Resource Analysis of Complex Programs with Cost Equations

Antonio Flores-Montoya and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science
`aflores|haehnle@cs.tu-darmstadt.de`

Abstract. We present a novel static analysis for inferring precise complexity bounds of imperative and recursive programs. The analysis operates on cost equations. Therefore, it permits uniform treatment of loops and recursive procedures. The analysis is able to provide precise upper bounds for programs with complex execution flow and multi-dimensional ranking functions. In a first phase, a combination of control-flow refinement and invariant generation creates a representation of the possible behaviors of a (possibly inter-procedural) program in the form of a set of execution patterns. In a second phase, a cost upper bound of each pattern is obtained by combining individual costs of code fragments. Our technique is able to detect dependencies between different pieces of code and hence to compute a precise upper bounds for a given program. A prototype has been implemented and evaluated to demonstrate the effectiveness of the approach.

1 Introduction

Automatic resource analysis of programs has been subject to intensive research in recent years. This interest has been fuelled by important advances in termination proving, including not only ranking function inference [6, 15], but complete frameworks that can efficiently prove termination of complex programs [3, 7, 10]. Termination proving is, however, only one aspect of resource bound inference.

There are several approaches to obtain upper bounds for imperative programs [3, 8, 9, 11–14, 16, 17]. Most pay little attention to interprocedural, in particular, to recursive programs. Only SPEED [13] and the recent paper [8] address recursive procedures. The extent to which SPEED can deal with complex recursive procedures is hard to evaluate (they provide only one example). The approach of [8] ignores the output of recursive calls which, however, can be essential to obtain precise bounds (see Fig.1).

A different line of work is based on *Cost Equations*, a particular kind of *non-deterministic recurrence relations*, annotated with constraints. This is the approach followed by the COSTA group [1, 2, 4, 5]. One advantage of Cost Equations is that they can deal with both loops and recursion in a *uniform* manner. However, the approach does not cope well with loops that exhibit multiple phases or with programs whose termination proof requires multiple linear ranking functions for a single loop/recursive procedure.

We use the program in Fig.1 to illustrate some of the problems we address in this paper. The program is annotated with structured comments containing cost labels of the form [*Cost x*]. These indicate that at the given program point x resource units are consumed. The program consists of two methods. Method `move` behaves differently depending on the value of boolean variable `fwd`. If `fwd` is true, it may call itself recursively with $n' = n + 1$ and consume two resource units. If `fwd` is false, it may call itself with $n' = n - 1$ and consume one resource unit. Method `main` has a loop that calls `move` and updates the value of n with the result of the call. Additionally, at any iteration, it can change the value of `fwd` to true.

Program 1

```
1 main(int m,int n){
2  //assume(m>n>0)
3  bool fwd=false;
4  while(n > 0){
5     n=move(n,m,fwd);
6     if(?) fwd=true;
7  }
8 }
9 int move(int n,m,bool fwd){
10  if(fwd){
11    if(m > n && ?){
12       ...; //[Cost 2]
13       return move(n+1,m,fwd);
14    }
15  }else{
16    if(n > 0 && ?){
17      ...; //[Cost 1]
18      return move(n-1,m,fwd);
19    }
20  }
21  return n;
22 }
```

Fig. 1. Program example

This example is challenging for several reasons: (i) `move` behaves differently depending on the value of `fwd`, so we ought to analyse its different behaviors separately; (ii) the return value of `move` influences the subsequent behavior of the `main` method and has to be taken into account; (iii) the `main` method might not terminate and yet its cost is finite. Moreover, the upper bound of terminating and non-terminating executions is different. Below we present a table that summarizes the possible upper bounds of this program.

Execution pattern	(1)	(2)	(3)
Upper bound	$n + 2m$	$2(m - n)$	n
Terminating	×	×	✓

Pattern (1) occurs when `move` decrements `n` for a while but without reaching 0 (the initial `n` is an upper bound of the cost); then the guard in line 6 is true and `move` increases `n` up to `m`, incurring a cost of $2m$. The loop in `main` never terminates because n does not reach 0. In pattern (2) the guard in line 6 is true at the beginning and `move` increases `n` to `m` consuming $2 * (m - n)$. Finally, in pattern (3), the guard in line 6 is never true (or only when $n = 0$). Then `move` decrements `n` to 0 and the main loop may terminate, consuming n resource units.

The techniques presented in our paper can deal fully automatically with complex examples such as the program above. Our main contributions are: first, a static analysis for both imperative and (linearly) recursive programs that can infer precise upper bounds for programs with complex execution patterns as above. The analysis combines a control-flow refinement technique in the abstract context of cost equations and a novel upper bound inference algorithm. The latter exploits dependencies between different parts of a program during the computation of upper bounds and it takes into account multiple upper bound candidates at the same time. Second, we provide an implementation of our approach. It is publicly available (see Sec. 6) and it has been evaluated in comparison with KoAT [8], PUBS [1] and Loopus[16]. The experimental evaluation shows how the analysis deals with most examples presented as challenging in the literature.

2 Cost Equations

In this section, we introduce the necessary concepts for the reasoning with cost equations. The symbol $\overline{x}$ represents a sequence of variables $x_1, x_2, \cdots, x_n$ of any length. The expression $vars(t)$ denotes the set of variables in a generic term t. A variable assignment $\alpha : V \mapsto D$ maps variables from the set of variables V to elements of a domain D and $\alpha(t)$ denotes the replacement of each $x \in vars(t)$ by $\alpha(x)$. A *linear expression* has the form $q_0 + q_1 * x_1 + \cdots + q_n * x_n$ where $q_i \in \mathbb{Q}$ and $x_1, x_2, \cdots, x_n$ are variables. A *linear constraint* is $l_1 \leq l_2$, $l_1 = l_2$ or $l_1 < l_2$, where l_1 and l_2 are linear expressions. A *cost constraint* φ is a conjunction of linear constraints $l_1 \wedge l_2 \wedge \cdots \wedge l_n$. The expression $\varphi(\bar{x})$ represents a cost constraint φ instantiated with the variables $\bar{x}$. A cost constraint φ is *satisfiable* if there exists an assignment $\alpha : V \mapsto \mathbb{Z}$ such that $\alpha(\varphi)$ is valid (α satisfies φ).

Definition 1 (Cost expression). *A* cost expression e *is defined as:*

$$e ::= q \mid nat(l) \mid e + e \mid e * e \mid nat(e - e) \mid \max(S) \mid \min(S)$$

where $q \in \mathbb{Q}^+$, l is a linear expression, S is a non-empty set of cost expressions and $nat(e) = \max(e, 0)$. We often omit $nat()$ wrappings in the examples.

Definition 2 (Cost equation). *A* cost equation c *has the form $\langle C(\overline{x}) = e + \sum_{i=1}^{n} D_i(\overline{y_i}), \varphi\rangle$ ($n \geq 0$), where C and D_i are cost relation symbols; all variables $\overline{x}$, $\overline{y_i}$, and $vars(e)$ are distinct; e is a cost expression; and φ is a conjunction of linear constraints that relate the variables of c.*

A cost equation $\langle C(\overline{x}) = e + \sum_{i=1}^{n} D_i(\overline{y_i}), \varphi\rangle$ states that the cost of $C(\overline{x})$ is e plus the sum of the costs of each $D_i(\overline{y_i})$. The relation φ serves two purposes: it restricts the applicability of the equation with respect to the input variables and it relates the variables $\overline{x}$, $vars(e)$, and $\overline{y_i}$. One can view C as a non-deterministic procedure that calls $D_1, D_2, \ldots, D_n$.

Fig. 2 displays the cost equations corresponding to the program in Fig. 1. To simplify presentation in the examples we reuse some variables in different relation symbols. In the implementation they are in fact different variables with suitable equality constraints in φ.

We restrict ourselves to linear recursion, i.e., we do not allow recursive equations with more than one recursive call. Our approach could be combined with existing analyses for multiple recursion such as the one in [4]. Input and output variables are both included in the cost equations and treated without distinction. By convention, output variable names end with "o" so they can be easily recognized. In a procedure, the output variable corresponds to the return variable (*no*

SCC	*Nr*	*Cost Equation*
S_1	1	$main(n,m) = while(n,m,0) \quad n \geq 1 \wedge m \geq n+1$
S_2	2	$while(n,m,fwd) = 0 \quad n \leq 0$
	3	$while(n,m,fwd) = move(n,m,fwd,no) + while(no,m,fwd) \quad n > 0$
	4	$while(n,m,fwd) = move(n,m,fwd,no) + while(no,m,1) \quad n > 0$
S_3	5	$move(n,m,fwd,no) = 2 + move(n+1,m,fwd,no) \quad fwd = 1 \wedge n < m$
	6	$move(n,m,fwd,no) = 0 \quad fwd = 1 \wedge n = no$
	7	$move(n,m,fwd,no) = 1 + move(n-1,m,fwd,no) \quad fwd = 0 \wedge n > 1$
	8	$move(n,m,fwd,no) = 0 \quad fwd = 0 \wedge n = no$

Fig. 2. Cost equations of the example program from Fig. 1

in the method move). In a loop, the output variables are the local variables that might be modified inside the loop. In the while loop from Fig.2, we would have $while(n, m, fwd, no, fwdo)$ where no and $fwdo$ are the final values of n and fwd, but the cost equations have been simplified for better readability.

Generating Cost Equations Cost equations can be generated from source code or low level representations. Loop extraction and partial evaluation are combined to produce a set of cost equations with only direct recursion [1]. The details are in the cited papers and omitted for lack of space. The resulting system is a sequence of strongly connected components (SCCs) $S_1, \ldots, S_n$ such that each S_i is a set of cost equations of the form $\langle C(\overline{x}) = e + \sum_{j=1}^{k} D_j(\overline{y_j}) + \sum_{j=1}^{n} C(\overline{y_j}), \varphi\rangle$ with $k \geq 0$ and $n \in \{0, 1\}$ and each $D_j \in S_{i'}$ where $i' > i$. Each SCC is a set of directly recursive equations with at most one recursive call and k calls to SCCs that appear later in the sequence. Hence, S_1 is the outermost SCC and entry point of execution while S_n is the innermost SCC and has no calls to other SCCs. Each resulting cost equation is a complete iteration of a loop or recursive procedure.

Example 1. In Fig. 2, the cost equations of Program 1 are grouped by SCC. Each SCC defines only one cost relation symbol: *main*, *while*, and *move* occur in S_1, S_2, and S_3, respectively. However, the cost equations in any SCC may contain references to equations that appear later. For instance, equations 3 and 4 in S_2 have references to *move* in S_3.

A concrete execution of a relation symbol C in a set of cost equations is generally defined as a (possibly infinite) evaluation tree $T = node(r, \{T_1, \ldots T_n\})$, where $r \in \mathbb{R}^+$ is the cost of the root (an instance of the cost expression in C) and $T_1, \ldots T_n$ are sub-trees corresponding to the calls in C. In the following we will not need this general definition. A formal definition of evaluation trees and their semantics is in [1].

3 Control-flow Refinement of Cost Equations

As noted in Sec. 1, we have to generate all possible execution patterns and discard unfeasible patterns that might reduce precision or even prevent us from obtaining an upper bound. Our cost equation representation allows us to look at one SCC at a time. If we consider only the cost equations within one SCC, we have sequences of calls instead of trees (we are only considering SCCs with linear recursion). That does not prevent each cost equation in the sequence from having calls to other SCCs.

