On the Limits of the Classical Approach to Cost Analysis

Diego Alonso and Samir Genaim

DSIC, Complutense University of Madrid (UCM), Spain

Abstract. The classical approach to static cost analysis is based on transforming a given program into cost relations and solving them into closed-form upper-bounds. It is known that for some programs, this approach infers upper-bounds that are asymptotically less precise than the actual cost. As yet, it was assumed that this imprecision is due to the way cost relations are solved into upper-bounds. In this paper: (1) we show that this assumption is partially true, and identify the reason due to which cost relations cannot precisely model the cost of such programs; and (2) to overcome this imprecision, we develop a new approach to cost analysis, based on SMT and quantifier elimination. Interestingly, we find a strong relation between our approach and amortised cost analysis.

1 Introduction

Cost analysis (a.k.a. resource usage analysis) aims at *statically* determining the amount of resources required to safely execute a given program, i.e., without running out of resources. By *resource*, we mean any quantitative aspect of the program, such as memory consumption, execution steps, etc. Several cost analysis frameworks are available [2,10,12,14,15,17]. Although different in their underlying theory, all of them usually report the cost of a program as an upper-bound function (UBF for short) such that: when evaluated on (an abstraction of) a given input, the UBF gives an upper-bound on the amount of resources required for safely running the program on that specific input.

Many automatic cost analysis tools are based on the *classical approach* of Wegbreit [22], which we describe using its extension for JAVA bytecode [2]. This analysis is done in three steps: (1) the JAVA program is transformed into an *abstract program*, in which data-structures are abstracted to their sizes, e.g., length of lists, depth of trees, etc.; (2) the abstract program is transformed into a set of *cost relations* (CRs for short), which are a non-deterministic form of *recurrence equations* that define the cost of executing the program in terms of its *input* parameters; and (3) the CRs are solved into UBFs.

This analysis performs well in practice, however, for some classical examples, it infers UBFs that are asymptotically less precise than the actual cost. Clearly, the abstraction at step (1) may involve a loss of precision since it can introduce spurious traces, which do not occur in the original program. This imprecision is out of the scope of this paper. Instead, we focus on the imprecision at steps (2)

and (3). As yet, it was *assumed* that this imprecision is due to the way CRs are solved into UBFs in step (3), and that, in principle, it could be overcome using more precise resolution techniques.

The *first contribution* of this paper shows that this assumption is not true, namely, that the cost of some programs cannot be modeled precisely with CRs. This is because CRs are defined only in terms of the input parameters, and thus they fail to capture dependencies between the output of a program and its cost. These dependencies are crucial for programs in which the output of one part is passed as input to another part, and transforming them into CRs introduces spurious scenarios. Any resolution technique that solves CRs into UBFs must cover these spurious scenarios, hence it would fail to obtain precise UBFs.

To eliminate these spurious scenarios, an UBF must be defined in terms of both input and output. Our *second contribution* is a novel cost analysis that uses the this notion of cost. It is based on quantifier elimination and template UBFs. Briefly, it takes a given set of template UBFs, with some unknown parameters, and uses satisfiability modulo theory (SMT) and quantifier elimination to instantiate those parameters, such that the resulting UBFs are safe.

The rest of the paper is organised as follows. Sec. 2 presents our running examples and formally defines the language on which we apply our analysis. Sec. 3 studies the limitations of CRs. Secs. 4 and 5 are the technical core of the paper, in which we develop our cost analysis. Sec. 6 discusses the relation of our analysis to amortised cost analysis. Sec. 7 describes a prototype implementation. Sec. 8 overviews related work, and Finally, Sec. 9 concludes.

2 Motivating examples and preliminaries

In this section we describe an *abstract cost rules* (ACR for short) language [2], which we use to formally present our cost analysis. In [2], a JAVA program is *automatically abstracted* to this language. The abstraction guarantees that every *concrete* trace has a corresponding *abstract* one with the same cost, but there might be spurious abstract traces, which do not correspond to concrete ones. Recall that our interest is in analysing ACR programs, the translation from JAVA is out of the scope of this paper. We first explain the language using some examples that we use along the paper. Then, we formally define its syntax, semantics and the concrete notions of cost. As a notation, we refer to line number `n` in a given JAVA (resp. ACR) program by `Jn` (resp. `An`).

