the first argument (that is, the statement) of the call for which `solve__n(...)` is a renamed version. We then use the statement type to construct a new name, reusing the index n so as to ensure a unique predicate name. In this way, the predicate `solve__1` shown above is renamed to `main__1`, and a call of the form `solve__n(....)` where `solve__n` is a renaming of `solve(while(...), ...)` is renamed as `while__n`, and so on.

A further post-processing step is performed, unfolding calls to basic statements such as assignments, statement sequences (semicolon) and local variable declarations.

3.1.3 Eliminating redundant state arguments

The predicates generated by Logen, with filter described above, represent big steps and have the form `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. However, a statement typically only affects some variables of the state. The values of the unaffected variables are just "passed through" from initial to final state. That is, $x_i = x'_i$ holds for the variables that are unaffected.

For instance, for the statement `if (x>y) x=x+1; else y=y-1;` where x,y,z and w are the variables in scope, then the big-step predicate would be `solve__n(X,Y,Z,W,X',Y',Z',W',R)` with definition:

```
solve__n(X,Y,Z,W,X',Y',Z',W',R) :- X<Y, X'=X+1, Y'=Y,Z'=Z,W'=W.
solve__n(X,Y,Z,W,X',Y',Z',W',R) :- X≥Y, X'=X, Y'=Y-1,Z'=Z,W'=W.
```

Thus only the values of x and y are affected by the statement, while the values of z and w pass through the transition unchanged. Furthermore, the value of `R` is not assigned.

We apply known logic program analysis and transformation techniques to eliminate some variables from predicates. Firstly we use an abstract interpretation to produce a strengthened or *more specific* program [30, 19]. In the example above, we can prove that the constraints `Z'=Z,W'=W` can safely be conjoined to all calls to `solve__n` and the arguments `Z',W'` removed from the call, resulting in the equivalent call `solve__n(X,Y,Z,W,X',Y',R),Z'=Z,W'=W.` .

Having strengthened the clauses, we apply the *redundant argument filtering* algorithms [27] to remove arguments from predicates. This technique was previously used to improve the result of interpreter specialisation [14], where the relationship between argument filtering and liveness analysis was shown.

The removal of the redundant variables from the transitions simplifies the clauses, makes them more readable and can reduce the complexity of analyses, since the complexity of some constraint operations is affected by the number of variables in the constraints.

Example 1 *Let the source program be the program in Figure 4. Using a big-step semantics interpreter, we obtain the following clauses by partial evaluation, followed by predicate renaming and redundant argument removal as described above.*

```
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.
```

A feature of clauses generated from the big-step interpreter is that the source program's statement nesting is preserved. Thus the inner while loop corresponds to `while__18`*, which is called from within the outer while loop represented by* `while__9`*. Also, note that although variables x, y, a and n are all in scope when the inner loop of the program is reached, the predicate* `while__18` *has only 4 arguments, a reduction from the 9 arguments that are generated from the interpreter (4 variables each for input and output states, plus the statement outcome).*

3.2 Correctness of the translation

In summary, specialisation of the big-step interpreter yields a translation from the source program to Horn clauses where there is a clear relationship between the predicates and the source code, and where the predicate arguments contain only the values affected by the statement represented by the predicate, rather than the whole state as is the case with some other translations. The whole process, including the parsing of the input C program, has been automated.

Correctness of the translation follows from the correctness of both the interpreter and the specialiser. The interpreter can be obtained directly from a formal definition of the imperative language semantics as described in Section 2, thus giving confidence in its correctness. In our experiments we use an established tool for specialisation of Horn clauses, namely the Logen partial evaluator, whose correctness follows from established theory of fold-unfold transformations applied to Horn clauses [34]. In addition we have applied semantics-preserving logic program transformations whose correctness has been proved.

4 Small-step semantics and linear clauses

A similar procedure could be followed to translate imperative programs into constrained Horn clauses, specialising an interpreter for small-step semantics. Such an interpreter and translation was described in [33] and [14].

```
solve([A|As]) :-
      clpClause(A,B),
      solveConstraints(B,B1),
      append(B1,As,As1),
      solve(As1).
```

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

Figure 5: Clauses from a linear interpreter.

