Regular Path Clauses and Their Application in Solving Loops

Bishoksan Kafle[*]
IMDEA Software Institute, Spain

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

Manuel V. Hermenegildo
IMDEA Software Institute, Spain
U. Politécnica de Madrid, Spain

Maximiliano Klemen
IMDEA Software Institute, Spain

Pedro López-García
IMDEA Software Institute, Spain
Spanish Council for Sci. Research (CSIC)

José F. Morales
IMDEA Software Institute, Spain
U. Politécnica de Madrid, Spain

A well-established approach to reasoning about loops during program analysis is to capture the effect of a loop by extracting recurrences from the loop; these express relationships between the values of variables, or program properties such as cost, on successive loop iterations. Recurrence solvers are capable of computing closed forms for some recurrences, thus deriving precise relationships capturing the complete loop execution. However, many recurrences extracted from loops cannot be solved, due to their having multiple recursive cases or multiple arguments. In the literature, several techniques for approximating the solution of unsolvable recurrences have been proposed. The approach presented in this paper is to define transformations based on regular path expressions and loop counters that (i) transform multi-path loops to single-path loops, giving rise to recurrences with a single recursive case, and (ii) transform multi-argument recurrences to single-argument recurrences, thus enabling the use of recurrence solvers on the transformed recurrences. Using this approach, precise solutions can sometimes be obtained that are not obtained by approximation methods.
Keywords: Horn clauses, path programs, multi-path recurrences, multi-argument recurrences.

1 Introduction

Inferring the effect of loops is a critical program analysis task, as loops can give rise to an infinite number of program states, thus necessitating approximate solutions in general. One approach, often used in automatic resource analysis, is to extract recurrence relations from loops, and then try to solve the recurrences to get a closed form expression [34, 8, 7, 26, 2]. This approach has also been used for non-linear invariant synthesis in a series of papers [11, 23, 22] by Kincaid et al. and Humenberger et al. [19]. When the recurrences are solvable, precise solutions are obtained. By contrast, approximation methods such as abstract interpretation over some abstract domain construct solutions whose precision is limited by the expressiveness of the domain.

We present an approach to solving such loops formulated in terms of constrained Horn clauses (CHCs), which are capable of expressing the semantics of imperative programs [28, 17, 24, 13, 14, 16, 6, 21, 12]. Clauses are assumed to have numerical variables (of infinite precision); we assume that variables values have been abstracted with respect to their numerical sizes, by performing a program transformation that uses the desired metrics (such as list length, term depth, term size, etc. [26]).

Existing computer algebra systems (CASs) are powerful tools that can obtain an exact closed-form solution for many mathematical recurrences, usually deterministic single-argument recurrences with a single recursive case. However, many loops give rise to recurrences that are not of this form and are unsolvable by CASs. That is, they contain multiple recursive cases, or define functions with multiple arguments, or both.

[*]Email. `bishoksan.kafle@imdea.org`

On multiple recursive cases. Our approach differs from previous works [11, 23, 22, 2] on treating multi-path loops that give rise to multiple recursive cases, in that we transform each multi-path loop into a program containing nested loops, each of which has a single path. The transformation preserves all the different paths and uses regular path expressions. Previous approaches generate one recursive equation for each multi-path loop by merging different loop paths, which may result in a significant loss of precision. Various forms of *control-flow refinement* [15, 31] can transform a restricted form of multi-path loops, called multi-phase loops, into a semantically equivalent sequence of single path loops. Polyvariant specialisation [10, 25, 30] can achieve the same goal. These specialisation techniques are in any case orthogonal and complementary to the ones that we propose.

As regards solving, Kincaid et al. [11, 23, 22] can obtain a closed-form solution of a system of inequations, whereas, like [2], we rely on existing CASs to obtain a closed-form solution of a system of recurrences. Humenberger et al. [19] can generate polynomial invariants from multi-path numeric loops with polynomial assignments (also known as extended p-solvable loops). They take each single path loop separately, derive a system of recurrences from it and solve them to a closed-form. Then, from the closed-form, they derive the polynomial invariant ideal and combine the ideals of each loop-path iteratively until a fixed-point is reached. In contrast to this, we do not need a fixed point computation that exploits the special properties of polynomial ideals, but rather aim to transform multi-path loops into a form to which existing solvers can be applied.

On multiple-argument functions. Our approach in this area is motivated by and adapted from the work of Farzan and Kincaid [11]. In their work, separate recurrences are derived by constructing, for each argument, a function of one argument, namely a loop counter k, which defines the value of the argument on the k^{th} loop iteration. We go beyond [11] and present a systematic approach for constructing these functions, based on regular path programs, and show that the separate recurrences can be solved to yield a solution for the original multi-argument recurrence. In [2] the recurrence is re-expressed as a function of a single loop counter variable, but separate functions are not derived for each argument. Farzan and Kincaid [11] use a quantifier elimination procedure available only to a handful of theories to eliminate the counter, whereas we follow the approach of Albert et al. [2] in using a ranking function to estimate its value.

In both of the aspects above, the concept of a path program plays an essential role in our approach. The main contributions of the paper are as follows:

- A transformation that reformulates multi-path loops (that give rise to recurrences with multiple recursive cases) as single-path loops (that give rise to recurrences with single recursive case) (§3).
- A transformation that derives single argument recurrences from multi-argument ones (§4).

The paper is organized as follows. After covering introductory materials in §2, we introduce regular path programs in §3 and path programs with counters in §4. Finally, we present concluding remarks in §6.

2 Preliminaries

We assume that a program P is represented as a set of constrained Horn clauses (CHCs) of the form $\mathsf{c} : p(\mathbf{x}) \leftarrow \phi \wedge p_1(\mathbf{x_i}) \wedge \ldots \wedge p_n(\mathbf{x_n}), n \geq 0$, where c is a unique identifier for the clause in P, p and p_i are predicates, $\mathbf{x}$ and $\mathbf{x_i}$ are sequences of distinct variables (the symbol $\mathbf{x}$ represents a sequence of variables $x_1, \ldots, x_n$), and the formula ϕ is a conjunction of constraints over some constraint theory $\mathbb{T}$. Sometimes we treat a conjunction of constraints as a set of constraints. $p(\mathbf{x})$ is called the *head* and

$\phi \wedge p_1(\mathbf{x_i}) \wedge \ldots \wedge p_n(\mathbf{x_n})$ is called the *body* of the clause. If $n = 0$, then the clause is called a *constrained fact*, if $n \leq 1$ then it is called a *linear* clause and *non-linear* if it is not linear. For convenience we sometimes write an empty conjunction as true, thus a constrained fact $p(\mathbf{x}) \leftarrow \phi$ is sometimes written as as $p(\mathbf{x}) \leftarrow \phi, \mathsf{true}$; and the head of the clause can be the predicate false. A program is called linear if it contains only linear clauses. Normally, we write a clause as $p(\mathbf{x}) \leftarrow \phi, p_1(\mathbf{x_i}), \ldots, p_n(\mathbf{x_n})$ using comma instead of $\wedge$. The expression *vars(t)* represents the set of variables of a term t.

A finite *leftmost-derivation* (or simply *derivation*) in a program P is a sequence $\leftarrow G_0, \ldots, \leftarrow G_n$ where for $0 \leq i \leq n-1$ $G_i = \varphi_i, q_1(\mathbf{y_1}), \ldots, q_k(\mathbf{y_k})$, P contains a (suitably renamed) clause $q_1(\mathbf{y_1}) \leftarrow \phi, r_1(\mathbf{z_1}), \ldots, r_m(\mathbf{z_m})$, $\phi_{i+1} = \varphi_i \wedge \phi$ and $G_{i+1} = \varphi_{i+1}, r_1(\mathbf{z_1}), \ldots, r_m(\mathbf{z_m}), q_2(\mathbf{y_2}), \ldots, q_k(\mathbf{y_k})$. The derivation is *feasible* if φ_n is satisfiable. A derivation is *successful* if it is feasible and $G_n = \varphi_n, \mathsf{true}$.