Example 1. The JAVA code of the first example is depicted in Fig. 1 (on the left). It implements a Stack data-structure using a linked list whose first element is the top of the stack (field top points to this list). Method main has a loop (J14-19) that in each iteration invokes method randPop (J15), which in turn pops an arbitrary number of elements (J6-9), and then pushes a new element (J16). Note that coin() at J6 non-deterministically returns *true* or *false*. Each pop operation consumes m resources, as specified by the annotation @acquire(m) at J8, and each push consumes 1 resource (J17). This example is based on a classical example

```
1 class Stack {
2   Node top;
3
4   //@requires m >= 1
5   void randPop(int m) {
6     while( top != null && coin() ) {
7       top=top.next; //pop
8       //@acquire(m)
9     }
10  }
11
12  //@requires m >= 0
13  void main(int m) {
14    while( m > 0 ) {
15      randPop(m);
16      top = new Node('a',top); //push
17      //@acquire(1)
18      m = m−1;
19    }
20  }
21 }
```

```
1  rpop([s, m], [s1]) ←
2    m ≥ 1,
3    s ≥ 0,
4    s1 = s.
5  rpop([s, m], [s1]) ←
6    m ≥ 1,
7    s ≥ 1,
8    acq(m),
9    s2 = s − 1,
10   rpop([s2, m], [s1]).
11
12 main([s, m], [s1]) ←
13   m = 0,
14   s1 = s,
15 main([s, m], [s1]) ←
16   m ≥ 1,
17   rpop([s, m], [s2]),
18   s3 = s2 + 1,
19   acq(1),
20   m1 = m − 1,
21   main([s3, m1], [s1]).
```

Fig. 1. Java code for Stack and its ACR program.

for amortised analysis [9], the only difference is that pop costs m units instead of 1, to showcase some unique features of our analysis. These m units can be seen as the cost of executing m iterations of a loop (which we omit).

Fig. 1 (on the right) includes the ACR version of Stack. It has been automatically generated, and simplified for clarity, using the tools of [2]. A1-10 define a procedure *rpop* that corresponds to randPop. It has two input parameters: s is the size of the stack (i.e., the length of list top); and m is the value of variable m. Note that s is an abstraction of top. It also has one output parameter s_1 which corresponds to the size of the stack upon exit from randPop. Procedure *rpop* is defined by means of two rules: the first one (A1-4) corresponds to the case in which we do not enter the loop; and the second one (A5-10) corresponds to executing one iteration and calling *rpop* recursively (A10) for more iterations. The instruction $s_2 = s - 1$ at A9 corresponds to removing an element from the stack (J7). The translation of method main into procedure *main* (A12-20) is done in a similar way. Just note that calling *rpop* (A17) with a stack of size s results in a stack of size s_2, and that $s_3 = s_2 + 1$ at A18 corresponds to J16.

A call $main([s, m], [s_1])$ executes exactly m push operations, and thus, it can execute at most $s + m$ pop operations. Each push costs exactly 1, and each pop at most m. Since m varies from one call to *rpop* to another, then $s{\cdot}m + \frac{1}{2}(m^2 + m)$ is an UBF on the resource consumption of $main([s, m], [s_1])$. The analysis of [2] infers the cubic UBF $m^3 + s{\cdot}m^2 + m$, which is asymptotically less precise.

```
1 //@requires n>=0
2 void p(int n) {
3   if (n > 0) {
4     m = q(n);
5     //@release(m)
6     p(n−m);
7     //@release(m)
8   }
9 }
```

```
10 //@requires n>=1
11 int q(int n) {
12   int i = n/2;
13   do {
14     A x = new A();
15     B y = new B();
16     //@acquire(2)
17     i−−;
18     // [...]
19   } while(i>0 && coin());
20   return n/2 − i;
21 }
```

```
1  p([n], [ ]) ←
2    n = 0.
3  p([n], [ ]) ←
4    n ≥ 1,
5    q([n], [m]),
6    rel(m),
7    n1 = n − m,
8    p([n1], [ ]),
9    rel(m).
10 l([i], [i1]) ←
11   i ≥ 0,
12   acq(2),
13   i1 = i − 1.
14 l([i], [i1]) ←
15   i ≥ 1,
16   acq(2),
17   i2 = i − 1,
18   l([i2], [i1]).
19 q([n], [m]) ←
20   n ≥ 1
21   i = n/2,
22   l([i], [i1]),
23   m = i − i1.
```

Fig. 2. Java code for the peak, and its ACR program.

Example 2. The second example is depicted in Fig. 2. We use it to explain the notion of *peak* resource consumption. Method `q` (J10-21) receives an integer `n`, executes at least `1` and at most `n/2` iterations of a loop (J13-19), and returns the number of iterations that have been performed. This loop creates 2 objects in each iteration (J14-15). Method `p` executes a loop (using recursion) where in each iteration it calls `q` with the current value of the loop counter `n`, and then performs a recursive call where the loop counter is decremented by `m` (the number of iterations that `q` has performed). The ACR version, depicted in Fig. 2 on the right, its relation to the JAVA code is as in Ex. 1. We skip details and only comment that procedure l (A10-23) corresponds to the while loop (J13-19). Note that the ACR includes explicit resource release instructions (A6 and A9).