Example 2. Given S_3 from Fig. 2, the sequence $5 \cdot 5 \cdot 6$ represents a feasible execution where equation 5 is executed twice followed by one execution of 6. On the other hand, the execution $5 \cdot 8$ is infeasible, because the cost constraints of its elements are incompatible ($fwd = 1$ and $fwd = 0$).

Given an SCC C consisting of cost equations S_C, we can represent its execution patterns as regular expressions over the alphabet of cost equations in S_C. We use a specific form of execution patterns that we call *chain*:

Definition 3 (Phase, Chain). *Let $S_C = c_1, \ldots, c_r$ be the cost equations of an SCC C. A* phase *is a regular expression $(c_{i_1} \vee \ldots \vee c_{i_m})^+$ over S_C (executed a positive number of times). A special case is a phase where exactly one equation is executed: $(c_{i_1} \vee \ldots \vee c_{i_m})$.*

A chain *is a regular expression over S_C composed of a sequence of phases $ch = ph_1 \cdot ph_2 \cdots ph_n$ such that its phases do not share any common equation. That is, if $c \in ph_i$, then $c \notin ph_j$ for all $j \neq i$.*

We say that a cost equation that has a recursive call is *iterative* and a cost equation with no recursive calls is *final*. Given an SCC C consisting of cost equations S_C, we use the name convention $i_1, i_2 \ldots i_n$ for the iterative equations and $f_1, f_2 \ldots f_m$ for the final equations in S_C. All possible executions of an SCC can be summarized in three basic chains: (1) $ch_n = (i_1 \vee i_2 \vee \cdots \vee i_n)^+ \cdot (f_1 \vee f_2 \vee \cdots \vee f_m)$ an arbitrary sequence of iterations that terminates with one of the base cases; (2) $ch_b = (f_1 \vee f_2 \vee \cdots \vee f_m)$ a base case without previous iterations; (3) an arbitrary sequence of iterations that never terminates $ch_i = (i_1 \vee i_2 \vee \cdots \vee i_n)^+$.

Example 3. The basic chains of method *move* (SCC S_3 of Fig.2) are: $ch_n = (5 \vee 7)^+(6 \vee 8)$, $ch_b = (6 \vee 8)$ and $ch_i = (5 \vee 7)^+$. Obviously, these chains include a lot of unfeasible call sequences which we want to exclude.

3.1 Chain Refinement of an SCC

Our objective is to specialize a chain into more refined ones according to the constraints φ of its cost equations. To this end, we need to analyse the possible sequences of phases in a chain. We use the notation $c \in ch$ to denote that the cost equation c appears in the chain ch.

Definition 4 (Dependency). *Let $c, d \in ch$, $c = \langle C(\bar{x}_c) = \ldots + C(\bar{z}), \varphi_c \rangle$, $d = \langle C(\bar{x}_d) = \ldots, \varphi_d \rangle$; then $c \preceq d$ iff the constraint $\varphi_c \wedge \varphi_d \wedge (\bar{z} = \bar{x}_d)$ is satisfiable. Intuitively, $c \preceq d$ iff d can be executed immediately after c. The relation $\preceq^*$ is the transitive closure of $\preceq$.*

We generate new phases and chains according to these dependencies. Define $c \equiv d$ iff $c = d$ (syntactic equality) or $c \preceq^* d$ and $d \preceq^* c$. Each equivalence class in $[c]_\equiv$ gives rise to a new phase. If $[c]_\equiv = \{c\}$ and $c \not\preceq c$, the new phase is (c). If $[c]_\equiv = \{c_1, \ldots, c_n\}$, the new phase is $(c_1 \vee \cdots \vee c_n)^+$. To simplify notation we identify an equivalence class with the phase it generates. Then $ph \prec ph'$ iff $ph \neq ph'$, $c \in ph$, $d \in ph'$ and $c \preceq d$. $ch' = ph_1 \cdots ph_n$ is a *valid chain* iff for all $1 \leq i < n$: $ph_i \prec ph_{i+1}$.

Example 4. The dependency relation of *move* (SCC S_3 from Fig. 2) is the following: $5 \preceq 5$, $5 \preceq 6$, $7 \preceq 7$ and $7 \preceq 8$. This produces the following phases: $(5)^+$, $(7)^+$, (6) and (8), which in turn give rise to chains: non-terminating chains $(5)^+$, $(7)^+$; terminating chains $(5)^+(6)$, $(7)^+(8)$ and the base cases (6), (8). This refinement captures the important fact that the method cannot alternate the behavior that increases n (cost equation 5) with the one that decreases it (cost equation 7).

Theorem 1 (Refinement completeness). *Let $ch_1, \ldots, ch_n$ be the generated chains for a SCC S from the basic chains of S. Any possible sequence of cost equation applications of S is covered by at least one chain ch_i, $i \in 1..n$ (a proof can be found in App A.1).*

3.2 Forward and Backward Invariants

We can use invariants to improve the precision of the inferred dependencies and to discard unfeasible execution patterns. Given a chain $ch = ph_1 \cdots ph_n$ in S_i with C as cost relation symbol, we can infer forward invariants (*fwdInv*) that propagate the context in which the chain is called from ph_1 to the subsequent phases. Additionally, we can propagate the relation between the variables from the final phase ph_n to the previous phases until calling point ph_1, obtaining backward invariants (*backInv*). These invariants provide us with extra information at each phase ph_i coming from the phases that appear before (*fwdInv*) or after (*backInv*) ph_i.

$\mathit{fwdInv}_{ch}(ph_i)$ and $\mathit{backInv}_{ch}(ph_i)$ denote forward and backward invariants valid at any application of the equations in the phase ph_i of chain ch. If it is obvious which chain is referred to, we leave out the subscript ch. The forward invariant at the beginning of a chain ch in an SCC S_i is given by the conditions under which ch is called in other SCCs. The backward invariant at the end of a chain ch is defined by the constraints φ of the base case ph_n for terminating chains. For non-terminating chains, the backward invariant at the end of a chain is the empty set of constraints (*true*). The backward invariant of the first phase of a chain ch represents the input-output relations between the variables. It can be seen as a summary of the behavior of ch. The procedure for computing these invariants can be found in App B.

Additionally, we define φ_{ph} and φ_{ph^*} for iterative phases. The symbol φ_{ph} represents the relation between the variables before and after any positive number of iterations of ph, while φ_{ph^*} represents the relation between the variables before and after zero or more iterations.

Example 5. Some of the inferred invariants for the chains of S_3 of our example:
$\mathit{backInv}_{(5)^+(6)}((5)^+) = fwd = 1 \land m > n \land m \geq no \land no > n$
$\mathit{backInv}_{(7)^+(8)}((7)^+) = fwd = 0 \land n > 0 \land no \geq 0 \land n > no$
These invariants reflect applicability conditions (Such as $fwd = 0$) and the relation between the input and the output variables. For example, $no > n$ holds when n is increased and $n > no$ when it is decreased. The condition $m \geq no$ is derived from the fact that at the end of phase $(5)^+$ we have $m > n$, in phase (6) $n' = no'$ and the transition is $n' = n + 1 \land no' = no$.

We can use forward and backward invariants to improve the precision of the inferred dependencies. At the same time, a more refined set of chains will allow us to infer more precise invariants. Hence, we can iterate this process (chain refinement and invariant generation) until no more precision is achieved or until we reach a compromise between precision and performance. We can also use the inferred invariants to discard additional cost equations or chains. Let $c = \langle C(\bar{x}) = \ldots + C(\bar{z}), \varphi \rangle \in ph_i$, if $\varphi \land \mathit{backInv}_{ch}(\overline{ph_i}) \land \mathit{fwdInv}_{ch}(ph_i)$ is unsatisfiable, c cannot occur and can be eliminated from ph_i in the chain ch. If any invariant belonging to a chain is unsatisfiable its pattern of execution cannot possibly occur and the chain can be discarded.

3.3 Terminating Non-termination

In our refinement procedure, we distinguish terminating and non-terminating chains explicitly. Given a chain $ph_1 \cdots ph_n$, it is assumed that every phase ph_i with $i \in 1..n{-}1$ is terminating. This is safe, because for each ph_i that is iterative we generated another chain of the form $ph_1 \cdots ph_i$, where ph_i is assumed not to terminate. That is, we consider both the case when ph_i terminates and when it does not terminate. Given a non-terminating chain, if we prove termination of its final phase, we can safely discard that chain.

Consider a phase $(c_1 \lor c_2 \lor \ldots \lor c_m)^+$, we obtain a (possibly empty) set of linear ranking functions for each c_i, denoted RF_i, using the techniques of [15, 6]. A linear ranking function of a cost equation $\langle C(\overline{x}) = \cdots + C(\overline{x}'), \varphi \rangle$ with a recursive call $C(\overline{x}')$ is a linear expression f such that (1) $\varphi \Rightarrow f(\overline{x}) \geq 0$ and (2) $\varphi \Rightarrow f(\overline{x}) - f(\overline{x}') \geq 1$.

For each ranking function f of c_i, we check whether its value can be incremented in any other $c_j = \langle C(\overline{x}) = \cdots + C(\overline{x}'), \varphi_j \rangle$, $j \neq i$ (whether $\varphi_j \land f(\overline{x}) - f(\overline{x}') < 0$ is satisfiable). If f can be increased in c_j we say that f depends on c_j. As in [3], the procedure for proving termination consists in eliminating the cost equations that have a ranking function without dependencies first. Then, incrementally eliminate the cost equations that have ranking functions whose dependencies have been already removed until there are no cost equations remaining. The set of ranking functions and their dependencies will be used again later to introduce specific bounds for the number of calls to each c_i.

Example 6. The ranking functions for the phases $(5)^+$ and $(7)^+$ are $m - n$ and n respectively. With such ranking functions, we can discard the non-terminating chains $(5)^+$ and $(7)^+$. The remaining chains are $(5)^+(6)$, $(7)^+(8)$, (7) and (8).

Nr	$Cost\ Equation$
3.1	$while(n,m,fwd) = move_{(5)^+(6)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd = 1} \wedge \mathbf{m > n} \wedge \mathbf{m \geq no} \wedge \mathbf{no > n}$
3.2	$while(n,m,fwd) = move_{(6)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd = 1} \wedge \mathbf{no = n}$
3.3	$while(n,m,fwd) = move_{(7)^+(8)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd = 0} \wedge \mathbf{no \geq 0} \wedge \mathbf{n > no}$
3.4	$while(n,m,fwd) = move_{(8)}(n,m,fwd,no) + while(no,m,fwd)$
	$n > 0 \wedge \mathbf{fwd = 0} \wedge \mathbf{n = no}$

Fig. 3. Refinement of Cost equation 3 from Fig. 2

3.4 Propagating Refinements

The refinement of an SCC S_i in a sequence $S_1, \ldots, S_n$ can affect both predecessors and successors of S_i. The initial forward invariants from SCCs that are called in S_i, the forward invariants of the SCCs $S_{i+1}, \ldots, S_n$ might be strengthened by the refinement of S_i. The preceding SCCs that have calls to S_i can be specialized so they call the refined chains. The backward invariants can be included in the calling cost equations thus introducing a "loop summary" of S_i's behavior.

Each cost equation containing a call to S_i, say $\langle D(\bar{x}) = \ldots + C_{ch}(\bar{z}), \varphi\rangle \in S_j$ with $j < i$, can be replaced with a set of cost equations $\langle D(\bar{x}) = \ldots + C_{ch'}(\bar{z}), \varphi'\rangle$, where $ch' = ph_1 ph_2 \cdots ph_m$ is one of the refined chains of ch, and $\varphi' := \varphi \wedge \mathit{backInv}_{ch'}(\overline{ph_1})$. If φ' is unsatisfiable, the cost equation can be discarded.