$\mathtt{c_1}$. $\mathtt{wh(a,b)} \leftarrow \mathtt{a} > 0, \mathtt{b} > 0, \mathtt{wh(a, b-1)}$
$\mathtt{c_2}$. $\mathtt{wh(a,b)} \leftarrow \mathtt{a} > 0, \mathtt{b} \leq 0, \mathtt{wh(a-1, b+a)}$
$\mathtt{c_3}$. $\mathtt{wh(a,b)} \leftarrow \mathtt{a} \leq 0$

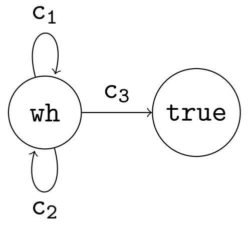

Figure 1: (left) Multi-path loop and (right) its CFG.

Definition 1 (Control flow graph, path, loop) *A control flow graph (CFG) of a linear program P is a labelled directed graph $\langle N, E \rangle$ where*

- *N is a set of nodes consisting of predicates of P (including* true *and* false*);*
- *E is a set of labelled edges. There is an edge from p to p_1 labelled by* c *if there is a clause $p(\mathbf{x}) \leftarrow \phi \wedge p_1(\mathbf{x_1}) \in P$ having identifier* c.

A path *of length n ($n \geq 0$) in a CFG is a possibly empty sequence of n connected edges, and is written as a sequence of* edge labels. *A* loop *is a non-empty path which starts and ends with the same node n and does not visit n in between. A loop is* directly recursive *if it is of length 1.*

Figure 1 shows an example of a linear program and its CFG.

Let $c_1 \ldots c_n$ ($n \geq 0$) be a (possibly empty) path in the CFG for P. Then the derivation $\leftarrow G_0, \ldots, \leftarrow G_n$ *corresponds* to $c_1 \ldots c_n$ if (i) $G_0 = \varphi_0, p_0(\mathbf{x_0})$, where φ_0 is empty and (if $n > 0$) the edge c_1 starts at p_0; (ii) for $1 \leq i \leq n$, $G_{i-1} = \varphi_{i-1}, p_i(\mathbf{x_i})$ and c_i is the identifier of a (renamed) clause $p_{i-1}(\mathbf{x_{i-1}}) \leftarrow \phi_i, p_i(\mathbf{x_i})$ in P, $\varphi_i = \varphi_{i-1} \wedge \phi_i$ and $G_i = \varphi_i, p_i(\mathbf{x_i})$. If $n = 0$ then p_0 is an arbitrary predicate from P.

We say that a path is feasible or successful if and only if the derivation corresponding to the path is feasible or successful respectively.

Definition 2 (Recurrence equation) *A recurrence equation is an equation of the form $\mathsf{f}(\mathbf{x}) = e(\mathbf{x}) + \sum_{i=1}^{n} a_i * \mathsf{f_i}(\mathbf{x_i}), \phi$ where $e(\mathbf{x})$ is an arbitrary arithmetic expression, a_i some constant,* f *and* $\mathsf{f_i}$ *are function symbols, where* $\mathsf{f_i}$ *is not necessarily different from* f, $\mathbf{x}, \mathbf{x_i}$ *are sequence of variables and ϕ is a formula over $\mathbb{Z}$. If $n \leq 1$, then it is called a linear (recurrence) equation.*

A set of such equations is called a (recurrence) equation system. Each equation in a system is given an id which uniquely identifies it. A linear equation system is the one in which all equations are linear.

Definition 3 (Equation graph) *An equation graph (EG) of a linear equation system S is a labelled directed graph $G_S = \langle N, E, entry, exit \rangle$ where*

c	$path_c(p(\mathbf{x}),q(\mathbf{x}')) \leftarrow \phi.$, where clause $p(\mathbf{x}) \leftarrow \phi, q(\mathbf{x}') \in P$ has identifier c
ε	$path_\varepsilon(p(\mathbf{x}),p(\mathbf{x})) \leftarrow \mathsf{true}.$
$\emptyset$	no clause
e_1e_2	$path_{e_1e_2}(p(\mathbf{x}),z) \leftarrow path_{e_1}(p(\mathbf{x}),q(\mathbf{x}')), path_{e_2}(q(\mathbf{x}'),z).$, for each $q \in \mathit{firstpred}(e_2)$
e_1+e_2	$path_{e_1+e_2}(p(\mathbf{x}),z) \leftarrow path_{e_1}(p(\mathbf{x}),z).$ $path_{e_1+e_2}(p(\mathbf{x}),z) \leftarrow path_{e_2}(p(\mathbf{x}),z).$
e^*	$path_{e^*}(p(\mathbf{x}),p(\mathbf{x})) \leftarrow \mathsf{true}.$ $path_{e^*}(p(\mathbf{x}),p(\mathbf{x}'')) \leftarrow path_{e^*}(p(\mathbf{x}),p(\mathbf{x}')), path_e(p(\mathbf{x}'),p(\mathbf{x}'')).$

Figure 2: Formation of path clauses corresponding to a regular expression e, starting from node p.

- *N is a set of nodes consisting of functions of S,*
- *E is a set of edges. There is an edge from f to f_1 labelled by* eq *if there is an equation* $(\mathsf{f}(\mathbf{x}) = e(\mathbf{x}) + a_1 * \mathsf{f}_1(\mathbf{x_1}), \phi) \in S$ *having identifier* eq.
- *entry and exit respectively represent the entry and the exit node of G_S.*

3 Regular path clauses

In this section we introduce *path* predicates and clauses; given a set of linear CHCs P, the meaning of the predicate $path_e(p(\mathbf{x}),q(\mathbf{x}'))$, where e is a regular expression describing paths in the CFG for P, is that (i) there is a feasible path in the CFG for P starting from the node p and ending at the node q, such that the path is contained in the set described by e and (ii) $p(\mathbf{x})$ and $q(\mathbf{x}')$ are the atoms at the start and end of the derivation corresponding to the path. (We abuse notation and overload p and q as both function and predicate symbols). A well known algorithm by Tarjan [33] computes a regular path expression describing all paths in the CFG from a designated entry node to an exit node. We previously developed an interpreter that is guided by a regular path expression [12], and partially evaluated it with respect to a set of linear CHCs P. Here, we outline the idea behind the interpreter, which is that given a regular expression e, each subexpression of e defines a set of subpaths, which are composed to give the overall paths. These path predicates are defined by non-linear CHCs even though the CFG is defined from a set of linear CHCs.

Regular expressions e over some alphabet Σ are expressions of the form $e ::= c \mid \varepsilon \mid \emptyset \mid e_1e_2 \mid e_1 + e_2 \mid e^*$, where $c \in \Sigma$. Σ in this case is the set of clause identifiers in P. Given a set of linear CHCs P, construct its CFG. Compute a regular path expression from the entry node to the exit node of the CFG. Recall that for any regular expression e, we can compute the set $\mathit{first}(e)$, which is the set of elements of Σ that can start a path described by e. An element of $\mathit{first}(e)$ is thus an edge in the CFG, corresponding to a clause in P. The function $\mathit{firstpred}(e)$ returns the set of predicates in the head of some element of $\mathit{first}(e)$. Figure 2 shows how path clauses are constructed based on the structure of the regular expressions. Note that we choose to define the clauses for an expression e^* using left recursion, that is, corresponding to the expansion $e^* = \varepsilon + e^*e$, rather than the equivalent $e^* = \varepsilon + ee^*$, which might be expected. This enables the generation of suitable recurrence equations, which will be discussed in Section 4.