A call $p([n], [\,])$ creates exactly $2{\cdot}n$ objects. However, assuming that objects of type `A` (resp. `B`) become unreachable at J5 (resp. J7), then m objects can be garbage collected when reaching J5 (resp. J7). Thus, at any given moment there cannot be more than n reachable objects, which means that a memory for n objects (the peak consumption) is enough for safely executing this program. The analysis of [2,3] infers the UBF $\frac{n\cdot(n+1)}{2}$ which is asymptotically less precise.

In both programs of Exs. 1 and 2, the resource consumption is specified with the annotations $\texttt{acq}(e)$ and $\texttt{rel}(e)$, for acquiring and releasing e resources respectively. It should be clear that we are interested in inferring safe UBFs assuming the given annotations, and not in inferring the annotations.

Syntax Formally, an ACR program is a set of procedures. A procedure p is defined by a set of rules of the form $p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$ where $\bar{x}$ (resp. $\bar{y}$) is a sequence of input (resp. output) parameters, and each b_i is one of the following instructions: a (linear) constraint φ; a procedure call $q(\bar{w}, \bar{z})$; or a resource consumption instruction $\texttt{acq}(e)$ or $\texttt{rel}(e)$ where e is an arithmetic expression that evaluates to a non-negative value. In the rest of the paper we assume a given program P (to avoid repeating "for a given program P").

$$\textcircled{1}\ \frac{q(\bar{x},\bar{y}) \leftarrow \bar{b}' \in P}{\langle \psi, q(\bar{x},\bar{y}) \cdot \bar{b}\rangle \overset{0}{\rightsquigarrow} \langle \psi, \bar{b}' \cdot \bar{b}\rangle} \qquad \textcircled{2}\ \frac{\psi \wedge \varphi \not\models \mathit{false}}{\langle \psi, \varphi \cdot \bar{b}\rangle \overset{0}{\rightsquigarrow} \langle \psi \wedge \varphi, \bar{b}\rangle}$$

$$\textcircled{3}\ \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathtt{acq}(e) \cdot \bar{b}\rangle \overset{v}{\rightsquigarrow} \langle \psi, \bar{b}\rangle} \qquad \textcircled{4}\ \frac{\mathit{eval}(e,\psi) = v \geq 0}{\langle \psi, \mathtt{rel}(e) \cdot \bar{b}\rangle \overset{-v}{\rightsquigarrow} \langle \psi, \bar{b}\rangle}$$

Fig. 3. Semantics of ACR programs

Semantics A state s takes the form $\langle \psi, \bar{b}\rangle$, where $\bar{b}$ is a sequence of instructions pending for execution, and ψ is a constraint over $\mathit{vars}(\bar{b})$ and possibly other existentially quantified variables. The *store* ψ imposes relations between variables (e.g., $x = 1$, $x > y$). An execution starts from an initial state $\langle \bar{x} = \bar{v}, p(\bar{x},\bar{y})\rangle$, where $\bar{v}$ is a sequence of integers, which is then rewritten according to the rules in Fig. 3. These rules define a transition relation $s_1 \overset{v}{\rightsquigarrow} s_2$, meaning that there is a transition from s_1 to s_2 that consumes v resources. Rule ① handles procedure calls, it (non-deterministically) selects a rule from P that matches the call, and adds its instructions $\bar{b}'$ to the sequence of pending instructions. Variables in $\bar{b}'$ (except $\bar{x} \cup \bar{y}$) are renamed such that they are different from $\mathit{vars}(\bar{b}) \cup \mathit{vars}(\psi)$. Rule ② handles constraints by adding them to the store, if the resulting state is satisfiable. Rules ③-④ handle resource consumption. They evaluate e to a non-negative value v, and label the corresponding transition with v or $-v$.

The execution stops when no rule is applicable, which happens when the execution reaches (1) a *final state* $\langle \psi', \epsilon\rangle$ where ϵ is the empty sequence; or (2) a *blocking state* $\langle \psi', \varphi \cdot \bar{b}\rangle$ where $\varphi \wedge \psi' \models \mathit{false}$. A trace t is a finite or infinite sequence of states in which there is a valid transition between each pair of consecutive states. Traces that end in a final state and infinite traces are called *complete*. Namely, we exclude traces that end in a blocking state. We write $s_1 \overset{*}{\rightsquigarrow} s_2$ for a finite trace starting from s_1 and ending at s_2.