Example 7. We propagate the refinement of method *move* (SCC S_3) to *while* (SCC S_2). Fig. 3 shows how cost equation 3 is refined by substituting the calls to *move* by calls to specific chains of *move* and by adding the backward invariants of the callees to its cost constraint φ. Analogously, cost equation 4 is refined into 4.1, 4.2, 4.3, and 4.4. The only difference is that the latter have a recursive call to *while* with $fwd = 1$. The cost equations of *move* are not changed because the do not have calls to other SCCs.

The new phases are $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$, $(3.3 \vee 3.4)^+$, (4.3), (4.4) and (2). Phase $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$ represents iterations of the loop when $fwd = 1$. The fact that fwd is explicitly set to 1 in 4.1 and 4.2 does not have any effect. Phase $(3.3 \vee 3.4)^+$ represents the iterations when $fwd = 0$ and is kept that way in the recursive call. Finally, (4.3) and (4.4) are the cases where fwd is changed from 0 to 1. If we use the initial forward invariant $n \geq 1 \wedge m > n$ of *main* (in SCC S_1), we obtain the following chains:

Pattern (1)	Pattern (2)	Pattern (3)
$(3.3 \vee 3.4)^+(4.3)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(4.3)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(3.3 \vee 3.4)^+(2)$
$(3.3 \vee 3.4)^+(4.4)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(4.4)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$	$(3.3 \vee 3.4)^+$

They are grouped according to the execution patterns that were intuitively presented in Sec. 1. Note that neither $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$ or $(3.3 \vee 3.4)^+$ are always terminating as we can iterate indefinitely on 3.2, 4.2 and 3.4. These cases correspond to a call to *move* that immediately returns without modifying n. Therefore, we cannot discard any of the non-terminating chains.

4 Upper Bound Computation

4.1 Cost Structures

At this point, a refined program consists of a sequence of SCCs $S_1, \ldots, S_n$ where each SCC S_i contains a set of chains. We want to infer safe upper bounds for each chain individually but, at the same time, take their dependencies into account. The standard approach on cost equations [1] consists in obtaining a cost expression that represents the cost of each SCC S_i and substituting any call to that S_i by the inferred cost expression. That way, we can infer closed-form upper bounds for all SCCs in a bottom up approach (From S_n to S_1). This approach turns out not to be adequate to exploit the dependencies between different parts of the code as we illustrate in the next example.

Example 8. Let us obtain an upper bound for method *main* when it behaves as in chain $(3.3 \vee 3.4)^+(2)$. This is a simple pattern, where *move* only increases or leaves n unchanged. Following the standard approach, we first obtain the upper bound for *move* when called in 3.3 and 3.4, that is, when *move* behaves as in $(7)^+(8)$ and (8). By multiplying the maximum number of recursive calls with the maximum cost of each call the upper bound we obtain is n and 0, respectively. The cost of $(3.3 \vee 3.4)^+(2)$ is then the maximum cost of each iteration n multiplied by the maximum number of iterations. However, 3.4 can iterate indefinitely, so we fail to obtain an upper bound.

If we apply the improved method of [4] after the refinement, we consider 3.3 and 3.4 independently. Phase 3.3 has zero cost and 3.4 has a ranking function n, yielding a bound of n^2 for this chain (while a more precise bound is n).

To overcome this problem, we define a new upper bound computation method based on an intermediate structure that summarizes all the cost components while maintaining part of the internal structure of what generated the cost.

Definition 5 (Cost Structure). *A* cost structure *CT is a pair $SE : CS$. Here SE is a cost expression of the form $SE = \sum_{i=1}^{n} SE_i * iv_i + e$ $(n \geq 0)$, where e is a cost expression and iv_i is a symbolic variable representing a natural number. We refer to the iv_i as* iteration variables*, to a product $SE_i * iv_i$ as* iteration component *and to SE as* structured cost expression*. CS is a (possibly empty) set of constraints of the form $\sum_{j=1}^{m} iv_j \leq e$ $(m \geq 1)$, such that all its iteration variables appear in SE. The constraints relate iteration variables with cost expressions. We use the notation $\sum iv \leq e$ when the number of iteration variables is irrelevant.*

Intuitively, a structured cost expression represents a fixed cost e plus a set of iterative components $SE_i * iv_i$, where each iterative component is executed

iv_i times and each iteration has cost SE_i. The set of constraints CS binds the values of the iteration variables iv and can express dependencies among iteration components. For instance, a constraint $iv_1 + iv_2 \leq e$ expresses that the iteration components iv_1 and iv_2 are bound by e and that the bigger iv_1 is, the smaller iv_2 must be.

We denote with IV the set of iteration variables in a cost structure. Let $val : IV \to E$ be an assignment of the iteration variables to cost expressions, a valid cost of a cost structure $CT = \sum_{i=1}^{n} SE_i * iv_i + e : CS$ is defined as $val(SE) = \sum_{i=1}^{n} val(SE_i) * val(iv_i) + e$ such that $val(CS)$ is valid.[1] A cost structure can represent multiple upper bound candidates.

Example 9. Consider a cost structure $a*iv_1+b*iv_2+c : \{iv_1 \leq d, iv_1+iv_2 \leq e\}$ where a, b, c, d, and e are cost expressions. If $a > b$ and $d < e$, an upper bound is $a*d+b*nat(e-d)+c$ (The $nat()$ wrapping can be omitted). In case of $a < b$, an upper bound is $b*e+c$.

We follow a bottom up approach from S_n to S_1 and infer cost structures for cost equations, phases and chains, detailed in Secs. 4.3, 4.4, and 4.5 below. Sec. 4.2 contains a complete example. In Sec. 5, we present a technique to obtain maximal cost expressions from cost structures. They key of the procedure is to safely combine individual cost structures while detecting dependencies among them. The intermediate cost structures are correct, that is, at the end of our analysis of our example (Fig. 1) we will not only have upper bounds of *main* but also a correct upper bound of *move*.

We define the operations that form the basis or our analysis.

Definition 6 (Cost Expression Maximization). *Given a cost expression e, a cost constraint φ, and a set of variables $\overline{v}$, the operation $bd(e, \varphi, \overline{v})$ returns a set E of cost expressions that only contain variables in $\overline{v}$ and that are safe upper bounds. That is, for each $e' \in E$, we have that for all variable assignments to integers $\alpha : vars(e') \cup vars(e) \to \mathbb{Z}$ that satisfy φ: $\alpha(e') \geq \alpha(e)$. It is possible that $bd(e, \varphi, \overline{v})$ returns the empty set. In this case, no finite upper bound is known.*

For $bd(e, \varphi, \bar{v}) = \{e_1, \ldots, e_n\}$ define $\min(bd(e, \varphi, \bar{v})) = \min(e_1, \ldots, e_n)$. Note that if $bd(e, \varphi, \bar{v}) = \emptyset$, $\min(bd(e, \varphi, \bar{v})) = \infty$. Cost expression maximization can be implemented using geometrical projection over the dimensions of $\bar{v}$ in the context of the polyhedra abstract domain or as existential quantification of the variables of e and φ that do not appear in $\bar{v}$. This operation is done independently for each l in the cost expression. The results can be safely combined as linear expressions appear always inside a $nat()$ in cost expressions.

Definition 7 (Structured Cost Expression Maximization). *We define recursively the bound of a structured cost expression as $Bd(\sum_{i=1}^{n} SE_i * iv_i + e, \varphi, \bar{v}) = \sum_{i=1}^{n} Bd(SE_i, \varphi, \bar{v}) * iv_i + \min(bd(e, \varphi, \bar{v}))$.*

[1]Cost structures have some similarities to the multiple counter instrumentation described in [13]. Iteration variables can be seen as counters for individual loops or recursive components and constraints represent dependencies among these counters.

SCC	*Chain*	*Execution*	
2	$(3.3 \vee 3.4)^+(2)$	$c_{3.?}(\overline{x}_1) \rightarrow \cdots c_{3.3}(\overline{x}_i) \rightarrow \cdots \rightarrow c_{3.?}(\overline{x}_f) \rightarrow$	$c_2(\overline{x}_{f+1})$
		$\downarrow \cdots \qquad \downarrow \qquad \cdots \downarrow$	
3	$(7)^+(8)$	$c_7(\overline{y}_1) \rightarrow \cdots \rightarrow c_7(\overline{y}_f) \rightarrow$	$c_8(\overline{y}_{f+1})$

Fig. 4. Schema of executing chain $(3.3 \vee 3.4)^+(2)$

4.2 Example of upper bound computation

Fig. 4 represents the execution of chain $(3.3 \vee 3.4)^+(2)$. The execution of the phase $(3.3 \vee 3.4)^+$ consists on a series of applications of either 3.3 or 3.4. Each equation application has a call to *move*. In particular, 3.3 calls $move_{(7)^+(8)}$ and 3.4 calls $move_{(8)}$. In Fig. 4, only one call to $move_{(7)^+(8)}$ is represented. $c_n(\overline{x})$ represents an instance of cost equation n with variables $\overline{x}$.

Cost of move In order to compute the cost of the complete chain, we start by computing the cost of the innermost SCCs. In this case, the cost of *move*. The cost of one application of 8 ($c_8(\overline{y}_{f+1})$) and 7 ($c_7(\overline{y}_i)$) are 0 and 1 respectively (taken directly from the cost equations in Fig. 2). The cost of phase $(7)^+$ is the sum of the costs of all applications of c_7: $c_7(\overline{y}_1), c_7(\overline{y}_2), \cdots, c_7(\overline{y}_f)$. If c_7 is applied iv_7 times, the total cost will be $1 * iv_7$. Instead of giving a concrete value to iv_7, we collect constraints that bind its value and build a cost structure. In Sec. 3.3 we obtained the ranking function n for 7 so we have $iv_7 \leq nat(n_1)$. Moreover, the number of iterations is also bounded by $nat(n_1 - n_f)$, the difference between the initial and the final value of n in phase $(7)^+$ (see Lemma 1). Consequently, the cost structure for $(7)^+$ is $1 * iv_7 : \{iv_7 \leq n_1, iv_7 \leq n_1 - n_f\}$ (we omit the $nat()$ wrappings). If we had more ranking functions for 7, we could add extra constraints. This is important because we do not know yet which ranking function will yield the best upper bound. Additionally, we keep the cost per iteration and the number of iterations separated so we can later reason about them independently (detect dependencies). The cost of $(7)^+(8)$ is the cost of $(7)^+$ plus the cost of (8) but expressed according to the initial variables $\overline{y}_1$. We add the cost structures and maximize them (Bd) using the corresponding invariants. We obtain $1 * iv_7 : \{iv_7 \leq n_1, iv_7 \leq n_1 - no_1\}$ (because $n_f > n_{f+1} = no_{f+1} = no_1$).

Cost of one application of 3.3, 3.4 *and* 2 The cost of (2) is 0. The cost of one application of 3.4 is the cost of a call to $move_{(8)}$, that is, 0. Conversely, the cost of one application of 3.3 is the cost of one call to $move_{(7)^+(8)}$. We want the cost of $c_{3.3}(\overline{x}_i)$ expressed in terms of the entry variables $\overline{x}_i$ and the variables of the corresponding recursive call $\overline{x}_{i+1}$. We maximize the cost structure of $move_{(7)^+(8)}$ using the cost constraints of 3.3 ($\varphi_{3.3}$). This results in the cost structure $1 * iv_7 : \{iv_7 \leq n_i, iv_7 \leq n_i - n_{i+1}\}$ (the output *no* is n_{i+1} in the recursive call).