Theorem 1 *Let P be a set of linear CHCs, G be its CFG and let p and q be (possibly identical) predicates in P (including* false *and* true*). Let e be a regular expression describing all paths from p to q in the CFG. Then there is a feasible derivation $\leftarrow p(\mathbf{x}), \ldots, \leftarrow \varphi_n, q(\mathbf{x}')$ in P if and only if there is a successful derivation $\leftarrow path_e(p(\mathbf{x}),q(\mathbf{x}')), \ldots, \leftarrow \varphi_n,$* true *in the path clauses for P and e.*

In other words, the path clauses capture all feasible derivations of the original set of clauses P. It should be emphasised that the path clauses capture derivations in P, as stated by Theorem 1, but the path clauses themselves are not intended as an executable program. Clearly the left-recursive form is not amenable to execution with the standard computational strategy, and the epsilon steps after loops are likely to introduce non-determinism. So is not the intention to analyse the computational cost of the path program or its termination behaviour. The only property of the path clauses that we exploit in Section 4 is that *path* predicates capture the relationship between the values of variables at the start and end of derivations in P. The notion of path clauses is related to the big-step semantics [12] and path predicaes are sometimes called summary predicates.

3.1 Eliminating multi-path loops

Theorem 1 holds for any expression e that describes all paths from p to q in the CFG, thus we can freely transform e into an equivalent expression e' and perform the generation of the path clauses with respect to e'. A regular expression of the form $(e_1 + \cdots + e_m)^*$ is called a *multi-path-loop expression*, where $e_1, \ldots, e_m$ are the paths through the loop. Any regular expression can be transformed to an equivalent regular expression not containing any multi-path-loop expression, by repeatedly replacing any subexpression of the form $(e_1 + e_2)^*$ by an equivalent regular expression $(e_1^* e_2^*)^*$, $e_1^*(e_2 e_1^*)^*$, or $e_2^*(e_1 e_2^*)^*$, among others. We discuss the choice of replacement expression in Section 6.

An algorithm for removing multi-path loops using regular expression transformation is given in Alg. 1. Given a set of linear clauses with its CFG and distinguished entry and exit nodes, it first computes regular expression describing all paths from entry to the exit node using using Tarjan's algorithm [33]. The expression is then rewritten without the choice (+) operator which is then used to transform the original program based on Figure 2. The resulting program does not contain multi-path loops.

Algorithm 1 Algorithm for removing multi-path loops

```
1: **Input**: CFG $G$ of linear clauses $P$, entry node $p$ and exit node $q$
2: **Output**: Program $P'$ without multi-path loops
3: $e \leftarrow reg\_path\_expr(G, p, q)$                ▷ Compute regular path expression [33]
4: $e \leftarrow rewrite\_wo\_choice(e)$                  ▷ Rewrite $e$ without choice operator within star
5: $P' \leftarrow construct\_path\_cls(P, e, p)$          ▷ Figure 2
6: **return** $P'$
```

Theorem 2 *The output of Algorithm 1 is a program that contains no multi-path loop.*

```
path(wh(a,b),true) ← wh2(a,b,a',b'),wh5(a',b',a'',b''), a'' ≤ 0.
wh2(a,b,a,b) ← true.
wh2(a,b,a',b'−1) ← wh2(a,b,a',b'), a' > 0, b' > 0.
wh5(a,b,a,b) ← true.
wh5(a,b,a'',b'') ← wh5(a,b,a',b'), a' > 0, b' ≤ 0,wh2(a'−1,b'+a',a'',b'').
```

Figure 3: The path clauses for Fig. 1 (left) wrt $c_1^*(c_2 c_1^*)^* c_3$ (equivalent to $(c_1 + c_2)^* c_3$).

Example 1 *Let P be the set of clauses in Figure 1 (left). A regular path expression for paths in its CFG from wh to* true *is* $(c_1 + c_2)^* c_3$. *An equivalent path expression with no multi-path-loop expressions is* $c_1^*(c_2 c_1^*)^* c_3$. *Figure 3 shows the path clauses for P derived using the latter expression, after some path predicates have been unfolded and the following renamings applied:* $wh_2(a,b,a',b')$ *is a renaming of* $path_{c_1^*}(wh(a,b), wh(a',b'))$ *and* $wh_5(a,b,a',b')$ *is a renaming of* $path_{(c_2 c_1^*)^*}(wh(a,b), wh(a',b'))$.

Thus we can replace the original problem of analysing a multi-path loop by the problem of analysing a program with nested single-path loops. In the example, the predicate wh_2 is nested within wh_5. Furthermore, each loop predicate has multiple input and output values. In the next section, we investigate how to analyse the relationship between the input and output values of each loop.

4 Regular path clauses with counters

In Section 3 it was shown that multi-path loops could be eliminated in path clauses, by transforming the regular path expression for the loop. Thus, all loops after the transformation are represented by a single directly recursive path clause of the form: $\mathtt{path}_{\mathtt{e}^*}(\mathbf{x},\mathbf{x_2}) \leftarrow \mathtt{path}_{\mathtt{e}^*}(\mathbf{x},\mathbf{x_1}),\mathtt{path}_{\mathtt{e}}(\mathbf{x_1},\mathbf{x_2})$. Here $\mathtt{path}_{\mathtt{e}}(\mathbf{x_1},\mathbf{x_2})$ is the path for the loop body, which may itself contain loops, but these do not depend on the recursive predicate $\mathtt{path}_{\mathtt{e}^*}$ and can be solved separately; let us say that the solution is an expression $\phi(\mathbf{x_1},\mathbf{x_2})$. Then the loop to be solved is a single directly recursive clause $\mathtt{path}_{\mathtt{e}^*}(\mathbf{x},\mathbf{x_2}) \leftarrow \phi(\mathbf{x_1},\mathbf{x_2}),\mathtt{path}_{\mathtt{e}^*}(\mathbf{x},\mathbf{x_1})$. Due to the left-recursive form of the path, the input arguments $\mathbf{x}$ remain constant from one recursive call to the next; this is an important property when forming recurrence equations, as will be seen.

We now introduce a counter $\mathtt{k}$ to such clauses, representing the length of the loop path. This allows us to capture the effect of the loop after $\mathtt{k}$ iterations. After adding the counter, we obtain the following clauses for each single-path loop. $\mathtt{k}$ is considered to be an input and is decremented until it reaches 0.

$$\mathtt{path}_{\mathtt{e}^*}(\mathtt{k},\mathbf{x},\mathbf{x_2}) \leftarrow \mathtt{k} > 0, \phi(\mathbf{x_1},\mathbf{x_2}), \mathtt{path}_{\mathtt{e}^*}(\mathtt{k}-1,\mathbf{x},\mathbf{x_1}).$$
$$\mathtt{path}_{\mathtt{e}^*}(\mathtt{k},\mathbf{x},\mathbf{x}) \leftarrow \mathtt{k} = 0.$$

We also assume that there is a *ranking function* [32, 29, 3] for the loop, guaranteeing termination; that is, there is an upper bound on the number of iterations k. A ranking function for the loop is a function on the input values such that $r(\mathbf{x}) \in \mathbb{N}$ and $\phi(\mathbf{x_1},\mathbf{x_2}) \rightarrow r(\mathbf{x_1}) > r(\mathbf{x_2})$. Thus the maximum value of $\mathtt{k}$ in successful derivations starting from $\mathtt{path}_{\mathtt{e}^*}(\mathtt{k},\mathbf{x},\mathbf{x_2})$ is $r(\mathbf{x})$.