Definition 1 (trace cost). *Given a finite trace t, its net-cost $\tilde{\tau}(t)$ is the sum of the cost labels on its transitions. Given a complete trace t, its peak-cost $\hat{\tau}(t)$ is defined as* $\mathsf{max}\{\tilde{\tau}(t') \mid t' \text{ is a prefix of } t\}$.

Note that the peak-cost is always non-negative since the empty trace is a prefix of any trace t. However, the net-cost can be also negative. This is because we do not require that resources are acquired before they are released. This is useful for modeling consumer/producer programs, where the produced data can be viewed as resources. Though, we do not address such scenarios in this paper.

Definition 2 (procedure cost). *Given a procedure p with m input and n output parameters, its net-cost $\tilde{\pi}(p)$ and peak-cost $\hat{\pi}(p)$ are defined as*

$$\begin{aligned}\tilde{\pi}(p) = \{\langle \bar{v}_1, \bar{v}_2, \tilde{\tau}(t)\rangle &\mid \bar{v}_1 \in \mathbb{Z}^m, \bar{v}_2 \in \mathbb{Z}^n, t \equiv \langle \bar{x} = \bar{v}_1, p(\bar{x},\bar{y})\rangle \overset{*}{\rightsquigarrow} \langle \psi, \epsilon\rangle, \bar{y} = \bar{v}_2 \models \psi\}\\ \hat{\pi}(p) = \{\langle \bar{v}_1, \hat{\tau}(t)\rangle &\mid \bar{v}_1 \in \mathbb{Z}^m, t \text{ is a complete trace and starts in } \langle \bar{x} = \bar{v}_1, p(\bar{x},\bar{y})\rangle\}\end{aligned}$$

Intuitively, the net-cost tells what is the balance between the resources that have been acquired and released during the execution of p. Note that it only considers traces that terminate in a final state. The peak-cost tells what is the maximum amount of resources that a program can hold (i.e., acquired but not released yet) at any given state during the execution. Note that Def. 2 does not consider traces that terminate in a blocking state. This is because they do not correspond to valid traces in the JAVA program, and obtained due to the abstraction.

We say that $\mathcal{C} \geq 0$ resources are enough for safely executing $p(\bar{v}, \bar{y})$ without running out of resources if $\mathcal{C} \geq \mathsf{max}\{c \mid \langle \bar{v}, c\rangle \in \hat{\pi}(p)\}$. Note that for terminating programs that only acquires resources, one could also use $\mathcal{C} \geq \mathsf{max}\{c \mid \langle \bar{v}, \bar{v}', c\rangle \in \tilde{\pi}(p)\}$. This is the case for example of the Stack program. Our main interest is in inferring UBFs on the peak-cost of each procedure, however, this will require inferring first UBFs on the net-cost of each procedure p as we will see later.

3 Shortcomings of the classical approach to cost analysis

As explained in Sec. 1, the classical approach to cost analysis first transforms a given program into a set of CRs, and then solves these CRs into UBFs. The following CRs are automatically generated by [2] for the Stack program of Fig. 1

(1)	$rpop(s, m) = 0$	$\{m \geq 1 \wedge s \geq 0\}$
(2)	$rpop(s, m) = m + rpop(s_2, m)$	$\{m \geq 1 \wedge s \geq 1 \wedge s_2 = s - 1\}$
(3)	$main(s, m) = 0$	$\{m{=}0 \wedge s \geq 0\}$
(4)	$main(s, m) = 1 + rpop(s, m) + main(s_3, m_1)$	$\{m{\geq}1 \wedge s_3{=}s_2{+}1 \wedge m_1{=}m{-}1 \wedge \underline{s{\geq}s_2{\geq}0}\}$

Eqs. (1)-(2) capture the cost of executing procedure *rpop* on the input s and m, and Eqs. (3)-(4) capture the cost of executing procedure *main* on the input s and m. Eq. (4) states that when $m \geq 1$, the cost of executing $main(s, m)$ is 1 (for the push operation); plus the cost of executing $rpop(s, m)$; plus the cost of executing $main(s_3, m_1)$. The constraints on the right side of each equation define the applicability conditions for that equation (e.g., $m \geq 1$) and relations between its variables (e.g., $s_3 = s_2 + 1$). Note that the above CRs have a similar structure to the corresponding ACR program of Fig. 1.

A fundamental difference between ACRs and CRs is that the latter do not include the output parameters. For example, in Eq. (4), the output parameter s_2 in the call to *rpop* has been removed, and the constraint $s \geq s_2 \geq 0$ (underlined in Eq. (4)) has been added to indicate that, upon exit from *rpop*, the value of s_2 is non-negative and smaller than or equal to s. Note that this is the most precise relation between the input and the output parameters of *rpop*. This information is obtained by value analysis (at the level of the ACR program) that infers relations between the input and the output parameters [6].