Cost of phase $(3.3 \vee 3.4)^+$ The cost of phase $(3.3 \vee 3.4)^+$ is the sum of the cost of all applications of $c_{3.3}$ and $c_{3.4}$: $c_{3.?}(\overline{x}_1), c_{3.?}(\overline{x}_2), \cdots, c_{3.?}(\overline{x}_f)$. We group the summands originating from 3.3 and from 3.4 and assume that $c_{3.3}$ and $c_{3.4}$ are applied $iv_{3.3}$ and $iv_{3.4}$ times respectively. The sum of all applications of $c_{3.4}$ is $0{*}iv_{3.4} = 0$. However, the cost of each $c_{3.3}(\overline{x}_i)$ might be different (depends on $\overline{x}_i$) so we cannot simply multiply. Using the invariant $\varphi_{(3.3\vee 3.4)^*}$ and $\varphi_{3.3}$ we know that $n_1 \geq n_i \wedge n_i > n_{i+1} \wedge n_{i+1} \geq 0$. Maximizing each of these constraints yields $iv_7 \leq n_1$ and we obtain a cost structure $1 * iv_7 : \{iv_7 \leq n_1\}$ that is greater or equal than all $1 * iv_7 : \{iv_7 \leq n_i, iv_7 \leq n_i - n_{i+1}\}$ (because $n_1 \geq n_i$). Therefore, a valid (but imprecise) cost of $(3.3 \vee 3.4)^+$ is $(1 * iv_7) * iv_{3.3} : \{iv_7 \leq n_1, iv_{3.3} \leq n_1, iv_{3.3} \leq n_1 - n_f\}$ (n is a ranking function of 3.3). If we solve the cost structure, we will obtain the upper bound n^2.

Inductive constraint compression Because we kept the different components of the cost separated, we can easily obtain a more precise cost structure Each call to *move* starts where the last one left it and all of them together can iterate at most n times. This is reflected by the constraint $iv_7 \leq n_i - n_{i+1}$. We can compress all the iterations $(n_1 - n_2) + (n_2 - n_3) + \cdots + (n_{f-1} - n_f) \leq n_1 - n_f$, pull out the iteration component $1 * iv_7$ and obtain a more precise cost structure $(1 * iv_7) + (0 * iv_{3.3}) : \{iv_7 \leq n_1 - n_f, iv_{3.3} \leq n_1, iv_{3.3} \leq n_1 - n_f\}$. Then, we can eliminate $(0 * iv_{3.3})$ arriving at $(1 * iv_7) : \{iv_7 \leq n_1 - n_f\}$ which will result in an upper bound n.

4.3 Cost Structure of an Equation Application

Consider a cost equation $c = \langle C(\bar{x}) = \sum_{i=1}^{n} D_i(\bar{y_i}) + e + C(\bar{x}'), \varphi\rangle$, where $C(\bar{x}')$ is a recursive call. We want to obtain a cost structure $SE_c : CS_c$ that approximates the cost of $\sum_{i=1}^{n} D_i(\bar{y_i}) + e$ and we want such a cost structure to be expressed in terms of $\bar{x}$ and $\bar{x}'$.

Example 10. Consider cost equation 3.3 from Fig. 3 which is part of SCC S_2:

$$while(n, m, fwd) = move_{(7)^+(8)}(n'', m'', fwd'', no) + while(n', m', fwd')$$

Assume φ contains $n'' = n \wedge n' = no$. The cost of one application of 3.3 is the cost of $move_{(7)^+(8)}(n, m, fwd, no)$ expressed in terms of n, m, fwd and n', m', fwd'. Let the cost of $move_{(7)^+(8)}$ be $1 * iv_7 : \{iv_7 \leq n'', iv_7 \leq n'' - no\}$, then we obtain an upper bound by maximizing the structured cost expression and the constraints in terms of the variables n, m, fwd and n', m', fwd'. The obtained cost structure is $1 * iv_7 : \{iv_7 \leq n, iv_7 \leq n - n'\}$.

Let $SE_i : CS_i$ be the cost structure of the chain D_i, then the structured cost expression can be computed as $SE_c = \sum_{i=1}^{n} Bd(SE_i, \varphi, \bar{x}) + min(bd(e, \varphi, \bar{x}))$. By substituting each call $D_i(\bar{y_i})$ by its structured cost expression and maximizing with respect to $\bar{x}$, we obtain a valid structured cost expression in terms of the entry variables.

A set of valid constraints CS_c is obtained simply as the union of all sets CS_i expressed in terms of the entry and recursive call variables ($\bar{x}$ and $\bar{x}'$): $CS_c \supseteq \{\sum iv \leq e' | \sum iv \leq e \in CS_i, e' \in bd(e, \varphi, \bar{x}\bar{x}')\}$. Should the cost equation not have a recursive call, all the maximizations will be performed only with respect to the entry variables $\bar{x}$.

Constraint Compression In order to obtain tighter bounds, one can try to detect dependencies among the constraints when they have a linear cost expression. Let $\sum iv_i \leq nat(l_i) \in CS_i$ and $\sum iv_j \leq nat(l_j) \in CS_j$, $j \neq i$. Now assume there exist $l_{new} \in bd(l_i + l_j, \varphi, \bar{x}\bar{x'})$, $l'_i \in bd(l_i, \varphi, \bar{x}\bar{x'})$, and $l'_j \in bd(l_j, \varphi, \bar{x}\bar{x'})$ such that $\varphi \Rightarrow (l_{new} \leq (l'_i + l'_j) \wedge l_{new} \geq l_i \wedge l_{new} \geq l_j)$. $nat(l_{new})$ might bind $nat(l_i)$ and $nat(l_j)$ tighter than $nat(l'_i)$ and $nat(l'_j)$. Then we can add $\sum iv_i + \sum iv_j \leq nat(l_{new})$ to the new set of constraints CS_c.

Example 11. Suppose the cost equation from the previous example had two consecutive calls to *move*: $while(n, m, fwd) = move_{(7)^+(8)}(n_1, m_1, fwd_1, no_1) + move_{(7)^+(8)}(n_2, m_2, fwd_2, no_2) + while(n', m', fwd')$ with $\{n_1 = n \wedge no_1 = n_2 \wedge no_2 = n'\} \subseteq \varphi$. The resulting cost structure would be $1 * iv_{7.1} + 1 * iv_{7.2} * 2$: $\{iv_{7.1} \leq n, iv_{7.1} \leq n - n', iv_{7.2} \leq n, iv_{7.2} \leq n - n'\}$ ($iv_{7.1}$ and $iv_{7.2}$ correspond to the iterations of the two instances of phase $(7)^+$). However, we could compress $iv_{7.1} \leq n_1 - no_1$ and $iv_{7.2} \leq n_2 - no_2$ (from Ex. 10) into $iv_{7.1} + iv_{7.2} \leq n - n'$ and add it to the final set of constraints. This set represents a tighter bound and captures the dependency between the first and the second call.

4.4 Cost Structure of a Phase

Refined phases have the form of a single equation (c) or an iterative phase $(c_1 \vee c_2 \vee \ldots \vee c_n)^+$. The cost of (c) is simply the cost of c. The cost of an iterative phase is the sum of the costs of all applications of each c_i (see Sec. 4.2). Let $CT_i = SE_i : CS_i$ be the cost of one application of c_i, we group the summands according to each c_i and assign a new iteration variable iv_i that represents the number of times such a cost equation is applied. The total cost of the phase is $\sum_{i=1}^{n}(\sum_{j=1}^{iv_i} SE_i(x_j))$ where $SE_i(x_j)$ is an instance of SE_i with the variables corresponding to the j-th application of c_i.

For each c_i in the phase $(c_1 \vee c_2 \vee \ldots \vee c_n)^+$ we obtain a structured cost expression $Bd(SE_i, \varphi_{ph^*}, \bar{x}_1)$ where φ_{ph^*} is an auxiliary invariant that relates $\bar{x}_1$ (the variables at the beginning of the phase) to any $\bar{x}_j$ as defined in Sec. 3.2. That structured cost expression is valid for any application of c_i during the phase. This allows us to transform each sum $\sum_{j=1}^{iv_i} SE_i(\bar{x}_j)$ into a product $iv_i * Bd(SE_i, \varphi_{ph^*}, \bar{x}_1)$. Similarly, we maximize the cost expressions in the constraints. A set of valid constraints is $CS_{ph} = \bigcup_{i=1}^{n}(\{\sum iv_i \leq e'_i | \sum iv_i \leq e_i \in CS_i, e'_i \in bd(e_i, \varphi_{ph^*} \wedge \varphi_{c_i}, \bar{x}_1\}) \cup CS_{new}$, where CS_{new} is a new set of constraints that bounds the new iteration variables $(iv_1, iv_2, \cdots, iv_n)$. The maximization of the constraints is equivalent to the maximization of the iteration variables inside SE_i (proof in Appendix A.2).

Bounding the iterations of a phase To generate the constraints in CS_{new}, we use the ranking functions and their dependencies obtained when proving termination (see Sec. 3.3).

Example 12. Consider a phase formed by the following cost equations expressed in compact form (we assume that all have the condition $a, b, c \geq 0$):

$1 : p(a, b, c) = p(a-1, b, c) \,|\, 2 : p(a, b, c) = p(a+2, b-1, c) \,|\, 3 : p(a, b, c) = p(a, c, c-1)$
(3) has a ranking function c with no dependencies. We can add $iv_3 \leq c$ to the constraints. (2) has b as a ranking function but it depends on (3). Every time (3) is executed, b is "restarted". Fortunately, the value assigned to b has a maximum (the initial c). Therefore, we can add the constraint $iv_2 \leq b + c * c$. Finally, (1) has a as a ranking function that depends on (2). a is incremented by 2 in every execution of (2) whose number of iterations is at most $b + c * c$. We add the constraint $iv_1 \leq a + 2 * (b + c * c)$.

More formally, we have a set RF_i for each c_i in a phase. Each $f \in RF_i$ has a (possibly empty) dependency set to other c_j. Given a ranking function f that occurs in all sets $RF_{i_1}, \ldots, RF_{i_m}$ for a maximal m, $i_k \in 1..n$. If f has no dependencies, then $nat(f)$ expressed in terms of $\bar{x}_1$ is an upper bound on the number of iterations of $c_{i_1}, \ldots, c_{i_m}$ and we add $\sum_{k=1}^{m} iv_{i_k} \leq nat(f)$ to CS_{new}.

If f depends on $c_{j_1}, \ldots, c_{j_l}$ ($j_i \in 1..n$) and $ub_{j_1}, \ldots, ub_{j_l}$ are upper bounds on the number of iterations of $c_{j_1}, \ldots, c_{j_l}$, then we distinguish two types of dependencies: (1) if c_{j_i} increases f by a constant t_{j_i} then each execution of c_{j_i} can imply t_{j_i} extra iterations. We add $ub_{j_i} * t_{j_i}$ to f; (2) otherwise, if f can be "restarted" in every execution of c_{j_i}, then $R^f_{j_i} \in bd(f(\bar{x}_3), \varphi_{ph^*}(\bar{x}_1\bar{x}_2) \wedge \varphi_{c_{j_i}}(\bar{x}_2\bar{x}_3), \bar{x}_1)$ represents the maximum value that f can take in c_{j_i} (if it exists) and we add $ub_{j_i} * nat(R^f_{j_i})$. Taken together, we can add $\sum_{k=1}^{m} iv_{i_k} \leq nat(f) + \sum_{i=1}^{p} ub_{j_i} * t_{j_i} + \sum_{i=p}^{l} ub_{j_i} * nat(R^f_{j_i})$ to CS_{new} where $c_{j_1}, c_{j_2} \cdots c_{i_p}$ are the dependencies of type (1) and $c_{i_p}, c_{i_{p+1}} \cdots c_{i_l}$ the ones of type (2).