Example 2 *Consider a loop clause* $\mathtt{wh}(\mathtt{x}_1,\mathtt{y}_1) \leftarrow \mathtt{x}_1 > 0, \mathtt{y}_1 > 0, \mathtt{x}_2 = \mathtt{x}_1 - 1, \mathtt{y}_2 = \mathtt{y}_1 + \mathtt{x}_1, \mathtt{wh}(\mathtt{x}_2,\mathtt{y}_2)$. *Starting from the input,* (x_1,y_1)*, we aim to find the values of these variables when the loop terminates, in terms of the original ones when entering the loop. For this purpose, let us first define a path of length k going from* $(\mathtt{x},\mathtt{y})$ *to* $(\mathtt{x}_2,\mathtt{y}_2)$ *as a relation* $\mathtt{path}(\mathtt{k},\mathtt{x},\mathtt{y},\mathtt{x}_2,\mathtt{y}_2)$*. The clauses for* `path` *are as follows.*

$$\begin{array}{lcl} \mathtt{path}(\mathtt{k},\mathtt{x},\mathtt{y},\mathtt{x}_2,\mathtt{y}_2) & \leftarrow & \mathtt{k} > 0, \mathtt{k}_1 = \mathtt{k}-1, \mathtt{x}_2 = \mathtt{x}_1 - 1, \mathtt{y}_2 = \mathtt{y}_1 + \mathtt{x}_1, \\ & & \mathtt{path}(\mathtt{k}_1,\mathtt{x},\mathtt{y},\mathtt{x}_1,\mathtt{y}_1), \mathtt{x}_1 > 0, \mathtt{y}_1 > 0. \\ \mathtt{path}(\mathtt{k},\mathtt{x},\mathtt{y},\mathtt{x},\mathtt{y}) & \leftarrow & \mathtt{k} = 0. \end{array}$$

The recursive clause can be read as saying: if there is a path of length $k-1$ *from* (x,y) *to* (x_1,y_1)*, and* $x_1 > 0, y_1 > 0$*, then the path can be extended on the right to a path of length k from* (x,y) *to* (x_2,y_2)*, where* (x_1,y_1,x_2,y_2) *satisfy the constraints from the loop clause above. In other words,* $\mathtt{path}(\mathtt{k},\mathtt{x},\mathtt{y},\mathtt{x}_2,\mathtt{y}_2)$ *means that given* x,y,k *as inputs,* x_2 *and* y_2 *are the values of x and y after k iterations of the* `wh` *loop.*

4.1 Deriving recurrences from path clauses with counters

Given such a loop clause with a counter, we formulate recurrence equations for each of the variables at the end of the path (called the *output* variables of the loop). This can be done systematically using the method described in [9, 26]; here we give an informal account of the construction. Given $\mathtt{path}_{\mathtt{e}^*}(\mathtt{k},\mathbf{x},\mathbf{y})$ representing a loop predicate p, where $\mathbf{x},\mathbf{y}$ are m-tuples of variables respectively representing the values of variables at the start and end respectively of the path, each of the m variables in $\mathbf{y}$ is a function of $\mathtt{k},\mathbf{x}$.

This is due to the fact that given $\mathtt{k}$ and $\mathbf{x}$, the values of the output $\mathbf{y}$ are completely determined. Hence, we can define m functions $\mathtt{p}^1(\mathtt{k},\mathbf{x}),\ldots,\mathtt{p}^{\mathtt{m}}(\mathtt{k},\mathbf{x})$, giving the value of the respective elements of the m-tuple $\mathbf{y}$. That is $\mathtt{path}_{\mathtt{e}^*}(\mathtt{k},\mathbf{x},\mathbf{y})$ is equivalent to $\mathtt{p}^1(\mathtt{k},\mathbf{x}) = \mathtt{y}_1 \wedge \ldots \wedge \mathtt{p}^{\mathtt{m}}(\mathtt{k},\mathbf{x}) = \mathtt{y}_{\mathtt{m}}$. The recurrence equations are then obtained by substituting this functional expression for the $\mathtt{path}_{\mathtt{e}^*}$ atoms in the loop clauses.

We say that function f *depends on* function g in a system of recurrences S if g is needed in order to evaluate f. There is a *cyclic* dependency between f and g if they depend on each other. If S contains no cyclic dependencies, then we can send the recurrences to a CAS to be solved in topological order.

Example 3 (Continued from Example 2) *Define two functions* $\mathtt{wh}^{\mathtt{x}}(k,x,y)$ *and* $\mathtt{wh}^{\mathtt{y}}(k,x,y)$ *defining the values of* $\mathtt{x}$ *and* $\mathtt{y}$ *after* k *iterations of the* $\mathtt{wh}$ *loop. The functions are defined as follows.*

$$\mathtt{wh}^{\mathtt{x}}(k,x,y) = \begin{cases} \mathtt{wh}^{\mathtt{x}}(k-1,x,y)-1, & for \quad k>0, \\ x, \quad for \quad k=0 \end{cases} \tag{1}$$

$$\mathtt{wh}^{\mathtt{y}}(k,x,y) = \begin{cases} \mathtt{wh}^{\mathtt{y}}(k-1,x,y)+\mathtt{wh}^{\mathtt{x}}(k-1,x,y), & for \quad k>0, \\ y, \quad for \quad k=0 \end{cases} \tag{2}$$

Since there are no cyclic dependencies between $\mathtt{wh}^{\mathtt{x}}$ *and* $\mathtt{wh}^{\mathtt{y}}$*, they can be solved in reverse topological order of the strongly connected components (SCCs); that is equations for* $\mathtt{wh}^{\mathtt{x}}$ *are solved first and then equations for* $\mathtt{wh}^{\mathtt{y}}$*, reusing solutions from the lower to the higher SCC when necessary.*

4.2 Removal of symbolic constant arguments

Since Equation 1 has multiple arguments, it is not a *mathematical recurrence* that can directly be solved by the CASs. Note, however, that the arguments x,y remain constant and unconstrained throughout all recursions of the equation. In other words, x and y play no role in the recursive case of Equation 1; x appears in the base-case and will be present in the solution of the equation. But they can be removed as arguments of $\mathtt{wh}^{\mathtt{x}}$ and the occurrence of x in the base case can be replaced by a constant function returning x. The resulting recurrence equation will have the same solution as the original. We call this type of arguments *symbolic constants*, and propose a method for inferring which arguments can be classified as such in Section 4.3.

Example 4 (Continued from Example 3) *Making these transformations, we obtain the simplified Equation 3, where* c_x *is a constant function representing the argument* x*. We abuse notation and overload* $\mathtt{wh}^{\mathtt{v}}$ *with different arities throughout the paper.*

$$\mathtt{wh}^{\mathtt{x}}(k) = \begin{cases} \mathtt{wh}^{\mathtt{x}}(k-1)-1, & for \quad k>0, \\ c_x, \quad for \quad k=0 \end{cases} \tag{3}$$

Observe that this equation does satisfy the syntactic form required by the CASs and can be solved to obtain $\mathtt{wh}^{\mathtt{x}}(k) = c_x - k$ *as a solution. This is also the solution of Equation 1, as we shall see later (Lemma 1). Thus, we have* $\mathtt{wh}^{\mathtt{x}}(k,x,y) = x-k$*. Replacing the solution of* $\mathtt{wh}^{\mathtt{x}}(k,x,y)$ *in Equation 2 and simplifying, we obtain Equation 4:*

$$\mathtt{wh}^{\mathtt{y}}(k,x,y) = \begin{cases} \mathtt{wh}^{\mathtt{y}}(k-1,x,y)+x-k+1, & for \quad k>0, \\ y, \quad for \quad k=0 \end{cases} \tag{4}$$

Now, following similar reasoning, x *and* y *are classified as symbolic constants. This allows further simplification, generating Equation 5:*

$$\mathtt{wh}^{\mathtt{y}}(k) = \begin{cases} \mathtt{wh}^{\mathtt{y}}(k-1)+c_x-(k-1), & for \quad k>0, \\ c_y, \quad for \quad k=0 \end{cases} \tag{5}$$

which can be solved to yield $\mathtt{wh}^y(k) = c_y - \frac{1}{2}k(k-2x-1)$ *and hence* $\mathtt{wh}^y(k,x,y) = y - \frac{1}{2}k^2 + kx + \frac{1}{2}k$.

This approach has advantages and disadvantages: on the one hand (i) identifying symbolic constants and simplifying equations can be done locally, for instance, at SCC level (where global analysis is not required) and (ii) when solving equations corresponding to a higher SCC, all variables except the counter k in lower SCCs become *symbolic constants* resulting in equations with an unary argument k.

On the other hand, the solution shifts the problem to estimating the value of the path counter variable (k) in terms of the original arguments. Observe that k decreases precisely by 1 in each recursive call to the function and always stays non-negative. Since k cannot always be computed precisely, it needs to be approximated. As mentioned earlier, we assume that the loop has a ranking function that decreases at least by 1 in each recursive call. Let r be the value of the ranking function in the initial state. Then, $k \in [0,r]$ is a safe bound for k.