CRs can be evaluated (they are similar to a functional program with constraints) to obtain the cost of a corresponding procedure. E.g., $main(v_1, v_2)$ can be evaluated to obtain the cost of executing $main([v_1, v_2], [y])$. Clearly, due to the non-determinism (e.g., in the constraints), the evaluation of $main(v_1, v_2)$ might result in several possible values. Soundness requires that the cost of any

trace for $main([v_1, v_2], [y])$ is a possible result for $main(v_1, v_2)$. Nevertheless, the interest is not in evaluating CRs, since it is like executing the ACR program, but rather in statically computing UBFs that bound their results. For example, the solver of [1] infers the UBF $m^3 + m^2{\cdot}s + m$ for $main(s, m)$. Intuitively, it does this as follows: (a) it infers the maximum number of iterations that *main* can perform, which is m; (b) it infers a worst-case behaviour for all iterations, which is $1 + (s + m){\cdot}m$ since the stack can have at most $s + m$ elements; and (c) it multiplies (a) and (b) to get the above UBF.

It is known that, in practice, cost analysers that are based on CRs fail to obtain the desired UBFs for programs like those in Fig. 1 and 2. Moreover, as yet, it was assumed that this failure is due to (i) the way CRs are solved into UBFs; and (ii) the imprecision in the value analysis which is used to infer input-output relations (as $s \geq s_2 \geq 0$ above). It was also assumed that, in principle, one could develop more sophisticated techniques for solving CRs [4] or use more precise value analysis (e.g., non-linear) that would obtain precise UBFs for such programs. In what follows we show that these assumptions are not true. In particular, that Eqs. (3)-(4) in the above CRs do not model *precisely* the cost of procedure *main*, and thus any sound UBF for *main* would be imprecise.

Let us consider an evaluation of $main(s, m)$ in the above CRs. It is easy to see that, using Eq. (4), we can choose $s_2 = s$ and thus get $main(s, m) = 1 + rpop(s, m) + main(s + 1, m - 1)$. Then, in the same way, we can get $main(s + 1, m - 1) = 1 + rpop(s + 1, m - 1) + main(s + 2, m - 2)$, and so on for each $main(s+i, m-i)$. Thus, an evaluation of $main(s, m)$ admits $\sum_{i=0}^{m-1}(1 + rpop(s + i, m - i))$ as a possible result. Since $rpop(s, m)$ can always evaluate to $s{\cdot}m$, the above sum can be reduced to $u(s, m) = \frac{(m-1)}{6}{\cdot}(m^2 + 3{\cdot}s{\cdot}m + m + 6)$. This means that any UBF $f(s, m)$ for Eqs. (3)-(4) must satisfy $\forall s, m : f(s, m) \geq u(s, m)$, which is asymptotically less precise than the UBF from Ex. 1. Thus, we conclude that the imprecision is not related to how CRs are solved, and not to imprecision in the value analysis since the input-output relation $s \geq s_2 \geq 0$ that we used above is the most precise one.

The actual reason for this imprecision is that, in Eq (4), the value for s_2, i.e., the output of *rpop*, and the cost of $rpop(m, s)$ can be chosen independently. For example, in the original program it is not possible that $s_2 = s$ and that the cost of $rpop(s, m)$ is $s{\cdot}m$, in which case s_2 must be 0. However, in the above CRs this scenario is possible. This relation cannot be captured if the UBFs are defined only in terms of the input parameters, an observation that lead us to the idea of defining UBFs in terms of both input and output parameters.

Example 3. Consider again procedure $rpop([s, m], [s'])$ of Fig. 1. The CRs-based approach infers the UBF $s{\cdot}m$ for *rpop*, which depends only on the input parameters s and m. This indeed is the most precise UBF if only input parameters are allowed, since there exists an execution in which we remove all stack elements. However, if we allow the use of output parameters also, then $(s - s'){\cdot}m$ describes the exact cost of *rpop*: $s - s'$ is the number of elements that have been removed from the stack, and removing each one costs m.

At this point, the use of output parameters to define UBFs might look inappropriate. This is because UBFs are usually used to *statically* estimate the amount of resource required for safely executing the program. However, requiring information on the output parameters in order to evaluate a given UBF is like actually requiring to execute the program. This is not really the case because of the following two reasons. First, when inferring UBFs on the net-cost, we distinguish between the entry procedure (e.g., *main*), and intermediate procedures (e.g., *rpop*). The UBF for the entry procedure will (almost always) be definable in terms of its input parameters only, however, in order to infer a precise UBF for the entry procedure, we need UBFs for the intermediate procedures in terms of input and output parameters. Second, UBFs on the peak-cost, which are the important ones for safety, will use only input parameters, however, inferring them will make use of net-cost UBFs that depend on input and output parameters.