On top of this, we add constraints that depend on the value of the variables after the phase (see the cost of $(7)^+$ Sec.4.2). This will allow us to perform constraint compression afterwards.

Lemma 1. *Given a sequence of r calls $c_{i_1}(\bar{x}_1) \cdot c_{i_2}(\bar{x}_2) \cdots c_{i_r}(\bar{x}_r) \cdot c'(\bar{x}_{r+1})$, during which c_i occurred p times and $f \in RF_i$, and for all $\langle c_{i_j}(\bar{x}_j) = \cdots + c_{i_{j+1}}(\bar{x}_{j+1}), \varphi \rangle$, $\varphi \Rightarrow (f(\bar{x}_j) - f(\bar{x}_{j+1}) \geq 0)$. We have that $f(\bar{x}_1) - f(\bar{x}_{r+1}) \geq p$.*

If f is a ranking function in $RF_{i_1}, \ldots, RF_{i_m}$ as above, if f has no dependencies, we can use Lemma 1 (proof in Appendix A.2) to add $\sum_{k=1}^{m} iv_{i_k} \leq nat(f(\bar{x}_1) - f(\bar{x}_f))$ to CS_{new} where $\bar{x}_f$ are the variables at the end of the phase.

Inductive constraint compression We generalize the *constraint compression* presented in Sec. 4.3. Instead of compressing two constraints, we compress an arbitrary number of them inductively. This is the mechanism used to obtain a linear bound for the chain $(3.3 \vee 3.4)^+$ at the end of Sec. 4.2.

When a constraint is compressed, its iteration variables should be removed from constraints that cannot be compressed. Removing an iteration variable from a constraint is always safe but can introduce imprecision.

Given a cost expression e_i that we want to compress to $\sum iv \leq e_i$, we start with a copy e'_i of e_1 as our candidate. First, prove the base case $\varphi_i \Rightarrow e'_i \geq e_i$ (which is trivial given that e_i and e'_i are equal). Then prove the induction step $\varphi_{ph}(\bar{x}_1\bar{x}_2) \wedge \varphi_{ph^*}(\bar{x}_2\bar{x}_3) \wedge \varphi_i(\bar{x}_3\bar{x}_4) \Rightarrow e'_i(\bar{x}_1\bar{x}_4) \geq e'_i(\bar{x}_1\bar{x}_2) + e_i(\bar{x}_3\bar{x}_4)$. Assuming e'_i

is valid for a number of iterations (represented as $\varphi_{ph}(\bar{x}_1\bar{x}_2)$), this shows that it is valid for one more iteration ($\varphi_i(\bar{x}_3\bar{x}_4)$) even if there are interleavings with other c_j ($\varphi_{ph^*}(\bar{x}_2\bar{x}_3)$). Once we proved that, we can add the constraint $\sum iv' \leq e'_i$ and pull the corresponding iteration components out of the corresponding product (proof in App A.2).

If we can prove the stronger inequality $e'_i(\bar{x}_1\bar{x}_4) \geq e'_i(\bar{x}_1\bar{x}_2) + e_i(\bar{x}_3\bar{x}_4) + 1$, then we know that e'_i also decreases with the iterations of c_i. In this case we derive a new constraint $\sum iv' + iv_i \leq e'_i$. We can generalize this procedure to compress constraints that originate from different equations. This is demonstrated by the following example.

Example 13. Consider the phase $(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$. Both 3.1 and 4.1 have a call to $move_{(5)^+(6)}$ and their cost structures are $iv_{5.1} * 2 : \{iv_{5.1} \leq n' - n, iv_{5.1} \leq m - n\}$ and $iv_{5.2} * 2 : \{iv_{5.2} \leq n' - n, iv_{5.2} \leq m - n\}$. We can compress both iteration variables obtaining $iv_{5.1} * 2 + iv_{5.2} * 2 : \{iv_{5.1} + iv_{5.2} \leq n' - n\}$ (3.2 and 4.2 have zero cost) that when maximized will give us $iv_{5.1} * 2 + iv_{5.2} * 2 : \{iv_{5.1} + iv_{5.2} \leq m - n\}$ which represents the upper bound $2(m - n)$.

4.5 Cost Structure of a Chain

Given a chain $ch = ph_1 \cdots ph_n$ whose phases have cost structures $CT_1, \ldots CT_n$, we want to obtain a cost structure $CT_{ch} = SE_{ch} : CS_{ch}$ for the total cost of the chain. This is analogous to computing the cost structure of an equation in Sec. 4.3. One constructs a cost constraint φ_{ch} relating all variables of the calls to the entry variables and to each other: $\varphi_{ch} = \varphi_{ph_1}(x_1x_2) \wedge \varphi_{ph_2}(x_2x_3) \wedge \cdots \wedge \varphi_{ph_n}(x_n)$. This cost constraint can be enriched with the invariants of the chain.

The structured cost expression is $SE_{ch} = \sum_{i=1}^{n} Bd(SE_i, \varphi_{ch}, \bar{x})$ and the constraints are $CS_c \supseteq \{\sum iv \leq e' | \sum iv_i \leq e \in CS_i, e' \in bd(e, \varphi_{ch}, \bar{x})\}$. Again, we can apply *constraint compression* to combine constraints from different phases.

Example 14. The cost of patterns (2) and (3) in Ex. 7 derive directly from the cost of their phases (see Sec. 4.2 and Ex. 13). We examine the cost of pattern (1), that is, $(3.3 \vee 3.4)^+(4.3)(3.1 \vee 3.2 \vee 4.1 \vee 4.2)^+$. Considering that variables are subscripted with $1, 2$ and 3 for their value before the first, second and third phase, the cost structures of the phases are: $1 * iv_{7.1} : \{iv_{7.1} \leq n_1 - n_2\}$, $1 * iv_{7.2} : \{iv_{7.2} \leq n_2 - n_3\}$ and $iv_{5.1} * 2 + iv_{5.2} * 2 : \{iv_{5.1} + iv_{5.2} \leq n_4 - n_3\}$. The joint invariants guarantee that $n_3 \geq 0 \wedge n_4 \leq m$. We can compress the constraints $iv_{7.1} \leq n_1 - n_2$ and $iv_{7.2} \leq n_2 - n_3$ and maximize with respect to the initial variables obtaining $1 * iv_{7.1} + 1 * iv_{7.2} + 2 * iv_{5.1} + 2 * iv_{5.2} : \{iv_{7.1} + iv_{7.2} \leq n_1, iv_{5.1} + iv_{5.2} \leq m_1\}$. Such a cost structure represents the bound $n + 2m$ as expected.

5 Solving Cost Structures

Solving a cost structure $SE : CS$ means to look for a maximizing assignment val_{max} from iteration variables to cost expressions (without iteration variables) such that $CS \Rightarrow val_{max}(SE) \geq SE$ is valid. Even though iteration variables range over natural numbers, we consider a relaxation of the problem where iteration variables can take any non-negative real number. The maximization of $val_{max}(SE)$ represents the cost structure SE where each iv has been substituted by $val_{max}(iv)$ and $val_{max}(SE)$ is an upper bound of the cost structure $SE : CS$.

Let $SE = \sum_{i=1}^{n} SE_i * iv_i + e$, The maximization of each SE_i can be performed independently, because its iteration variables depend neither on other iteration variables of SE_j for $j \neq i$ nor on any iv_i. Let e_i be the maximization of SE_i, then we obtain $\sum_{i=1}^{n} e_i * iv_i + e$ as well as a set of constraints over the iv_i. As the e_i's can be symbolic expressions, not necessarily comparable to each other, we need a procedure to find an upper bound independently of the e_i.

We group iteration components (Def. 5) based on dependencies. Two iteration components depend on each other if their iteration variables appear together in a constraint. An iteration group IG is a partial cost structure $\sum_{i=1}^{m} e_{j_i} * iv_{j_i} : CS$ $(1 \leq j_i \leq n$ for $i \in 1..m)$ where its iteration components depend on each other.

A constraint $\sum_{i=1}^{m} iv_{j_i} \leq e$ is *active* for assignment val iff $\sum_{i=1}^{m} val(iv_{j_i}) = e$. Let $C = \sum_{i=1}^{m} iv_{j_i} \leq e$, $C' = \sum_{i=1}^{m+k} iv_{j_i} \leq e'$ be constraints such that $C \subseteq C'$ and val any assignment: (i) If C is active for val, then $C = e$ and we substitute $\sum_{i=m+1}^{m+k} iv_{j_i} \leq nat(e' - e)$ for C' making the two constraints independent; (ii) If C is not active, we ignore C and consider the rest of the constraints.

Consider an IG $SE : CS$ that we want to maximize. For each $C, C' \in CS$ with $C \subseteq C'$, we use the observation in the previous paragraph to derive simplified constraints CS_1, CS_2. We solve both constraints and obtain val_1, val_2. The maximum cost of IG is $\min(val_1(SE), val_2(SE))$. Constraints with only one iv can always be reduced. We repeat the procedure until the constraints cannot be further simplified. The constraints can now be grouped into irreducible IGs. A trivial IG is one with a single iv constraint $iv \leq e$ whose maximal assignment is $val(iv) = e$. All constraints in an irreducible, non-trivial IG have at least two iteration variables.

Example 15. Consider the following cost structure $iv_1 * 1 + iv_2 * (b) + iv_3 * (iv_4 * 2) : \{iv_1 + iv_2 + iv_3 \leq a + b, iv_1 + iv_2 \leq c, iv_4 \leq d\}$. First, we maximize the internal iteration component $iv_4 * 2$ which contains a trivial IG $iv_4 \leq d$. The result is $iv_1 * 1 + iv_2 * (b) + iv_3 * (2d) : \{iv_1 + iv_2 + iv_3 \leq a + b, iv_1 + iv_2 \leq c\}$. This cost structure forms a single IG with two constraints one contained in the other. (1) We assume $iv_1 + iv_2 \leq c$ is active. Then we have $\{iv_3 \leq nat(a + b - c), iv_1 + iv_2 \leq c\}$ which contains two irreducible IG. The first one is $iv_3 = nat(a + b - c)$ and the second one has two possibilities $iv_1 = c, iv_2 = 0$ or $iv_1 = 0, iv_2 = c$ (Thm. 2 below). The result is then $nat(a+b-c) + max(b*c, 2d*c)$. (2) If $iv_1 + iv_2 \leq c$ is not active, we have only $iv_1 + iv_2 + iv_3 \leq a + b$ which yields $max(a+b, b*(a+b), 2d*(a+b))$. The cost is $min(nat(a+b-c) + max(b*c, 2d*c), max(a+b, b*(a+b), 2d*(a+b)))$.

We could have dropped the second constraint from the beginning and obtain a less precise bound $max(a + b, b * (a + b), 2d * (a + b))$. We can even split the constraint $iv_1 + iv_2 + iv_3 \leq a + b$ into $iv_1 \leq a + b$, $iv_2 \leq a + b$ and $iv_3 \leq a + b$ and obtain $(1 + b + 2d) * (a + b)$. That way we can balance precision and performance.