Small-step semantics is associated with *linearity*; when we specialise an interpreter for the rules of small-step semantics, we expect to get linear clauses, which are Horn clauses consisting of at most one non-constraint body atom. The clauses would have the form $p(\bar{x}) :- \theta(\bar{x},\bar{y}), q(\bar{y})$ representing one computation step, where $\bar{x}$ and $\bar{y}$ are the values of state variables before and after the step, and $\theta(\bar{x},\bar{y})$ is a constraint relating the values. The final transition is a clause $p(\bar{x}) :- \theta(\bar{x})$.

The clauses obtained from specialising the big-step interpreter are not in general linear, a fact illustrated by the clauses in Example 1. In this section, we show how to linearise big-step clauses, obtaining clauses that correspond directly to a small-step semantics for the program. This is an alternative to writing small-step semantics for the source language. In other words, we write just one semantics, namely big-step semantics, and then automatically obtain translations into Horn clauses corresponding to big- and small-step semantics.

4.1 A linear interpreter

Linear resolution with a fixed selection rule is known to be a complete proof method for Horn clauses [28]. This can be thought of as providing a small-step semantics for Horn clauses. We proceed as follows: we write a linear resolution interpreter for Horn clauses and then specialise it using Logen, with similar post-processing operations as described in Section 3.

Figure 5 shows the main clauses from a linear interpreter. We assume that the clauses being interpreted are included as facts in the interpreter of the form `clpClause(A,B)`, where `A` is the clause head and `B` is the body. The predicate `solve([A|As])` has as argument a list of atoms representing a conjunction to be proved. One step in the interpreter consists of picking the leftmost atom `A`, and either solving it if `A` is a constraint, or else resolving `A` with the head of a clause, appending the body of that clause to the remaining atoms `As`. The computation is completed when the argument of `solve` is empty.

The full linear interpreter can be found in [9](linearSolve.pl). Compared with the clauses in Figure 5, it contains an additional mechanism to encode the argument of `solve` in a way that records repeated variables. Much of the interpreter is concerned with this encoding and decoding. This is only for the purpose of improving the output of Logen, and is in fact rendered unnecessary by subsequent processing to eliminate redundant arguments, as described in Section 3.

4.2 Specialising the linear interpreter

We proceed to translate an arbitrary set of clauses P into linear form by specialising the linear interpreter with respect to P. The translation to linear form has previously been exploited in logic programming [5]. In specialising the interpreter, a key decision is how to define the filter for `solve`. If we can determine in advance that the length of conjunctions is bounded (and thus the length of the argument to `solve` is bounded), then the filter is defined so that only the arguments of atoms in the conjunction are dynamic. If the size of conjunctions is bounded, this is sufficient to ensure that only a finite number of distinct calls to `solve` arise, and partial evaluation terminates. On the other hand, if the size of the conjunction is unbounded, the filter has to be defined so that the whole conjunction is dynamic, and much specialisation will thereby be lost. We return to this point below.

The question of whether the size of the conjunction is bounded is related to the *tree dimension* of the clauses being interpreted by the linear interpreter [20]. In the case of the linear interpreter, where the leftmost atom is selected at each step, the problem amounts to determining whether there are non-tail-recursive predicates in the clauses.

Assuming that the conjunction is bounded, specialisation terminates and returns a set of linear clauses, whose arguments correspond to arguments of the original source predicates.

Example 2 *Consider the clauses produced by specialising the big-step interpreter in Example 1. After specialisation of the linear interpreter with respect to those clauses, followed by elimination of redundant arguments as previously described, we obtain the following linear clauses representing the function* `f`*.*

```
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 the predicates representing the nested loops, namely `while__9__4` *and* `while__18__5` *are mutually recursive instead of being nested as in the original clauses. Furthermore, the predicates do not directly produce output; the final state of the computation is represented in the variables at the point where* `while__9__4` *is called and terminates.*

4.3 Handling unbounded conjunctions

There are two approaches to handling the application of the linear interpreter to a set of clauses in which the size of the conjunction has no upper bound. One is to represent the conjunction explicitly. This would result in a specialised version of the linear interpreter in which the list argument of `solve` remains. For example, consider the clauses for the Fibonacci function, which is binary recursive. Specialising the linear interpreter results in linear clauses as follows.