4 Inference of net-cost

In this section we describe our approach for inferring UBFs on the net-cost of the program's procedures, which is based on defining the cost in terms of the input and output parameters. We show that it can infer the precise cost of the Stack example of Fig. 1. In Sec. 5, we extend it to infer UBFs on the peak-cost.

Definition 3 (safe net-cost UBFs). *Let p be a procedure with n input and m output parameters. A function $\tilde{f}_p : \mathbb{Z}^{n+m} \mapsto \mathbb{Q}$ is a safe UBF on the net-cost of p iff for any $\langle \bar{v}_1, \bar{v}_2, c\rangle \in \tilde{\pi}(p)$ it holds $\tilde{f}_p(\bar{v}_1, \bar{v}_2) \geq c$.*

Intuitively, a function $\tilde{f}_p$ is an UBF on the net-cost of p if for any possible execution that starts with input $\bar{v}_1$, terminates in a final state with an output $\bar{v}_2$, and have net-cost c, it holds that $\tilde{f}_p(\bar{v}_1, \bar{v}_2) \geq c$. Clearly, CRs cannot be used to infer such UBFs, since they do not use the output parameters.

In what follows we develop a novel approach for inferring such UBFs that is based on the use of quantifier elimination. We present our approach in two steps: (1) *verification*: in which we are given a set of *candidate* UBFs on the net-cost of each procedure, and our interest is to verify that these functions are safe, i.e., satisfy Def. 3; and (2) *inference*: in which we are given a set of *template* UBFs, and our interest is to instantiate the templates parameters into safe UBFs.

Verification of UBFs on the net-cost Let us start by explaining the basics of the verification step. Assume that we have a procedure p defined by the following single rule

$$p(\bar{x}, \bar{y}) \leftarrow \texttt{acq}(e), q_1(\bar{x}_1, \bar{y}_1), \ldots, q_n(\bar{x}_n, \bar{y}_n)$$

and that we have a set of *safe* UBFs $\tilde{f}_{q_1}, \ldots, \tilde{f}_{q_n}$ on the net-cost of $q_1, \ldots, q_n$. To verify that a given $\tilde{f}_p$ is a safe UBF on the net-cost of p, it is *sufficient* to check that the condition $\tilde{f}_p(\bar{x}, \bar{y}) \geq e + \tilde{f}_{q_1}(\bar{x}_1, \bar{y}_1) + \cdots + \tilde{f}_{q_n}(\bar{x}_n, \bar{y}_n)$ holds for any values of the program variables. Applying this principle to all rules of the program, it is possible to verify the safety of several candidate UBFs simultaneously.

Given a set $\tilde{F}$ of candidate UBFs on the net-cost that includes a function $\tilde{f}_p : \mathbb{Z}^{n+m} \mapsto \mathbb{Q}$ for each procedure $p \in P$, we build a *verification condition* (VC for short) whose validity implies the safety of each $\tilde{f}_p \in \tilde{F}$. The net-cost VC is generated from the program rules as follows.

Definition 4 (Net-cost VC). *Given a set $\tilde{F}$ of candidate UBFs, for each rule $r \equiv p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$, we generate a condition ψ_r as follows:*

1. *let φ be the conjunction of all constraints in r;*
2. *let the net-cost $\tilde{b}$ of an instruction b be defined as follows: if $b \equiv q_i(\bar{x}_i, \bar{y}_i)$ then $\tilde{b} \equiv \tilde{f}_q(\bar{x}_i, \bar{y}_i)$, if $b \equiv \texttt{acq}(e)$ then $\tilde{b} \equiv e$, if $b \equiv \texttt{rel}(e)$ then $\tilde{b} \equiv -e$, and if b is a constraint then $\tilde{b} \equiv 0$;*
3. *let $\psi_r \equiv \forall \bar{w} : \varphi \Rightarrow \tilde{f}_p(\bar{x}, \bar{y}) \geq \tilde{b}_1 + \cdots + \tilde{b}_n$ where $\bar{w} = vars(r)$.*

Then, the net-cost VC is defined as $\Psi(\tilde{F}) = \wedge_{r \in P} \psi_r$.

Note that ψ_r is the condition we explained before, but taking into account the constraints φ of the rule r which define the context in which this condition holds.