Example 5 (Continued from Example 4) *Going back to the original loop clause in Example 2, a ranking function for* $\mathtt{wh}(x,y)$ *is x (or more precisely max*$(0,x)$*, but we assume that x may not have negative values). Let us say that* $\mathtt{wh}(x_1,y_1)$ *is the call at the start of the loop. Then the loop executes k times where k is in the interval* $[0,x_1]$*). Substituting this interval for k in the solutions of Equations 1 and 2 and using interval arithmetic, we can infer that, when the loop terminates, the values of* x_1 *and* y_1 *lie in intervals* $[0,x_1]$ *and* $[y_1 - 1/2x_1^2, y_1 + 1/2x_1 + x_1^2]$ *respectively.*

The use of interval arithmetic gives sound results but can lose precision, in particular since the value substituted for k should be the same for both output variables x and y. This is a well known source of imprecision in interval arithmetic; for instance, the interval approximation of $x*x$ where x is in the interval $[-1,1]$ is $[-1,1]$ since it includes the result $-1*1$, whereas taking into account that the same value of x is used for each occurrence would give the positive interval $[0,1]$ since the sign of both multiplicands is the same. However, for the inference of upper bounds, interval arithmetic is expected to give reasonable results. Further experimentation will be needed.

Summary of the procedure. Based on the discussion above, we outline below the steps for inferring relations between variables before and after the loop (also known as loop summaries). We assume that the loop is defined by a single recursive linear path clause; that is, multi-path loops have been eliminated by the technique presented in Section 3, and the loop body (defined by predicates in lower SCCs of the path clauses) has been solved.

1. Given a loop clause expressed as a single recursive path clause, add a counter k to the clause, with a base case where $k = 0$.
2. Then, given an input tuple $\mathbf{x}$ and k, set up equations for output tuple $\mathbf{z}$ using the path clause, using methods such as as in [9].
3. Transform multi-argument equations to single-argument equations by detecting and removing symbolic constants, as discussed in Section 4.3.
4. Solve the resulting equations using CASs and replace the path counter in the solution with the value of the ranking function in the initial state of the original loop clause.

4.3 Inference of symbolic constants

In the procedure above, multi-argument functions were transformed to single-argument functions by removing arguments that were symbolic constants. Symbolic constants can be detected using an adapted data flow analysis technique based on reaching definitions [27], RD for short. RD provides information about where variables have most recently obtained their values. We adapt it for a system of linear recurrence equations.

The algorithm assigns to each node of an EG a set of pairs of the form (v,e) to express the fact that the variable v may have been last defined (or constrained) in the equation e. We refer the readers to [27] for details on the analysis. Next, we explain how the central concept of a "definition" of a variable is adapted for linear equation systems.

Definition 4 (Defined and constrained variable) *Given an EG $\langle V,E,entry,exit\rangle$ of a linear equation system, a variable v is* defined *in the edge e, where e is the equation* $(\mathsf{p}(\mathbf{x}) = expr(\mathbf{x}) + a_1 * \mathsf{q}(\mathbf{x}'), \phi)$ *if* $\phi \models_{\mathbb{T}} v \neq v'$, *where* $v' \in \mathbf{x}'$ *and* $v \in \mathbf{x}$ *are the corresponding occurrences of an argument in the right and left sides of e. We denote it by* $(v,e)^d$. *Similarly, v is* constrained *in that edge if* $v \in \mathtt{vars}(\mathsf{project}_{\mathbf{x}}(\phi))$. *We denote it by* $(v,e)^c$.

Based on this definition we now define symbolic constants.

Definition 5 (Symbolic constant) *Given an EG of a linear system of equations S, a variable of S is called a symbolic constant if it is neither defined nor constrained in any edges of the EG.*

That is, when adapting RD analysis to infer symbolic constants for a linear system of equations, we consider a weaker notion of "defined variable" and say that a variable is defined (denoted as (v,e)) if $(v,e)^d \vee (v,e)^c$. With this definition, the standard analysis can be used as follows. Let G_S be an EG of some equation system S and let $\mathtt{rd}(n)$ denote the results of the analysis for node n.

- Compute RD assignment for G_S, starting with $\mathtt{rd}(n) = \emptyset$ for all nodes n of G_S. Let $\mathtt{rd}(exit)$ be the result for the *exit* node.
- Let V be a set of variables of interest. $v \in V$ is a symbolic constant if $(v,\_) \notin \mathtt{rd}(exit)$. In other words, no definition of v reaches the exit node. Since it was not defined or constrained on entry, we can conclude that it was never defined or constrained in S.

Example 6 (RD analysis and symbolic constants) *Let $G_S = \langle N,E,entry,halt\rangle$ below be the EG of Equation 1 with its cases labelled as edges e_1 and e_2, where $N = \{entry, \mathtt{wh}^{\mathtt{x}}, halt\}$ and $E = \{e_0,e_1,e_2\}$, where e_0 is the entry edge from entry to $\mathtt{wh}^{\mathtt{x}}$. The result of the analysis of G_S is as follows.*

$$\mathtt{rd}(entry) = \emptyset,\ \ \mathtt{rd}(\mathtt{wh}^{\mathtt{x}}) = \{(k,e_1)\},\ \ \mathtt{rd}(halt) = \{(k,e_2)\}$$

It is easy to see that (i) $(k,e_1)^d \wedge (k,e_1)^c$: k is both defined and constrained in the recursive equation e_1; (ii) $(k,e_2)^c$: k is only constrained in the base case e_2; and (iii) variables are neither defined nor constrained in e_0. Since $(x,\_) \notin \mathtt{rd}(exit)$ and $(y,\_) \notin \mathtt{rd}(exit)$, both x and y are symbolic constants.

Lemma 1 (Soundness of removing symbolic constants from equations) *Let S be a system of equations and V be a set of symbolic constants such that $U \subseteq V$ appear as arguments of a function f in S. Let S' be an another system obtained by removing the variables in U from arguments of f and replacing any other occurrences of variables in U by constant functions that return the initial value they are assigned when evaluating a call to f. Then S and S' have the same solution.*

4.4 Solving multi-argument recursive equations

In previous sections, we showed how to solve loops represented as clauses using recurrences. We can apply the same technique to solve some equations with multi-argument functions by first transforming them to tail-recursive clauses and solving them. It is not always possible to transform all the recursive calls to tail recursive calls but we can rewrite some programs, for example, using accumulator(s), continuation-passing style (introducing a stack) or the standard techniques based on unfold-fold transformations [4] to achieve the required form. In this paper, we make use of the latter as it fits well with our approach.

Consider the simple case of recurrence equations of Definition 2, namely $\mathtt{f}(\mathbf{x}) = e(\mathbf{x}) + \mathtt{f}(\mathbf{x_i}), \phi$ or $\mathtt{f}(\mathbf{x}) = e(\mathbf{x}), \phi$. Functions of this form can be transformed by unfold-fold to a tail-recursive *accumulating* form, which can then be written as CHCs of the required form. We illustrate this with an example below.

$$\mathtt{wh}(\mathtt{x_1},\mathtt{y_1}) = \begin{cases} \mathtt{x_1}+\mathtt{y_1}+\mathtt{1}+\mathtt{wh}(\mathtt{x_1}-\mathtt{1},\mathtt{y_1}+\mathtt{1}) \text{ for } \mathtt{x_1} > \mathtt{0} \\ \mathtt{0} \text{ for } \mathtt{x_1} \le \mathtt{0} \end{cases} \tag{6}$$

Following a fold-unfold transformation, we can obtain the following equivalent system of equations.