Definition 8 (IG dependency graph). *Let $IG = SE : CS$. Its dependency graph $G(IG)$ is defined as follows: for each $C \in CS$ G has a node C. For each*

$C \cap C'$ such that $C, C' \in CS$ and $C \cap C' \neq \emptyset$ G has a node $d(C \cap C')$, and edges from C to $d(C \cap C')$ and from $d(C \cap C')$ to C'.

Example 16. Given the IG $\{iv_1 + iv_2 \leq a, iv_2 + iv_3 \leq b, iv_2 + iv_4 \leq c\}$, its dependency graph contains the nodes $n_1 = "iv_1 + iv_2 \leq a"$, $n_2 = "iv_2 + iv_3 \leq b"$, $n_3 = "iv_2 + iv_4 \leq c"$ and $n_4 = "d(iv_2)"$. The edges are $(n_1, n_4), (n_2, n_4), (n_3, n_4)$.

Theorem 2. *Given an irreducible, non-trivial IG. If $G(IG)$ is acyclic there exists a maximizing assignment val_{max} such that there is an active constraint with only one non-zero iteration variable.*

If $G(IG)$ is acyclic, we apply Thm. 2 to solve IG incrementally. Let $C_i = \sum_{j=1}^{r} iv_{i_j} \leq e \in CS$: we obtain a partial assignment val_{ik} such that $val_{ik}(iv_k) = e$ for some $iv_k \in C_i$ and all other iteration variables in C_i being assigned 0. We update CS with val_{ik} and obtain a constraint system with less iteration variables and constraints whose graph is still acyclic, and so on. Once no iteration variable is left, we end up with a set of assignments $MaxVal$. The maximum cost of $IG = SE : CS$ is $max_{val \in MaxVal} val(SE)$.

Example 17. We obtain one of the assignments in $MaxVal$ for the IG of Ex. 16. We take the constraint $iv_1 + iv_2 \leq a$ and assign $iv_1 = a$ and $iv_2 = 0$. The resulting constraints are $iv_3 \leq b$ and $iv_4 \leq c$ that are trivially solved. The resulting assignment is $iv_1 = a, iv_2 = 0, iv_3 = b$ and $iv_4 = c$.

The requirement of $G(IG)$ being acyclic can be relaxed. A discussion and the proof of Thm. 2 is in App A.3. One can always obtain an acyclic IG by dropping constraints or by removing iteration variables from a given constraint. Such transformations are safe since they only relax the conditions imposed on the iteration variables. In practice, we perform a pre-selection of the constraints to be considered based on heuristics to improve performance.

6 Related Work and Experiments

This work builds upon the formalism developed in the COSTA group [1, 2, 4, 5], however, the are important differences in how upper bounds are inferred. In [1], upper bounds are computed independently for each SCC and then combined without taking dependencies into account. The precision of that approach is improved in [2] for certain kinds of loops. The paper [5] presents a general approach for obtaining amortized upper bounds that, although powerful, does not scale well. In [4] SCCs are decomposed into sparse cost equations systems. Then it is possible to use the ideas of [5] to solve the sparse cost equations precisely.

In our work, we also decompose programs, but driven by possible sequences of cost applications. This technique, known as control-flow-refinement, has been applied to the resource analysis of imperative programs in [12, 9]. In addition, our refinement technique can deal with programs with linear recursion (non neccessarily tail recursive) and multiple procedures. In our analysis we do not refine the whole program at once. Instead, we refine each SCC and then propagate the changes. Our technique allows to leave parts of the program unrefined to increase performance. Paper [14] uses disjunctive invariants to summarize inner loops instead of control-flow-refinement. This technique can also deal with some kinds of non-terminating programs. However, it can only bound the number of visits to a single location in a single procedure. In contrast, our tool can count the number of visits to several locations in multiple procedures derived from cost annotations. The tool Loopus [17] uses disjunctive invariants, collects the inner paths of each loop and also uses contextualization which is a form of control-flow refinement. Both [14, 17] obtain ranking functions based on given patterns and combine them using proof rules. Instead, we infer linear ranking functions using linear programming [15, 6] and combine them to form lexicographic ranking functions (see Sec. 4.4).

SPEED [13] makes use of multiple counters to bound and detect dependencies of different loops. SPEED computes cost summaries for the (non-recursive) procedure calls. Therefore, it cannot detect dependencies among different procedure calls. KoAT [8] adopts an iterative approach, where size analysis and complexity analysis are interleaved and improve each other. That paper also extends transitions systems to deal with inter-procedural and recursive programs. Very recently, a new version of Loopus has been released [16]. They use a simple abstraction and achieve very high performance and great effectiveness. They can also obtain amortized cost for complex nested loops. However, their analysis is limited to imperative programs and cannot deal with recursion.

For our experimental evaluation we took the problem set used by KoAT's evaluation[2] [8], except those with multiple recursion (670 problems). We executed each problem with PUBS [1], KoAT, and our tool CoFloCo (SPEED and the first version of Loopus [17] are not publicly available). The problems are taken from the literature on resource analysis [3, 12–14, 17] and include most of the problems used in the evaluation of [7] (631 problems in the first part of the table) and the ones of the evaluation of PUBS [1] (39 problems in the second part).

	1	$log\ n$	n	$n\ log\ n$	n^2	n^3	$> n^3$	No res.
CoFloCo	115	0	141	0	52	2	3	318
KoAT	117	0	120	0	51	0	4	339
PUBS	90	2	85	5	37	3	3	406
Loopus[3]	128	0	140	0	73	11	4	275
CoFloCo	1	0	16	0	14	7	0	1
PUBS	1	2	13	3	12	6	0	2
Loopus[3]	2	0	11	0	7	4	0	15

The problems of the first part were automatically translated from KoAT's input format to cost equations. That includes performing loop extraction (and generating invariants for PUBS). No slicing took place so the input cost equations might have many more variables than needed. For the second set we used the original cost equations for PUBS and CoFloCo. We decided not to include these problems for KoAT as the translation generated in [8] is not sound (we found several problems where KoAT yields an incorrect upper bound). We summarize the number of problems solved by each tool in different complexity categories.

[2] `http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity/`

Each problem was run with a time-out of 60 secs. The same set of problems[3] has been used to evaluate the new version of Loopus [16]. We include the results of their evaluation[4] in a shaded row to emphasize that we did not run the experiments ourselves.

CoFloCo obtains a bound asymptotically better than KoAt in 60 problems and better than PUBS in 109 problems. Conversely, KoAt obtains a better bound than CoFloCo in 23 problems and PUBS is better than CoFloCo in 11 problems. CoFloCo obtains better results than Loopus in 48 of the problems analyzed by both. Loopus obtains better results than CoFloCo in 93 problems. However, in 51 of these problems, Loopus reports an upper bound as a function of `call_to_nondet_line_X` where `X` is a line number. It seems that Loopus assumes a specific symbolic value whenever a non-deterministic assignment is executed whereas CoFloCo does not make such an assumption and fails to provide a bound. The complete experimental data and the implementation are available.[5]

At this time, CoFloCo is just prototype and can be greatly improved. It fails on 27 problems because of irreducible loops. Irreducible loops can be transformed and the approach could be extended to handle other domains including non-linear constraints, logarithmic bounds, etc. The invariants could also be improved with the termination information of Sec.3.3 following the ideas of [8]. CoFloCo had 94 time-outs. Most occurred with problems with many variables where slicing could be applied. In some occasions, the control-flow-refinement of cost equations can generate exponentially many chains. However, these chains have many fragments in common and part of the invariant and upper bound computation can be reused. Moreover, some SCCs can be left unrefined to achieve a compromise between performance and precision.

We presented a control-flow-refinement algorithm that can be applied to linear recursive programs (other approaches do not support recursion). The algorithm distinguishes terminating and non-terminating executions explicitly which allows obtaining better invariants for the terminating executions. This also allows to have intermediate cost expressions depending on the output variables (see the cost of $(7)^{+}(8)$) and thus obtain amortized cost bounds. We obtain an upper bound for each execution pattern (chain), which often provides more precise information than a generic upper bound for any possible execution. The upper bounds are also precise because cost structures allow us to maintain several upper bound candidates, detect dependencies among different parts of the code (using *constraint compression*) and obtain complex upper bound expressions.

Acknowledgements Research partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services. We thank the anonymous reviewers for their careful reading which resulted in numerous improvements. We thank S. Genaim for valuable discussions and help with the experiments.

[3] 18 problems included here were left out of the evaluation of Loopus.

[4] `http://forsyte.at/static/people/sinn/loopus/CAV14/`

[5] `www.se.tu-darmstadt.de/se/group-members/antonio-flores-montoya/cofloco`

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. of Automated Reasoning*, 46(2):161–203, Feb. 2011.
2. E. Albert, S. Genaim, and A. N. Masud. More precise yet widely applicable cost analysis. In *VMCAI, Austin, TX*, volume 6538 of *LNCS*, pages 38–53. Springer, 2011.
3. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.
4. D. E. Alonso-Blas, P. Arenas, and S. Genaim. Precise cost analysis via local reasoning. In *ATVA*, LNCS. Springer, oct 2013.
5. D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *SAS*, volume 7460 of *LNCS*, pages 405–421. Springer, Sept. 2012.
6. R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215(0):47 – 67, 2012.
7. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 413–429. Springer, 2013.
8. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, 2014.
9. H. Chen, S. Mukhopadhyay, and Z. Lu. Control flow refinement and symbolic computation of average case bound. In *Automated Technology for Verification and Analysis*, volume 8172 of *LNCS*, pages 334–348. Springer, 2013.
10. B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 47–61. Springer, 2013.
11. B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Computer Aided Verification*, volume 5123 of *LNCS*, pages 370–384. Springer, 2008.
12. S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.
13. S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, New York, NY, USA, 2009. ACM.
14. S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI'10*, pages 292–304, New York, NY, USA, 2010. ACM.
15. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, volume 2937 of *LNCS*, pages 239–251, 2004.
16. M. Sinn, F. Zuleger, and H. Veith. A simple and scalable approach to bound analysis and amortized complexity analysis. In *CAV*, volume 8559 of *LNCS*, pages 743–759. Springer, 2014.
17. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In E. Yahav, editor, *Static Analysis*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

A Proofs

A.1 Proof of Refinement Completeness

Proof (of Thm. 1).

Let $ca_1 \cdot ca_2 \cdots$ be a sequence of cost equation applications, and consider the basic definition of dependency (without taking invariants into account). By definition we have that $ca_i \preceq ca_{i+1}$ for all i.

For the proof we use two consequences of the definition of phases. The phases $ph_1, \ldots, ph_n$ resulting from a refinement determine a partition of the set of cost equations. Whenever ca_i, $ca_j \in ph_r$ ($r \in 1..n$) such that $i < j$, all ca_k $i \leq k \leq j$ belong to ph_r as well. This is because of $ca_i \preceq^* ca_{i+1} \cdots ca_{j-1} \preceq^* ca_j$ and, by definition of a phase, $ca_i = ca_j$ or $ca_j \preceq^* ca_i$. In either case we conclude that $ca_{i+1}, ca_{i+2}, \cdots ca_{j-1} \in ph$.

Let ph be the phase where ca_1 occurs, we distinguish two cases: (i) for all $ca_i \in ph$ there is at least one ca_j with $j > i$ such that $ca_j \in ph$. This implies that ph is a valid chain that matches the sequence of cost equation applications. (ii) There exists a $ca_i \in ph$ such that for all ca_j with $j > i$ it is the case that $ca_j \notin ph$. If ca_i is the final equation application in the sequence, ph is a valid chain that matches the complete sequence. If ca_i has a successor ca_{i+1} we know that $ca_i \preceq ca_{i+1}$, but $ca_{i+1} \in ph' \neq ph$. By definition, $ph \prec ph'$, so our sequence matches a chain $ph{\cdot}ph' \cdots$. Now we can apply the same reasoning to the remaining subsequence $ca_{i+1} \rightarrow ca_{i+2} \rightarrow \cdots$ and phases $\{ph_1, \ldots, ph_n\} - \{ph\}$. This is guaranteed to terminate, because the set of phases is finite.