Example 4. Consider the program in Fig. 1, and let $\tilde{f}_r(s, m, s_1)$ and $\tilde{f}_m(s, m, s_1)$ be candidate UBFs on the net-cost of $rpop([s, m], [s_1])$ and $main([s, m], [s_1])$, respectively. The verification condition for this program w.r.t. $\tilde{F} = \{\tilde{f}_r(s, m, s_1), \tilde{f}_m(s, m, s_1)\}$ is $\Psi(\tilde{F}) = \psi_{r_1} \wedge \psi_{r_2} \wedge \psi_{r_3} \wedge \psi_{r_4}$ where:

$$\psi_{r_1} \equiv \forall \bar{w}_1 : m \geq 1 \wedge s \geq 0 \wedge s_1 = s \Rightarrow \tilde{f}_r(s, m, s_1) \geq 0$$
$$\psi_{r_2} \equiv \forall \bar{w}_2 : m \geq 1 \wedge s \geq 1 \wedge s_2 = s - 1 \Rightarrow \tilde{f}_r(s, m, s_1) \geq m + \tilde{f}_r(s_2, m, s_1)$$
$$\psi_{r_3} \equiv \forall \bar{w}_3 : m = 0 \wedge s_1 = s \wedge s \geq 0 \Rightarrow \tilde{f}_m(s, m, s_1) \geq 0$$
$$\psi_{r_4} \equiv \forall \bar{w}_4 : \begin{array}{l} m \geq 1 \wedge s_3 = s_2 + 1 \wedge \\ m_1 = m - 1 \wedge s \geq 0 \end{array} \Rightarrow \tilde{f}_m(s, m, s_1) \geq \tilde{f}_r(s, m, s_2) + 1 + \tilde{f}_m(s_3, m_1, s_1)$$

The condition ψ_{r_4}, for example, corresponds to the second rule of procedure *main*. It states that $\tilde{f}_m(s, m, s_1)$ is a safe UBF if it is greater than the cost of the call to *rpop*, i.e., $\tilde{f}_r(s, m, s_2)$, plus 1 for the push operation, plus the cost of the recursive call to *main*, i.e, $\tilde{f}_m(s_3, m_1, s_1)$. This condition should hold for any values that satisfy the constraint $m \geq 1 \wedge s_3 = s_2 + 1 \wedge m_1 = m - 1 \wedge s \geq 0$, i.e., in the context of the second rule. Let us consider now the validity of $\Psi(\tilde{F})$ for the following possible concrete definitions of $\tilde{f}_r(s, m, s_1)$ and $\tilde{f}_m(s, m, s_1)$

(a) $\tilde{f}_m(s, m, s_1) = s{\cdot}m + \frac{1}{2}(m^2 + m)$, and $\tilde{f}_r(s, m, s_1) = (s - s_1){\cdot}m$
(b) $\tilde{f}_m(s, m, s_1) = s{\cdot}m + \frac{1}{2}(m^2 + m)$, and $\tilde{f}_r(s, m, s_1) = s{\cdot}m$

Using (a), we get that $\Psi(\tilde{F})$ is a valid formula. Note that here we use the optimal UBFs for *main* and *rpop*. Using (b), we get that $\Psi(\tilde{F})$ is invalid, though both UBF are safe. This is because, in this case, using $s{\cdot}m$ as an UBF for *rpop* is not enough for proving that $s{\cdot}m + \frac{1}{2}(m^2 + m)$ is an UBF for *main*.

Theorem 1. *Given a set $\tilde{F}$ of candidate UBFs, if $\models \Psi(\tilde{F})$ then $\tilde{F}$ is safe.*

Note that checking the validity of $\Psi(\tilde{F})$ is a first order problem that can be solved using SMT solvers (see Sec. 7).

Inference of UBFs on the net-cost For many applications it is useful to infer the set $\tilde{F}$, instead of verifying the correctness of a given one. This can be formulated as seeking a set $\tilde{F}$ of UBFs for which $\Psi(\tilde{F})$ is valid, which means solving the formula $\exists \tilde{f}_1 \tilde{f}_2 \ldots \tilde{f}_k : \Psi(\tilde{F})$. However, this is a second order problem and solving it in general is impractical. A common approach to avoid solving a second order formula is the use of template functions that restrict the form of functions that we are looking for. A template for $\tilde{f}_p(\bar{x}, \bar{y})$ is a function with a fixed structure, defined over the variables $\bar{x} \cup \bar{y}$, and some unknown template parameters.