```
solve([fib(0,0)|As]) :- solve(As).
solve([fib(1,1)|As]) :- solve(As).
solve([fib(N,M)|As]) :-
        N>1,N1 is N-1,N2 is N-2,
        solve([fib(N1,M1),fib(N2,M2),M is M1+M2|As]).
solve([M is M1+M2|As]) :-
        M is M1+M2,
        solve(As).
```

Although formally a linear set of clauses, the list structure presents difficulties for the purposes of verification and analysis.

The other approach is to mix big-step and small-step semantics. A recursive function call is handled by a big step, often called a procedure summary in the literature, defined by a predicate that represents both input and output states. This is also the approach followed in translations to Horn clauses from compiler intermediate representations such as a control flow graph, where function calls are represented as an edge in the control flow graph, but the code for the function itself is in another graph. Examples of this approach are [31, 11, 10] and the Horn clause verification tools [13, 23]. De Angelis *et al.* [4] also derived a form that they call multi-step semantics by interpreter specialisation, allowing for recursive function calls, that is essentially the same.

Our proposal for realising this approach is simply to extend the linear interpreter with an extra non-linear clause for `solve`, handling recursive functions. It is assumed that the first clause in Figure 5 has an added condition so that it is applied only to predicates that do not represent recursive function calls.

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

This is no longer a linear interpreter, since there are two calls to `solve` in the body of the clause. However, specialising an interpreter including this clause ensures that the conjunction in calls to `solve` is bounded, and the code apart from the recursive calls is linearised. If the original C program input to the big-step interpreter was a single procedure, then specialising the resulting big-step clauses results in linear clauses as the clause above is never used. Tail-recursive functions will also result in linear clauses since the above clause will only be called when the tail of the conjunction `As` is empty and `solve(As)` is evaluated to `true`.

5 A path interpreter: from linear clauses to big steps

In this section we take a set of linear clauses and transform them to non-linear clauses that reflect the nested call structure of the clauses. Consider the result of linearising our running example, shown in Example 2. We noted that the predicates `while__9__4` and `while__18__5` are mutually recursive. However, when analysing this program, for example to perform resource or termination analysis, it might be an advantage to identify that `while__18__5` is the inner loop, which can be analysed on its own, and then its solution applied to the analysis of the outer loop `while__9__4`. Though in our running example we started with a big-step representation, we might have obtained the linear clauses from some less structured source code such as machine language or control flow graphs, and in such cases it is often useful to be able to reconstruct big-step clauses.

5.1 Regular path expressions

A set of linear clauses induces a directed graph in which the nodes are predicate names and there is an edge from p to q if there is a clause with head predicate p and a call to q in the body. The edge is labelled by an identifier for the clause. The call graph for the clauses in Example 2 is shown in Figure 6. For any graph with a designated entry node and exit node, there is a well known algorithm by Tarjan [38] that computes a regular path expression describing exactly the paths from the entry to exit. The alphabet of the regular expression is the set of clause identifiers. There are usually many equivalent path expressions that describe this set of paths; non-deterministic choices within the algorithm determine which expression is produced. For the graph in Figure 6, with entry node `f__2__3` and exit `true`, one regular path expression is $c1(c2\ c5^{*}c4)^{*}c3$ (where $c1 \ldots c5$ are the clause identifiers in the order they appear above). From this expression, it can be seen that the inner loop (clause $c5$) is nested inside the outer loop, since the structure of the regular expression is nested.

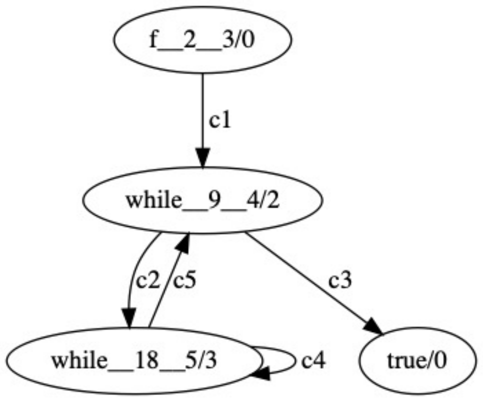

Figure 6: Predicate call-graph for the clauses in Example 2.

5.2 A path expression interpreter

Figure 7 shows the main clause of an interpreter for a set of linear clauses that follows a computation path given by a regular expression. The predicate `pathsolve(A,Z,Expr,F)` finds a path from atom `A` to atom `Z` that is recognised by the path expression `Expr`. The final argument `F` is a set of symbols that can follow a path described by `Expr`, and is present only to aid the partial evaluator. It can immediately be seen that the clauses for the cases of concatenation `E1:E2` and repetition `star(E)` are nonlinear clauses. The complete interpreter, which includes the option of calling Tarjan's algorithm, is available from [9](solveReg.pl).

Specialisation of the path interpreter with respect to a set of linear clauses and a path expression for paths from entry to `true` computed by Tarjan's algorithm results in a "path program". It has the characteristics of a big-step program in that each predicate has an input and output state (the start and end points of the subpath that is represented by that predicate). It is in general non-linear since any star expressions or path concatenations will result in non-linear clauses.

Example 3 *Consider the linear clauses resulting from the specialisation of the linear interpreter in Example 2. Computing a path expression* $c1(c2\ c5^{*}c4)^{*}c3$ *using Tarjan's algorithm (where* $c1,\ldots,c5$ *identify the clauses in the order they appear), and then specialising the path interpreter with respect to the linear clauses and the expression yields the following clauses.*