If we want to refine a specific chain ch, we can take its invariants into account. The reasoning is the same but it is only applicable as long as the invariants are correct, that is, for any sequence of cost equations that matches ch.

Any sequence of cost equation applications s of S matches a basic chain ch of S. We have proved that if s matches ch, s matches one of its refined chains $ch_1, \ldots, ch_n$. Therefore, any sequence of cost equation applications s of S matches one of the refined chains of the basic chains of S. □

A.2 Cost Structure of a Phase

Given a sequence of cost equation applications $ca_1(\bar{x}_1) \cdot ca_2(\bar{x}_2) \cdots ca_m(\bar{x}_m)$ of phase ph, where $m \in \mathbb{N} \cup \{\infty\}$. Let for all $i \in 1..m$: $ca_i \in \{c_1, \ldots, c_n\}$ and $\bar{x}_i$ are the entry variables at the i-th cost equation application.

Let $CT_i = SE_i : CS_i$ be the cost of one application of c_i, then the total cost of the phase is $\sum_{i=1}^{n}(\sum_{j=1}^{iv_i} SE_{ij})$ where SE_{ij} is an instance of SE_i with the variables corresponding to the j-th application of c_i.

Claim. The expression $SE_{ph} : CS_{ph}$, where $SE_{ph} = \sum_{i=1}^{n} iv_i{*}Bd(SE_i, \varphi_{ph^*}, \bar{x}_1)$, is a valid cost structure for phase ph, where iv_i is a new iteration variable that represents the number of applications of equation c_i and the constraints are given as $CS_{ph} = \bigcup_{i=1}^{n}(\{\sum_{k=1}^{l} iv_{ik} \leq e'_i | \sum_{k=1}^{l} iv_{ik} \leq e_i \in CS_i, e'_i \in bd(e_i, \varphi_{ph^*}, \bar{x}_1\}) \cup CS_{new}$.

Proof. Consider $\sum_{j=1}^{iv} SE(\bar{x}_j)$. Structured cost expressions are defined recursively ($SE = \sum_{i=1}^{n} SE_i * iv_i + e$), so we prove correctness by induction:

Base case: $\sum_{j=1}^{iv} SE(\bar{x}_j) = \sum_{j=1}^{iv} e(\bar{x}_j)$. By definition of Bd and bd

$$Bd(SE(\bar{x}_j), \varphi_{ph^*}(\bar{x}_1\bar{x}_j), \bar{x}_1)) = min(bd(e(\bar{x}_j), \varphi_{ph^*}(\bar{x}_1, \bar{x}_j), \bar{x}_1)) \geq e(\bar{x}_j)$$

yields the same result in terms of $\bar{x}_1$ for every j. Therefore, $\sum_{j=1}^{iv} e(\bar{x}_j) \leq \sum_{j=1}^{iv} Bd(SE, \varphi_{ph^*}, \bar{x}_1)$. We pull out the common factor $Bd(SE, \varphi_{ph^*}, \bar{x}_1) * \sum_{j=1}^{iv} 1 = Bd(SE_i, \varphi_{ph^*}, \bar{x}_1) * iv$.

Inductive case: $\sum_{j=1}^{iv} SE(\bar{x}_j) = \sum_{j=1}^{iv} \sum_{k=1}^{r} SE_k(\bar{x}_j) * iv_{kj} + e(\bar{x}_j)$. For each k we have $\sum_{j=1}^{iv} SE_k(\bar{x}_j) * iv_{kj} \leq \sum_{j=1}^{iv} Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * iv_{kj}$ by the induction hypothesis. We pull out the factor $Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * \sum_{j=1}^{iv} iv_{kj}$. Now, for each constraint $\sum_{l=1}^{p} iv_{lj} \leq e(\bar{x}_j)$, we have $\sum_{l=1}^{p} iv_{lj} \leq e(\bar{x}_j) \leq e'$ where $e' \in bd(e(\bar{x}_j), \varphi_{ph^*}, \bar{x}_1)$. If we add all the constraints of the different j we obtain $\sum_{j=1}^{iv} \sum_{l=1}^{p} iv_{lj} \leq \sum_{j=1}^{iv} e'$ which is $\sum_{l=1}^{p} \sum_{j=1}^{iv} iv_{lj} \leq iv * e'$. Finally, we perform a variable substitution iv'_l for each $\sum_{j=1}^{iv} iv_{lj}$ such that $\sum_{j=1}^{iv} iv_{lj} = iv * iv'_l$. The remaining constraints $\sum_{l=1}^{p} iv * iv'_l \leq iv * e'$ can be simplified to $\sum_{l=1}^{p} iv'_l \leq e'$ as claimed (with each iteration variable renamed to its primed version). The structured cost expression $Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * \sum_{j=1}^{iv} iv_{kj}$ for each k becomes $Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * iv * iv'_k$. Finally, $\sum_{j=1}^{iv} \sum_{k=1}^{r} SE_k(\bar{x}_j) * iv_{kj} + e_k \leq \sum_{k=1}^{r} Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * iv'_k * iv + bd(e_k, \varphi_{ph^*}, \bar{x}_1) * iv = iv * Bd(SE, \varphi_{ph^*}, \bar{x}_1)$.

□

Inductive constraint compression. To prove correctness of inductive constraint compression, start as in the inductive case above:
$\sum_{j=1}^{iv} SE(\bar{x}_j) = \sum_{j=1}^{iv} \sum_{k=1}^{r} SE_k(\bar{x}_j) * iv_{kj} + e(\bar{x}_j)$. For each k, $\sum_{j=1}^{iv} SE_k(\bar{x}_j) * iv_{kj} \leq \sum_{j=1}^{iv} Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * iv_{kj}$ by induction hypothesis. We pull out the factor $Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * \sum_{j=1}^{iv} iv_{kj}$. Now, for each constraint $\sum_{l=1}^{p} iv_{lj} \leq e(\bar{x}_j)$ that we want to compress, we add all its instances $\sum_{l=1}^{p} \sum_{j=1}^{iv} iv_{lj} \leq \sum_{j=1}^{iv} e(\bar{x}_j)$. If we find $e' \geq \sum_{j=1}^{iv} e(\bar{x}_j)$ we have $\sum_{l=1}^{p} \sum_{j=1}^{iv} iv_{lj} \leq e'$. We perform a variable substitution such that $iv'_l = \sum_{j=1}^{iv} iv_{lj}$. The new constraint is $\sum_{l=1}^{p} iv'_l \leq e'$ and the structured cost expression $Bd(SE_k, \varphi_{ph^*}, \bar{x}_1) * iv'_k$. Therefore, for each iteration component k whose iv_{kj} are compressed into iv'_k, the maximized structured cost expression is not multiplied by iv.

Bounding the iterations of a phase.

Proof (of Lemma 1). By induction:

Base case: let $m = 1$, the sequence is $c_{i_1}(\bar{x}_1)\dot{c}'(\bar{x}_2)$ and we have two cases: if f is a ranking funcion of c_{i_1} ($f \in RF_{i_1}$), then $p = 1$ and $f(\bar{x}_1) - f(\bar{x}_2) \geq 1$ by the definition of ranking function; if $f \notin RF_{i_1}$, then $p = 0$ and $f(\bar{x}_1) - f(\bar{x}_2) \geq 0$ for all the applications of the equations considered (it is a condition of the lemma).

Inductive case: the induction hypothesis is that $f(\bar{x}_1) - f(\bar{x}_m) \geq p$ where p is the number of applications of the cost equations c_i such that $f \in RF_i$. Here we have also two cases: if f is a ranking funcion of c_{i_m} ($f \in RF_{i_m}$), then the number of applications is $p + 1$ and $f(\bar{x}_m) - f(\bar{x}_{m+1}) \geq 1$. By adding the induction hypothesis, we obtain $f(\bar{x}_1) - f(\bar{x}_m) + f(\bar{x}_m) - f(\bar{x}_{m+1}) \geq p + 1$ which simplifies to $f(\bar{x}_1) - f(\bar{x}_{m+1}) \geq p + 1$. Otherwise, if $f \notin RF_{i_m}$, the number of applications is p and $f(\bar{x}_m) - f(\bar{x}_{m+1}) \geq 0$. By adding the induction hypothesis, we obtain $f(\bar{x}_1) - f(\bar{x}_m) + f(\bar{x}_m) - f(\bar{x}_{m+1}) \geq p$, hence $f(\bar{x}_1) - f(\bar{x}_{m+1}) \geq p$. □

A.3 Solving cost structures

We prove Thm. 2 in two steps. We define a general property of a set of constraints, called *well-behaved constraints*, under which the property of interest, i.e "there exists a maximizing assignment val_{max} such that there is an active constraint with only one non-zero iteration variable", holds. Then we prove that the constraints of an irreducible, non-trivial IG whose graph is acyclic, are well-behaved.

Definition 9 (Assignment $val[iv_i+]$). *Let val be an assignment to iteration variables such that $val(iv_i) > 0$. Then $val[iv_i+]$ must have the following form:*

- *if $val(iv) = 0$, then $val[iv_i+](iv) = 0$.*
- *$val[iv_i+](iv_i) = val(iv_i) + \epsilon_i$ for some* positive *$\epsilon_i \in \mathbb{R}^+$.*
- *$val[iv_i+](iv_j) = val(iv_j) + \epsilon_j$ for some $\epsilon_j \in \mathbb{R}$ for $j \neq i$*

Each $val[iv_i+]$ uniquely determines an assignment $val[iv_i-]$ with the same ϵ for each variable:

- *if $val(iv) = 0$, $val[iv_i-](iv) = 0$.*
- *$val[iv_i-](iv_i) = val(iv_i) - \epsilon_i$ where $\epsilon_i \in \mathbb{R}^+$.*
- *$val[iv_i-](iv_j) = val(iv_j) - \epsilon_j$ where $\epsilon_j \in \mathbb{R}$ for $j \neq i$*

Definition 10 (Well-behaved constraint). *A set of constraints CS is* well-behaved *iff for any given assignment val and every active constraint C that contains at least two non-zero variables, there exist an assignment $val[iv_i+]$ for a non-zero variable iv_i such that the active constraints under val are also active under $val[iv_i+]$ (and, of course, $val[iv_i-]$).*

Example 18. An example of an IG that is not well-behaved is $3 * iv_1 + 2 * iv_2 + 2 * iv_3 : \{iv_1 + iv_2 \leq 10, iv_2 + iv_3 \leq 10, iv_1 + iv_3 \leq 10\}$. For an assignment $val = iv_1 \to 5, iv_2 \to 5, iv_3 \to 5$, we have that all constraints are active but if we increase any iteration variable, not all constraints can remain active. In fact, val is a maximizing assignment for this IG.

Theorem 3. *Given a well-behaved set of constraints CS, there exists a maximizing assignment val_{max} such that there is one active constraint with only one non-zero iv.*

Proof. Assume the contrary. All maximizing valuations cause every active constraint to have at least two non-zero iteration variables. Take such a maximizing valuation val (there must be at least one) with n active constraints and m zero iteration variables.