$$\mathtt{wh}(\mathtt{x_1},\mathtt{y_1}) = \begin{cases} \mathtt{wh\_aux}(\mathtt{x_1}-\mathtt{1},\mathtt{y_1}+\mathtt{1},\mathtt{x_1}+\mathtt{y_1}+\mathtt{1}) \text{ for } \mathtt{x_1} > \mathtt{0} \\ \mathtt{0} \text{ for } \mathtt{x_1} \le \mathtt{0} \end{cases} \tag{7}$$

$$\mathtt{wh\_aux}(\mathtt{x_1},\mathtt{y_1},\mathtt{z_1}) = \begin{cases} \mathtt{wh\_aux}(\mathtt{x_1}-\mathtt{1},\mathtt{y_1}+\mathtt{1},\mathtt{x1}+\mathtt{y1}+\mathtt{1}+\mathtt{z_1}) \text{ for } \mathtt{x_1} > \mathtt{0} \\ \mathtt{z_1} \text{ for } \mathtt{x_1} \le \mathtt{0} \end{cases} \tag{8}$$

The recursive function wh_aux can now be solved by transforming its first equation into the CHC.

$$\begin{array}{lll} \mathtt{wh\_aux}(\mathtt{x_1},\mathtt{y_1},\mathtt{z_1},\mathtt{w_1}) \leftarrow & \mathtt{x_1} > \mathtt{0}, \mathtt{x_2} = \mathtt{x_1}-\mathtt{1}, \mathtt{y_2} = \mathtt{y_1}+\mathtt{1}, \mathtt{z_2} = \mathtt{x_1}+\mathtt{y_1}+\mathtt{1}+\mathtt{z_1}, \mathtt{w_2} = \mathtt{w_1} \\ & \mathtt{wh\_aux}(\mathtt{x_2},\mathtt{y_2},\mathtt{z_2},\mathtt{w_2}). \end{array}$$

This loop clause has ranking function x_1. The corresponding path clauses with counter k is given as:

$$\begin{array}{ll} \mathtt{path}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{w},\mathtt{k},\mathtt{x_2},\mathtt{y_2},\mathtt{z_2},\mathtt{w_2}) \leftarrow & \mathtt{x_1} > \mathtt{0}, \mathtt{k} > \mathtt{0}, \mathtt{k_1} = \mathtt{k}-\mathtt{1}, \\ & \mathtt{x_2} = \mathtt{x_1}-\mathtt{1}, \mathtt{y_2} = \mathtt{y_1}+\mathtt{1}, \mathtt{z_2} = \mathtt{x_1}+\mathtt{y_1}+\mathtt{1}+\mathtt{z_1}, \mathtt{w_2} = \mathtt{w_1} \\ & \mathtt{path}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{w},\mathtt{k_1},\mathtt{x_1},\mathtt{y_1},\mathtt{z_1},\mathtt{w_2}). \\ \mathtt{path}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{w},\mathtt{k},\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{w}) \leftarrow & \mathtt{k} = \mathtt{0}. \end{array}$$

Given x,y,z,w,k as inputs, the value of z_2 represented as $\mathtt{wh\_aux}^{\mathtt{z}}(x,y,z,w,k)$ is given by the following equation, where we abbreviate wh_aux by f.

$$\mathtt{f}^{\mathtt{z}}(x,y,z,w,k) = \begin{cases} \mathtt{f}^{\mathtt{x}}(x,y,z,w,k-1) + \mathtt{f}^{\mathtt{y}}(x,y,z,w,k-1) + 1 + \mathtt{f}^{\mathtt{z}}(x,y,z,w,k-1), \quad \text{for } k > 0, \\ z, \quad \text{for } k = 0 \end{cases} \tag{9}$$

Assume that we have already computed the following: $\mathtt{f}^{\mathtt{x}}(x,y,z,w,k) = x-k$ and $\mathtt{f}^{\mathtt{z}}(x,y,z,w,k) = y+k$. Reusing them in Equation 9 and simplifying, we obtain Equation 10.

$$\mathtt{f}^{\mathtt{z}}(x,y,z,w,k) = \begin{cases} x+y+1+\mathtt{f}^{\mathtt{z}}(x,y,z,w,k-1), \quad for \quad k > 0, \\ z, \quad for \quad k = 0 \end{cases} \tag{10}$$

Noting that x, y, z and w are symbolic constants, this can be solved to yield $\mathtt{wh}^{\mathtt{z}}(x,y,z,k) = z + k * (x + y + 1)$. Now, mapping the results back to the original clause, we will have $k \in [\,0, x_1]$ and the value of z_1 after the termination of the loop in $[\,z_1, z_1 + x_1 * (x_1 + y_1 + 1)\,]$, which is the solution of Equation 8.

5 Solving the running example using interval arithmetic

We have now described all the necessary components for solving the running example in Fig. 1. We proceed to solve it by solving the corresponding path clauses in Fig. 3 in the reverse topological order of its SCC graph. That is, we first solve clauses corresponding to $\mathtt{wh_2}$, and then clauses for $\mathtt{wh_5}$

and finally path. Solving the clauses 2 and 3 following the procedure described above, we obtain $\mathtt{wh}_2(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}') \leftarrow \mathtt{a}' = \mathtt{a}, \mathtt{b}' = \mathtt{b} - \mathtt{k}_1, 0 \leq \mathtt{k}_1 \leq \mathtt{b}$ where $\mathtt{k}_1$ is the counter of the loop which is bounded from above by its ranking ranking function b. Now, clause 5, after reusing the solution of $\mathtt{wh}_2(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}')$, becomes a linear clause

$$\mathtt{wh}_5(\mathtt{a},\mathtt{b},\mathtt{a}'',\mathtt{b}'') \leftarrow \mathtt{wh}_5(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}'),\ \mathtt{a}' > 0,\ \mathtt{b}' \leq 0, \mathtt{a}'' = \mathtt{a}' - 1, \mathtt{b}'' = \mathtt{b}' + \mathtt{a}' - \mathtt{k}_1, 0 \leq \mathtt{k}_1 \leq \mathtt{b}' + \mathtt{a}'.$$

Note that the counter variable k_1 is not eliminated but rather carried along. Further, note that, its upper bound can be simplfied to $0 \leq \mathtt{k}_1 \leq \mathtt{a}'$ since $\mathtt{b}' \leq 0$ yielding a simplified clause:

$$\mathtt{wh}_5(\mathtt{a},\mathtt{b},\mathtt{a}'',\mathtt{b}'') \leftarrow \mathtt{wh}_5(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}'),\ \mathtt{a}' > 0,\ \mathtt{b}' \leq 0, \mathtt{a}'' = \mathtt{a}' - 1, \mathtt{b}'' = \mathtt{b}' + \mathtt{a}' - \mathtt{k}_1, 0 \leq \mathtt{k}_1 \leq \mathtt{a}'.$$

Now solving this clause along with clause 4, we get

$$\mathtt{wh}_5(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}') \leftarrow \mathtt{a}' = \mathtt{a} - \mathtt{k}_2, \mathtt{b}' = \mathtt{b} + \frac{1}{2} * \mathtt{k}_2(2 * \mathtt{a} - \mathtt{k}_2 - 2 * \mathtt{k}_1 + 1), 0 \leq \mathtt{k}_1 \leq \mathtt{a}' + 1, 0 \leq \mathtt{k}_2 \leq \mathtt{a}.$$

Finally, we reuse the solution of $\mathtt{wh}_2(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}')$ and $\mathtt{wh}_5(\mathtt{a},\mathtt{b},\mathtt{a}',\mathtt{b}')$ in the path clause obtaining

$$\begin{aligned}\mathtt{path}(\mathtt{wh}(\mathtt{a},\mathtt{b}),\mathtt{true}) \leftarrow{} & \mathtt{a}' = \mathtt{a}, \mathtt{b}' = \mathtt{b} - \mathtt{k}_3, 0 \leq \mathtt{k}_3 \leq \mathtt{b}, \mathtt{a}'' = \mathtt{a}' - \mathtt{k}_2, \\ & \mathtt{b}'' = \mathtt{b}' + \tfrac{1}{2} * \mathtt{k}_2(2 * \mathtt{a}' - \mathtt{k}_2 - 2 * \mathtt{k}_1 + 1), 0 \leq \mathtt{k}_1 \leq \mathtt{a}'' + 1, 0 \leq \mathtt{k}_2 \leq \mathtt{a}',\ \mathtt{a}'' \leq 0.\end{aligned}$$