```
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).
```

The structure of the clauses resembles those of the big-step clauses in Example 1. The inner loop, now represented by `while__18__5__9` is nested in the outer loop `while__9__4__4`. However the path program has some minor differences from the original big-step clauses; the base case of a loop arising from a "star" expression is the empty path, while any base case conditions in the original clauses now appear conjoined immediately after the star expression predicate. For example, the condition `F=<0` appears after the loop predicate `while__18__5__9(B,B,A1,E,F,G)` instead of in the base case of that predicate.

6 Discussion

Semantics-based translation of imperative programs into Horn clauses is a topic of growing importance, as Horn clause solvers become more powerful and effective [3, 13, 23, 21, 18], providing a general framework for imperative program verification [33, 31, 12, 1] and enabling the application of the large body of techniques and tools for semantics-based analysis and verification developed in the (constraint) logic programming field (e.g. [16]). These include abstract interpreters for a variety of domains including non-functional program properties such as complexity and resource usage, e.g.[29].

Big-step based interpretation using Horn clause interpreters was proposed by Kahn and others [22, 6, 7] but not as a basis for translation. Peralta *et al.* [33] proposed an approach to translation based on the first Futamura projection [8], specialising a Horn clause interpreter for imperative programs using small-step semantics. This only handled a small imperative language with procedure calls. De Angelis *et al.* [4] further developed this approach, handling also procedure calls using a "multi-step" semantics, combining aspects of small-step and big-step semantics as discussed in Section 4.

```
% Regular expressions
% E ::= symb(Id) | E1:E2 | E1+E2 | star(E) | null | eps