By definition of well-behaved, there is an assignment $val[iv_i+]$ such that the active constraints remain active. The ϵ in $val[iv_i+]$ can be as small as needed, in particular, we can choose them small enough so as not to violate any non-active constraint render iteration variables negative. We have that

$$val[iv_i+](\sum_{k=1}^{r} e_k * iv_k) = val(\sum_{k=1}^{r} e_k * iv_k) + \sum_{k=1}^{r} e_k * \epsilon_k \ .$$

Therefore, there are three possibilities:

1. If $\sum_{k=1}^{r} e_k * \epsilon_k > 0$, then $val(\sum_{k=1}^{r} e_k * iv_k) < val[iv_i+](\sum_{k=1}^{r} e_k * iv_k)$. This would contradict our assumption that val is maximizing.
2. If $\sum_{k=1}^{r} e_k * \epsilon_k < 0$, then $val(\sum_{k=1}^{r} e_k * iv_k) < val[iv_i-](\sum_{k=1}^{r} e_k * iv_k)$. Again, this contradicts our assumption that val is maximizing.
3. If $\sum_{k=1}^{r} e_k * \epsilon_k = 0$, then $val(\sum_{k=1}^{r} e_k * iv_k) = val[iv_i] + (\sum_{k=1}^{r} e_k * iv_k)$. We focus on this case. Now $val[iv_i+]$ is another maximizing valuation. Iterating this process gives $val[iv_i+][iv_i+], val[iv_i+][iv_i+][iv_i+], \ldots$ until one of four things happens:
   - A constraint that was inactive becomes active and it has at least two non-zero iteration variables. We repeat the process with $n+1$ active constraints and m zero iteration variables.
   - A constraint that was inactive becomes active and it has only one non-zero iteration variable. This valuation is maximizing.
   - A non-zero iv whose ϵ is negative becomes 0 but all constraints still have at least two non-zero iteration variables. We repeat the process (select a new $val[iv_i+]$) with n active constraints and $m+1$ zero iteration variables.
   - A non-zero iv whose ϵ is negative becomes 0 and there is one constraint with only one non-zero iteration variable. This valuation is maximizing.

   The second and fourth case conclude the proof. For the first and third case, we repeat the process (the condition for well-behaved is still true) with strictly more zero iteration variables or more active constraints. As the number of both is finite, the process is bound to terminate.

□

The next lemma that completes the proof of Thm. 2. We need one more definition: an iteration variable iv is *independent* with respect to a set of constraints CS iff it appears in only one constraint. Otherwise, it is *dependent.*

Lemma 2. *Let $IC = SE : CS$ If the dependency graph $G(IC)$ is acyclic and all $C \in CS$ have at least two iteration variables, then CS is well-behaved.*

Proof. For a given assignment val such that every active constraint contains at least two non-zero iteration variables, we need to construct an assignment $val[iv_i+]$ for a non-zero iv_i such that the active constraints under val are also active under $val[iv_i+]$.

To build $val[iv_i+]$ we need to make sure that for each constraint, the increase of an iteration variable is compensated by the decrease of another iteration variable. We adopt a simple approach where each non-zero iteration variable is either increased by ϵ, decreased by ϵ or left the same. Let Inc be the set of non-zero iteration variables being increased, Dec the ones that are decreased; $val[iv_i+]$ is defined in terms of these sets:

$$\begin{aligned}
&val[iv_i+](iv) = val(iv) + \epsilon \text{ if } iv \in Inc \\
&val[iv_i+](iv) = val(iv) - \epsilon \text{ if } iv \in Dec \\
&val[iv_i+](iv) = val(iv) \text{ if } iv \notin Inc \cup Dec \\
&0 < \epsilon \leq min_{iv \in Inc \cup Dec} val(iv)
\end{aligned}$$

An assignment $val[iv_i+]$ is *valid* if $iv_i \in Inc$ and for every constraint C, $|C \cap Inc| = |C \cap Dec|$. Moreover, we add the restriction $|C \cap Inc| = |C \cap Dec| \leq 1$. That is, in a constraint C, at most one iv is increased and one decreased.

Given an assignment val and the set of constraints CS', which is obtained from CS by removing all iteration variables iv such that $val(iv) = 0$. We pick a suitable constraint and prove by induction over the structure of $G(IC)$ (built over CS') that for any adjacent constraint one can modify the sets Inc and Dec in such a way that the property still holds. The induction claim says that for any $Cs \subseteq CS'$ with k constraints there exists a valid assignment $val[iv_k+]$.

Base case: There is at least one constraint $C = \sum_{j=1}^{n} iv_j = e$ in CS' with $n \geq 2$. We set $Inc = \{iv_1\}$ and $Dec = \{iv_j\}$ such that $1 < j \leq n$. Now $val[iv_1+]$ is valid because $iv_1 \in Inc$ and $|C \cap Inc| = |C \cap Dec| \leq 1$.

Inductive case: Let $Cs \subseteq CS'$ with k constraints. The induction hypothesis gives an assignment $val[iv_k+]$ and sets Inc, Dec such that for each $C \in Cs$ $|C \cap Inc| = |C \cap Dec| \leq 1$. Now we add a new adjacent constraint C'. If there are no adjacent constraints to Cs we are done and $val[iv_k+]$ is a valid assignment. Denote with $itDeps(C')$ the set of dependent iteration variables with respect to $\{C'\} \cup Cs$. The definition of a dependency graph ensures that $itDeps(C')$ has at least one iteration variable. There are two cases:

1. $itDeps(C')$ has more than one iteration variable. We claim: for every $C \in Cs$ either $itDeps(C') \subseteq C$ or $itDeps(C') \cap C = \emptyset$. Using this claim, we take iv, $iv' \in itDeps(C')$ and set $Inc' = \{iv\}$ and $Dec' = \{iv'\}$. Now $val[iv+]$ constructed with Inc' and Dec' is valid, because $iv \in Inc'$ and for each $C \in Cs$, either C contains iv and iv' ($|C \cap Inc'| = |C \cap Dec'| = 1$) or neither ($|C \cap Inc'| = |C \cap Dec'| = 0$).
   To prove the claim, assume the contrary: there exists a $C_1 \in Cs$ such that $itDeps(C') \cap C_1 = J \neq \emptyset$ and $itDeps(C') - C_1 = D \neq \emptyset$. There

exists a $C_2 \in Cs$ with $D \cap C_2 \neq \emptyset$, because the iteration variables in D are dependent. By definition, there are different nodes $d(J)$ and $d(C' \cap C_2)$ as well as paths $C' - d(J) - C_1$ and $C' - d(C' \cap C_2) - C_2$. Hence, the graph has a cycle as C_1 and C_2 are connected in Cs by construction.

2. If $\mathit{itDeps}(C')$ has only one iteration variable iv, we know there is a $iv' \in C'$ that is independent with respect to $\{C'\} \cup Cs$ (there are at least two non-zero iteration variables).
   - If $iv \in Inc$, $Inc' = Inc$ and $Dec' = Dec \cup \{iv'\}$;
   - if $iv \in Dec$, $Inc' = Inc \cup \{iv'\}$ and $Dec' = Dec$;
   - otherwise, $Inc' = Inc$ and $Dec' = Dec$.

   In all cases Inc and Dec are valid for all $C \in Cs$ since iv' does not appear in any $C \in Cs$ and they are also valid for C'. In addition, $iv_k \in Inc$ and therefore $val[iv_k+]$ constructed with Inc' and Dec' is valid for $Cs \cup \{Cs'\}$.

□

B Invariant Computation

The invariants are generated using the polyhedra abstract domain. The operations $\pi(\varphi, \bar{x})$ and $\varphi_1 \sqcup \varphi_2$ correspond to the projection of φ over the variables $\bar{x}$ and the convex hull (least upper bound) of φ_1 and φ_2 respectively.

The forward invariant at the beginning of a chain $ch = ph_1 \cdots ph_n$ in an SCC S_i is given by the conditions under which the chain is called in other SCCs.

$$\mathit{fwdInv}(|ph_1) = \sqcup\{\pi(\varphi, \bar{z}) | \langle C'(\bar{x}) = \ldots + C_{ch}(\bar{z}), \varphi\rangle \in S_j, j < i\}$$

It represents the join of all possible calling contexts of the chain ch. The backward invariant at the end of the chain is defined as

$$\mathit{backInv}(ph_n|) = \sqcup\{\pi(\varphi, \bar{x}) | \langle C(\bar{x}) = \ldots, \varphi\rangle \in ph_n\}$$

Once these two cases are defined, we can infer the invariants at intermediate points by propagating them through phases. First, we define $\tau^1_{ph_i}(a(\bar{x}))$ as a function that abstractly executes one iteration of ph_i over an abstract state $a(\bar{x})$, i.e., a state is a predicate over the variables $\bar{x}$. Conversely, $\tau^{-1}_{ph_i}(a(\bar{x}))$ represents a backwards step in the execution.

$$\tau^1_{ph_i}(a(\bar{x})) = \sqcup\{\pi(a(\bar{x}) \wedge \varphi, \bar{z}) | \langle C(\bar{x}) = \ldots + C(\bar{z}), \varphi\rangle \in ph_i\}$$
$$\tau^{-1}_{ph_i}(a(\bar{z})) = \sqcup\{\pi(a(\bar{z}) \wedge \varphi, \bar{x}) | \langle C(\bar{x}) = \ldots + C(\bar{z}), \varphi\rangle \in ph_i\}$$

Additionally, $\tau^i_{ph_i}(a(\bar{x}))$ means to apply $\tau^1_{ph_i}(a(\bar{x}))$ i times and $\tau^{-i}_{ph_i}(a(\bar{x}))$ to apply $\tau^{-1}_{ph_i}(a(\bar{x}))$ i times. Now define $\mathit{fwdInv}(|ph_i) = \tau^1_{ph_i}(\mathit{fwdInv}(\overline{ph_{i-1}})(\bar{x}))$ and $\mathit{backInv}(ph_i|) = \tau^{-1}_{ph_i}(\mathit{backInv}(\overline{ph_{i+1}})(\bar{x}))$. Each phase is guaranteed to iterate at least once.

The definitions above take a single iteration through a phase into account. For computing invariants valid for all iterations of a phase, we distinguish whether the phase iterates once or several times. For the latter, we do not assume any specific number of iterations.

$$\begin{array}{ll} \mathit{fwdInv}(\overline{ph_i}) = \mathit{fwdInv}(|ph_i)(\bar{x}) & \text{when } ph_i = (i_1 \vee \ldots \vee i_m) \\ \mathit{fwdInv}(\overline{ph_i}) = \sqcup\{\tau^i_{ph_i}(\mathit{fwdInv}(|ph_i)(\bar{x}))|i \in 0..\infty\} & \text{when } ph_i = (i_1 \vee \ldots \vee i_m)^+ \end{array}$$

The case of the backward invariant is symmetric but using $\tau^{-1}_{ph_i}(a(\bar{x}))$.

Progressive refinement. When computing invariants, we can use additional information from the invariants that have been previously computed. Forward invariants can be taken into account when computing backward invariants and vice-versa. We can adapt the definition of τ for this purpose:

$$\tau_{ph_i}(a(\bar{x})) = \sqcup\{\pi(a(\bar{x}) \wedge \varphi \wedge b(\bar{x}), \bar{z})|\langle C(\bar{x}) = \ldots + C(\bar{z}), \varphi\rangle \in ph_i\} \quad ,$$

where $b(\bar{x}) = \mathit{backInv}(\overline{ph_i})$. We can adapt the definition of $\tau^{-1}_{ph_i}$ with the forward invariant $\mathit{fwdInv}(\overline{ph_i})$ in the same way.