Note that $\mathtt{a}''$ and $\mathtt{b}''$ represent the values of variables a and b after the loop in Fig. 1 terminates. Since we are in the top-level, we simplify the final expression for $\mathtt{a}''$ and $\mathtt{b}''$ using interval arithmetics by replacing the counter variables with their corresponding bounds. Let us first express $\mathtt{b}''$ in terms of input variables.

$$\begin{array}{rll}\mathtt{b}'' = & \mathtt{b}' - \frac{1}{2} * \mathtt{k}_2(-2 * \mathtt{a}' + \mathtt{k}_2 + 2 * \mathtt{k}_1 - 1) & \\ = & \mathtt{b}' - \frac{1}{2} * \mathtt{a}(-\mathtt{a} + 2 * \mathtt{k}_1 - 1) & (\mathtt{a}' = \mathtt{a}, \mathtt{a}'' = \mathtt{a}' - \mathtt{k}_2, \mathtt{a}'' \leq 0, \mathtt{k}_2 \leq \mathtt{a}' \rightarrow \mathtt{k}_2 = \mathtt{a}) \\ = & \mathtt{b}' + \frac{1}{2} * \mathtt{a}(\mathtt{a} - 2 * \mathtt{k}_1 + 1) & \\ = & \mathtt{b}' + \frac{1}{2} * \mathtt{a}(\mathtt{a} - 2 * [0,1] + 1) & (\mathtt{k}_1 \in [0, \mathtt{a}'' + 1], \mathtt{a}'' = \mathtt{a} - \mathtt{k}_2, \mathtt{k}_2 = \mathtt{a} \rightarrow \mathtt{k}_1 \in [0,1]) \\ = & \mathtt{b}' + \frac{1}{2} * \mathtt{a}([\mathtt{a} - 1, \mathtt{a} + 1]) & \\ = & [0, \mathtt{b}] + [\frac{1}{2} * \mathtt{a}(\mathtt{a} - 1), \frac{1}{2} * \mathtt{a}(\mathtt{a} + 1)] & (\mathtt{b}' = \mathtt{b} - \mathtt{k}_3, \mathtt{k}_3 \in [0, \mathtt{b}] \rightarrow \mathtt{b}' \in [0, \mathtt{b}]) \\ = & [\frac{1}{2} * \mathtt{a}(\mathtt{a} - 1), \mathtt{b} + \frac{1}{2} * \mathtt{a}(\mathtt{a} + 1)] & \end{array}$$

Thus, we have $\frac{1}{2} * \mathtt{a}(\mathtt{a} - 1) \leq \mathtt{b}'' \leq \mathtt{b} + \frac{1}{2} * \mathtt{a}(\mathtt{a} + 1)$ when the program terminates. Similarly, we obtain $\mathtt{a}'' = 0$ as we have $\mathtt{a}' = \mathtt{a}, \mathtt{k}_2 = \mathtt{a},\ \mathtt{a}'' = \mathtt{a}' - \mathtt{k}_2$.

6 Discussion and future work

We presented an approach to solving numerical linear loops with linear or polynomial assignments, in which a central concept is that of a path program, to transform (i) recurrences with multiple recursive cases into recurrences with only single recursive case, and (ii) recurrences with multiple arguments into a set of recurrences for functions having only a single argument. Our approach, besides dealing with these, ensures that all loops in the resulting program are directly recursive (no mutual recursion) and recurrences are expressed in terms of an induction variable k (the path counter) allowing the recurrences to be solved and focussing the analysis problem on finding bounds for k. We suggested that it can be bounded from above by the value of a ranking function of the corresponding loop. These transformations help to overcome typical limitations of CASs, since recurrences for multi-case, multi-argument functions are usually not syntactically supported by these tools. But the success of our method depends on external tools such as ranking function synthesisers and CASs. As another limitation, our approach takes a multi-path loop and turns it into nested single-path loops. The result of such a transformation using regular expressions is linear in the number of paths, since our approach produces one path predicate for each distinct subexpression of the regular path expression [12], and the number of distinct subexpressions in the transformed expressions is linear in the number of paths. However, transforming a given program into a multi-path loop expression of form $(e_1 + \ldots + e_n)^*$ could cause an exponential blow-up in size; for example, the number of paths through a loop consisting of a sequence of conditionals is exponential in the number of conditionals. This may prove to be a bottleneck in real applications. Finally, the transformations are limited to linear clauses since they rely on specific properties of regular expressions, but techniques for linearisation of clauses can sometimes be applied.

The approach seems promising though much work remains, both theoretical and experimental. Our immediate task is experimentation with a first implementation of the transformations, and integration into an existing resource analysis tool [18]. Embedding this approach inside a recurrence solver such as Mathematica [1] could also provide several advantages: additional simplification and pre-solving, early detection of patterns, extrapolation and their propagation, and access to an efficient implementation.

With regard to the removal of multi-path loops through transformations based on a regular path expression, we have made some preliminary investigations on choosing good regular expression transformations. There are several (if not an infinite number) of valid equivalent transformations that induce different path programs. The choice of such expression can affect precision of the analysis (though not its soundness). The role of lexicographical ranking functions for multi-path loops seems to be a promising aspect to investigate. For instance, for a loop whose regular expression is $(a+b)^*$, representing a loop with two paths a and b, suppose there is a lexicographical ranking function $\langle r_a, r_b \rangle$, where r_a and r_b are the ranking functions for paths a and b respectively. That is, the sequence of pairs $\langle r_a(x_1), r_b(x_1) \rangle, \langle r_a(x_2), r_b(x_2) \rangle, \ldots$ is lexicographically ordered, where $x_1, x_2, \ldots$ is the sequence of values of loop variables in successive loop iterations. In this case, the replacement expression $b^*(ab^*)^*$ seems better than the equivalent $a^*(ba^*)^*$. We might also compute a refined expression (possibly not equivalent to the original) that produces sound results (preserving the feasible paths) but is more precise than the original. The latter has been studied in [5], and we believe that it is closely related to control-flow refinement for resource and termination analysis [10] or program specialisation, as performed in polyvariant analyzers [25, 30].

With regard to elimination of multi-argument functions in recurrences, replacing them by single-argument functions of a loop counter, the key issue to investigate is how to improve the bounds on the counter's value. We are using a ranking function, but this could be combined with other techniques, such as deriving invariants from the whole program, to improve the bounds. Interval arithmetic, which we used here, in general also loses precision, but could be combined with other analyses. The form of the chosen regular path expression can also play a role in the analysis of the bounds. For example, the path expression for a loop contains an ε expression (an empty path) as the "base case". Thus the encoding loses the connection between the loop and its original base case, which is moved to another path. Other regular expressions for loops could address this issue.

Extending our approach to extract and solve recurrence inequations as in [11] is another avenue for future work. We are also interested in extending the current work to handle non-linear recursions either natively (possibly transforming context free grammars) or through iterative linearisation of the source clauses as in [20].

Acknowledgements. We thank the anonymous reviewers for their constructive comments and for bringing related work to our attention. Research partially funded by MICINN PID2019-108528RB-C21 *ProCode* project, the Madrid P2018/TCS-4339 *BLOQUES-CM* program and the Tezos foundation.

References

[1] *Mathematica: the Way the World Calculates*. `http://www.wolfram.com/products/mathematica/index.html`.

[2] Elvira Albert, Samir Genaim & Abu Naser Masud (2013): *On the Inference of Resource Usage Upper and Lower Bounds*. *ACM Trans. Comput. Log.* 14(3), pp. 22:1–22:35. Available at `https://doi.org/10.1145/2499937.2499943`.

[3] R. Bagnara, F. Mesnard, A. Pescetti & E. Zaffanella (2010): *The Automatic Synthesis of Linear Ranking Functions: The Complete Unabridged Version*. Quaderno 498, Dipartimento di Matematica, Università di Parma, Italy. Available at `http://www.cs.unipr.it/Publications/`.