pathsolve(A,Z,symb(C),_) :-
      clpClause(C,A,Cs,[Z]),
      solveConstraints(Cs).
pathsolve(A,Z,E1:E2,F) :-
      first(E2,F1),
      member(P/N,F1),
      functor(X,P,N),
      pathsolve(A,X,E1,F1),
      pathsolve(X,Z,E2,F).
pathsolve(A,Z,E1+_,F) :-
      pathsolve(A,Z,E1,F).
pathsolve(A,Z,_+E2,F) :-
      pathsolve(A,Z,E2,F).
pathsolve(A,A,star(_),_).
pathsolve(A,Z,star(E),F) :-
      first(E,FE),
      setunion(FE,F,F1),
      member(P/N,F1),
      functor(X,P,N),
      pathsolve(A,X,E,F1),
      pathsolve(X,Z,star(E),F).
```

Figure 7: Main clauses for a path expression interpreter.

Big-step and small-step semantics each have their strong and weak points from the point of view of program analysis. Small-step programs are essentially transition systems and are amenable to well established model-checking techniques. Big-step programs are more structured, allowing compositional analysis, but at the cost of having predicates with a greater number of arguments, which can be expensive for analysis algorithms.

Analysis of linear programs, with various kinds of graph path analysis, has been the subject of some previous work. Kincaid *et al.* [24] have explored the use of Tarjan's regular path expression to perform compositional program analysis starting from a linear program representation such as a control flow graph. However, they do not perform a program transformation based on the path. Wei *et al.* [39] present an algorithm for discovering the nesting structure of loops in control flow graphs, for the purpose of decompilation.

The interpreter-based approach described in this paper has both practical and conceptual advantages. Being able to transform between big-step and small-step styles, or mix them, allows use of a single verification framework but with the flexibility of different program representations. The conceptual advantages relate to the understanding gained of the connections between different styles of semantics, and how they can be derived from each other.

Acknowledgements

Discussions on semantics and Horn clauses with Alberto Pettorossi, Maurizio Proietti, Fabio Fioravanti and Emanuele De Angelis are gratefully acknowledged.

References

[1] N. Bjørner, A. Gurfinkel, K. L. McMillan & A. Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner & W. Schulte, editors: *Fields of Logic and Computation II*, *LNCS* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[2] S. Blazy & X. Leroy (2009): *Mechanized Semantics for the Clight Subset of the C Language*. *J. Autom. Reasoning* 43(3), pp. 263–288, doi:10.1007/s10817-009-9148-3.

[3] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *Program verification via iterated specialization*. *Sci. Comput. Program.* 95, pp. 149–175, doi:10.1016/j.scico.2014.05.017.

[4] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2015): *Semantics-based generation of verification conditions by program specialization*. In M. Falaschi & E. Albert, editors: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, ACM, pp. 91–102, doi:10.1145/2790449.2790529.

[5] B. Demoen (1992): *On the Transformation of a Prolog Program to a More Efficient Binary Program*. In K. Lau & T. Clement, editors: *Logic Program Synthesis and Transformation, Proceedings of LOPSTR 92, International Workshop on Logic Program Synthesis and Transformation, University of Manchester, UK, 2-3 July 1992*, Workshops in Computing, Springer, pp. 242–252, doi:10.1007/978-1-4471-3560-9_17.

[6] T. Despeyroux (1984): *Executable Specification of Static Semantics*. In G. Kahn, D. B. MacQueen & G. Plotkin, editors: *Semantics of Data Types*, *LNCS* 173, Springer-Verlag, p. 215–233.

[7] V. Donzeau-Gouge, G. Huet, G. Kahn & B. Lang (1984): *Programming Environments Based on Structured Editors: The MENTOR experience*. In D. Barstow, E. Sandewall & H. Shrobe, editors: *Interactive Programming Environments*, McGraw-Hill, p. 128–140.

[8] Y. Futamura (1971): *Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler*. *Systems, Computers, Controls* 2(5), pp. 45–50.

[9] J. P. Gallagher, M. Hermenegildo, B. Kafle, M. Klemen, P. L. García & J. Morales (2020): *Interpreters Repository*. `http://webhotel4.ruc.dk/~jpg/Software/Semantics`.

[10] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard & P. J. Stuckey (2015): *Horn clauses as an intermediate representation for program analysis and transformation*. Theory Pract. Log. Program. 15(4-5), pp. 526–542, doi:10.1017/S1471068415000204.

[11] M. Gómez-Zamalloa, E. Albert & G. Puebla (2009): *Decompilation of Java bytecode to Prolog by partial evaluation*. Inf. Softw. Technol. 51(10), pp. 1409–1427, doi:10.1016/j.infsof.2009.04.010.

[12] S. Grebenshchikov, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In J. Vitek, H. Lin & F. Tip, editors: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[13] A. Gurfinkel, T. Kahsai, A. Komuravelli & J. A. Navas (2015): *The SeaHorn Verification Framework*. In D. Kroening & C. S. Pasareanu, editors: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, Lecture Notes in Computer Science 9206, Springer, pp. 343–361, doi:10.1007/978-3-319-21690-4_20.

[14] K. S. Henriksen & J. P. Gallagher (2006): *Abstract Interpretation of PIC Programs through Logic Programming*. In: Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA, IEEE Computer Society, pp. 184–196, doi:10.1109/SCAM.2006.1.