[4] Rod M. Burstall & John Darlington (1977): *A transformation system for developing recursive programs*. *Journal of the ACM* 24(1), pp. 44–67. Available at `https://doi.org/10.1145/356635.356640`.

[5] John Cyphert, Jason Breck, Zachary Kincaid & Thomas W. Reps (2019): *Refinement of path expressions for static analysis*. POPL, pp. 45:1–45:29. Available at `https://doi.org/10.1145/3290358`.

[6] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2015): *Semantics-based generation of verification conditions by program specialization*. In: *PPDP*, ACM, pp. 91–102. Available at `https://doi.org/10.1145/2790449.2790529`.

[7] S. K. Debray & N. W. Lin (1993): *Cost Analysis of Logic Programs*. *ACM Transactions on Programming Languages and Systems* 15(5), pp. 826–875. Available at `https://doi.org/10.1145/361002.361016`.

[8] S. K. Debray, N.-W. Lin & M. V. Hermenegildo (1990): *Task Granularity Analysis in Logic Programs*. In: *PLDI*, ACM Press, pp. 174–188. Available at `https://doi.org/10.1145/93548.93564`.

[9] Saumya K. Debray & Nai-Wei Lin (1993): *Cost Analysis of Logic Programs*. *ACM Trans. Program. Lang. Syst.* 15(5), pp. 826–875. Available at `https://doi.org/10.1145/161468.161472`.

[10] Jesús J. Doménech, John P. Gallagher & Samir Genaim (2019): *Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis*. *TPLP* 19(5-6), pp. 990–1005. Available at `https://doi.org/10.1017/S1471068419000310`.

[11] Azadeh Farzan & Zachary Kincaid (2015): *Compositional Recurrence Analysis*. In Roope Kaivola & Thomas Wahl, editors: *FMCAD*, IEEE, pp. 57–64. Available at `https://doi.org/10.5555/2893529.2893544`.

[12] J. Gallagher, M. V. Hermenegildo, B. Kafle, M. Klemen, P. Lopez-Garcia & J.F. Morales (2020): *From big-step to small-step semantics and back with interpreter specialization (invited paper)*. In: *VPT*, EPTCS, Open Publishing Association, pp. 50–65. Available at `https://doi.org/10.4204/EPTCS.320.4`.

[13] M. Gómez-Zamalloa, E. Albert & G. Puebla (2009): *Decompilation of Java Bytecode to Prolog by Partial Evaluation*. *JIST* 51, pp. 1409–1427. Available at `https://doi.org/10.1016/j.infsof.2009.04.010`.

[14] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *PLDI*, ACM, pp. 405–416. Available at `https://doi.org/10.1145/2254064.2254112`.

[15] Sumit Gulwani, Sagar Jain & Eric Koskinen (2009): *Control-flow refinement and progress invariants for bound analysis*. In Michael Hind & Amer Diwan, editors: *PLDI*, ACM, pp. 375–385. Available at `https://doi.org/10.1145/1543135.1542518`.

[16] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli & Jorge A. Navas (2015): *The SeaHorn Verification Framework*. In: *CAV*, *LNCS* 9206, Springer, pp. 343–361. Available at `https://doi.org/10.1007/978-3-319-21690-4_20`.

[17] Kim S. Henriksen & John P. Gallagher (2006): *Abstract Interpretation of PIC Programs through Logic Programming*. In: *SCAM '06*, IEEE Computer Society, pp. 184–196. Available at `https://doi.org/10.1109/SCAM.2006.1`.

[18] M. Hermenegildo, G. Puebla, F. Bueno & P. Lopez Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. Science of

Computer Programming 58(1–2), pp. 115–140. Available at `https://doi.org/10.1016/j.scico.2005.02.006`.

[19] Andreas Humenberger, Maximilian Jaroschek & Laura Kovács (2018): *Invariant Generation for Multi-Path Loops with Polynomial Assignments*. In Isil Dillig & Jens Palsberg, editors: *VMCAI*, *LNCS* 10747, Springer, pp. 226–246. Available at `https://doi.org/10.1007/978-3-319-73721-8_11`.

[20] Bishoksan Kafle, John P. Gallagher & Pierre Ganty (2016): *Solving non-linear Horn clauses using a linear Horn clause solver*. In John P. Gallagher & Philipp Rümmer, editors: *HCVS*, *EPTCS* 219, pp. 33–48. Available at `https://doi.org/10.4204/EPTCS.219.4`.

[21] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez & Martin Schäf (2016): *JayHorn: A Framework for Verifying Java Programs*. In Swarat Chaudhuri & Azadeh Farzan, editors: *CAV*, *LNCS* 9779, Springer, pp. 352–358. Available at `https://doi.org/10.1007/978-3-319-41528-4_19`.

[22] Zachary Kincaid, Jason Breck, John Cyphert & Thomas W. Reps (2019): *Closed forms for numerical loops*. Proc. ACM Program. Lang. 3(POPL), pp. 55:1–55:29. Available at `https://doi.org/10.1145/3290368`.

[23] Zachary Kincaid, John Cyphert, Jason Breck & Thomas W. Reps (2018): *Non-linear reasoning for invariant synthesis*. Proc. ACM Program. Lang. 2(POPL), pp. 54:1–54:33. Available at `https://doi.org/10.1145/3158142`.

[24] M. Méndez-Lojo, J. Navas & M. Hermenegildo (2007): *A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs*. In: *LOPSTR*, *LNCS* 4915, Springer-Verlag, pp. 154–168. Available at `https://doi.org/10.1007/978-3-540-78769-3_11`.

[25] K. Muthukumar & M. Hermenegildo (1990): *Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs*. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759. Available at `ftp://cliplab.org/pub/papers/tr153-90.mcc.ps.Z`.

[26] J. Navas, E. Mera, P. Lopez-Garcia & M. Hermenegildo (2007): *User-Definable Resource Bounds Analysis for Logic Programs*. In: *ICLP*, *LNCS* 4670, Springer, pp. 348–363. Available at `https://doi.org/10.1007/978-3-540-74610-2_24`. 10-year Test of Time Award.

[27] Flemming Nielson, Hanne Riis Nielson & Chris Hankin (1999): *Principles of program analysis*. Springer. Available at `https://doi.org/10.1007/978-3-662-03811-6`.

[28] J.C. Peralta, J. Gallagher & H. Sağlam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs*. In G. Levi, editor: *SAS*, *LNCS* 1503, pp. 246–261. Available at `https://doi.org/10.1007/3-540-49727-7_15`.

[29] A. Podelski & A. Rybalchenko (2004): *A Complete Method for the Synthesis of Linear Ranking Functions*. In: *VMCAI*, LNCS, Springer, pp. 239–251. Available at `https://doi.org/10.1007/978-3-540-24622-0_20`.

[30] G. Puebla & M. V. Hermenegildo (1999): *Abstract Multiple Specialization and its Application to Program Parallelization*. J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs 41(2&3), pp. 279–316. Available at `https://doi.org/10.1016/S0743-1066(99)00031-X`.

[31] Rahul Sharma, Isil Dillig, Thomas Dillig & Alex Aiken (2011): *Simplifying Loop Invariant Generation Using Splitter Predicates*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV*, *LNCS* 6806, Springer, pp. 703–719. Available at `https://doi.org/10.1007/978-3-642-22110-1_57`.

[32] Kirack Sohn & Allen Van Gelder (1991): *Termination detection in logic programs using argument sizes (extended abstract)*. In: *PODS*, ACM Press, New York, NY, USA, pp. 216–226. Available at `https://doi.org/10.1145/113413.113433`.

[33] Robert E. Tarjan (1981): *Fast Algorithms for Solving Path Problems*. J. ACM 28(3), pp. 594–614. Available at `https://doi.org/10.1145/322261.322273`.

[34] B. Wegbreit (1974): *The Treatment of Data Types in EL1*. Comm. of the ACM 17(5), pp. 251–264. Available at `https://doi.org/10.1145/364063.364092`.