[15] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, R. Haemmerlé, E. Mera, J. F. Morales & G. Puebla (2011): *An Overview of the Ciao System*. In N. Bassiliades, G. Governatori & A. Paschke, editors: Rule-Based Reasoning, Programming, and Applications - 5th International Symposium, RuleML 2011 - Europe, Barcelona, Spain, July 19-21, 2011. Proceedings, Lecture Notes in Computer Science 6826, Springer, p. 2, doi:10.1007/978-3-642-22546-8_2.

[16] M. V. Hermenegildo, G. Puebla, F. Bueno & P. López-García (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. Science of Computer Programming 58(1–2), doi:10.1016/j.scico.2005.02.006.

[17] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. Commun. ACM 12(10), pp. 576–580, doi:10.1145/363235.363259.

[18] H. Hojjat & P. Rümmer (2018): *The ELDARICA Horn Solver*. In N. Bjørner & A. Gurfinkel, editors: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[19] B. Kafle & J. P. Gallagher (2017): *Constraint specialisation in Horn clause verification*. Sci. Comput. Program. 137, pp. 125–140, doi:10.1016/j.scico.2017.01.002.

[20] B. Kafle, J. P. Gallagher & P. Ganty (2018): *Tree dimension in verification of constrained Horn clauses*. TPLP 18(2), pp. 224–251, doi:10.1017/S1471068418000030.

[21] B. Kafle, J. P. Gallagher & J. F. Morales (2016): *RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata*. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I, pp. 261–268, doi:10.1007/978-3-319-41528-4_14.

[22] G. Kahn (1987): *Natural Semantics*. In F. Brandenburg, G. Vidal-Naquet & M. Wirsing, editors: STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings, Lecture Notes in Computer Science 247, Springer, pp. 22–39, doi:10.1007/BFb0039592.

[23] T. Kahsai, P. Rümmer, H. Sanchez & M. Schäf (2016): *JayHorn: A Framework for Verifying Java programs*. In S. Chaudhuri & A. Farzan, editors: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I, Lecture Notes in Computer Science 9779, Springer, pp. 352–358, doi:10.1007/978-3-319-41528-4_19.

[24] Z. Kincaid, J. Breck, A. F. Boroujeni & T. W. Reps (2017): *Compositional recurrence analysis revisited*. In A. Cohen & M. T. Vechev, editors: Proceedings of the 38th ACM SIGPLAN Conference on Programming

Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, ACM, pp. 248–262, doi:10.1145/3062341.3062373.

[25] M. Leuschel & J. Jørgensen (1999): *Efficient Specialisation in Prolog Using the Hand-Written Compiler Generator LOGEN*. *Elec. Notes Theor. Comp. Sci.* 30(2), doi:10.1017/S1471068403001662.

[26] M. Leuschel, D. Elphick, M. Varea, S. Craig & M. Fontaine (2006): *The Ecce and Logen partial evaluators and their web interfaces*. In J. Hatcliff & F. Tip, editors: *PEPM*, ACM, pp. 88–94, doi:10.1145/1111542.1111557.

[27] M. Leuschel & M. H. Sørensen (1996): *Redundant Argument Filtering of Logic Programs*. In J. P. Gallagher, editor: *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, pp. 83–103, doi:10.1007/3-540-62718-9_6.

[28] J. Lloyd (1987): *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, doi:10.1007/978-3-642-83189-8.

[29] P. López-García, M. Klemen, U. Liqat & M. V. Hermenegildo (2016): *A general framework for static profiling of parametric resource usage*. *Theory Pract. Log. Program.* 16(5-6), pp. 849–865, doi:10.1017/S1471068416000442.

[30] K. Marriott, L. Naish & J. Lassez (1990): *Most Specific Logic Programs*. *Ann. Math. Artif. Intell.* 1, pp. 303–338, doi:10.1007/BF01531082.

[31] M. Méndez-Lojo, J. A. Navas & M. V. Hermenegildo (2007): *A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs*. In A. King, editor: *Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers*, *Lecture Notes in Computer Science* 4915, Springer, pp. 154–168, doi:10.1007/978-3-540-78769-3_11.

[32] H. R. Nielson & F. Nielson (1992): *Semantics with applications - a formal introduction*. Wiley professional computing, Wiley.

[33] J. Peralta, J. P. Gallagher & H. Sağlam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs*. In G. Levi, editor: *Static Analysis. 5th International Symposium, SAS'98, Pisa*, *Springer-Verlag Lecture Notes in Computer Science* 1503, pp. 246–261, doi:10.1007/3-540-49727-7_15.

[34] A. Pettorossi & M. Proietti (1999): *Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs*. *J. Log. Program.* 41(2-3), pp. 197–230, doi:10.1016/S0743-1066(99)00029-1.

[35] G. D. Plotkin (1981): *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University.

[36] G. D. Plotkin (2004): *The origins of structural operational semantics*. *J. Log. Algebr. Program.* 60-61, pp. 3–15, doi:10.1016/j.jlap.2004.03.009.

[37] G. D. Plotkin (2004): *A structural approach to operational semantics*. *J. Log. Algebr. Program.* 60-61, pp. 17–139.

[38] R. E. Tarjan (1981): *Fast Algorithms for Solving Path Problems*. *J. ACM* 28(3), pp. 594–614, doi:10.1145/322261.322273.

[39] T. Wei, J. Mao, W. Zou & Y. Chen (2007): *A New Algorithm for Identifying Loops in Decompilation*. In H. R. Nielson & G. Filé, editors: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, *Lecture Notes in Computer Science* 4634, Springer, pp. 170–183, doi:10.1007/978-3-540-74061-2_11.