Example 5. The following are UBF templates for procedure *main* and *rpop*:

1. $\tilde{f}_r(s, m, s_1) = \lambda_1 \cdot s \cdot m + \lambda_2 \cdot s_1 \cdot m + \lambda_3 \cdot s + \lambda_4 \cdot m + \lambda_5 \cdot s_1 + \lambda_0$
2. $\tilde{f}_m(s, m, s_1) = \mu_1 \cdot s \cdot m + \mu_2 \cdot m^2 + \mu_3 \cdot s_1 \cdot m + \mu_4 \cdot s + \mu_5 \cdot m + \mu_6 \cdot s_1 + \mu_0$

The variables $\bar{\lambda}$ and $\bar{\mu}$ are the template parameters.

Assuming that $\tilde{F}$ is a set of candidate UBF templates, and that $\mathcal{P}$ is the set of template parameters, the inference problem is reduced to solving the first order problem $\exists \mathcal{P} : \Psi(\tilde{F})$. This can be solved by combining quantifier elimination and SMT solvers (see Sec. 7). The idea behind UBF templates is that later we will assign values to the template parameters such that the resulting UBFs are safe.

Note that in Ex. 5 we have chosen simple templates just to keep the technical details in the next examples simple. We could also choose a cubic polynomial template, and later try to find an instantiation such that the parameters of the cubic parts are assigned 0 (in order to get the quadratic UBF). In principle, any template UBF can be used as far as it uses arithmetic expressions that are supported by the quantifier elimination procedure (see Sec. 7).

Example 6. Using the templates of Ex. 5 in the VC of Ex. 4, we get a VC $\Psi(\tilde{F})$ in which the template variables $\bar{\lambda} \cup \bar{\mu}$ are free variables. Eliminating the universally quantified variables, we get a formula ξ over $\bar{\lambda} \cup \bar{\mu}$ that is a conjunction of the following equalities and inequalities:

$$\begin{array}{l|l|l|l|l} \lambda_1 \geq 1 & \lambda_2 = -\lambda_1 & \lambda_1 + \lambda_3 \geq 1 & \mu_6 \geq \lambda_5 - \lambda_1 & 2 \cdot \mu_2 \geq \lambda_1 + \lambda_4 \\ \lambda_4 \geq 0 & \mu_1 = \lambda_1 & \lambda_3 + \lambda_5 \geq 0 & \mu_4 = \lambda_1 - \lambda_5 & \mu_5 + \mu_2 \geq \mu_4 + \lambda_0 + \lambda_4 + 1 \\ \mu_0 \geq 0 & \mu_3 = 0 & \lambda_0 + \lambda_4 \geq 0 & \lambda_1 \geq \lambda_3 + \lambda_5 & \end{array}$$

Each model of ξ assigns values to the template parameters $\bar{\lambda}$ and $\bar{\mu}$ such that $\tilde{f}_r(s, m, s_1)$ and $\tilde{f}_m(s, m, s_1)$ of Ex. 5 are safe UBFs for *rpop* and *main* respectively. For example, it is easy to check that

$$\mu_1 = 1, \mu_2 = \mu_5 = \frac{1}{2}, \mu_4 = \mu_6 = \mu_0 = 0, \lambda_1 = 1, \lambda_2 = -1, \lambda_3 = \lambda_4 = \lambda_5 = 0$$

is a model of ξ, which corresponds to the desired UBFs $s \cdot m + \frac{1}{2}(m^2 + m)$ and $(s - s_1) \cdot m$ for procedures *main* and *rpop* respectively. It is worth noting the inequalities $\lambda_1 \geq 1$ and $\lambda_2 = -\lambda_1$, meaning that any UBF for *rpop must* involve both $s \cdot m$ and $s_1 \cdot m$ (recall that s_1 is its output parameter). If we analyse *rpop* alone this would not be the case, and UBFs like $s \cdot m$ would be possible, however, this is essential in order to obtain the quadratic UBF for *main*.

It is important to note that once the constraints over the template parameters (i.e., ξ in the above example) are generated, then one should try to find a model of ξ that results in a tight UBF. This process usually depends on the kind of expression used in the templates. For example, in the case of polynomial templates one could try to first set the parameters of the higher degree components to 0, etc. Another possibility is to start from a polynomial with low degree, and increment it gradually until an UBF is found.

5 Inference of peak-cost

When a given program only acquires resources, the net-cost analysis can be used to estimate the amount of resources required for safely executing the program. This, however, is not the case when the program can also release resources. For example, the net-cost of the program in Fig 2 is 0, since all resources are released either at J5 or J7, however, it requires at least $n+1$ resources in order to execute correctly. In order to estimate the amount of resources required for safely executing such programs, what we need is the peak-cost, which is the maximum amount of resources that a program can hold simultaneously.

Definition 5 (safe peak-cost UBFs). *Let p be a procedure with n input parameters. A function $\hat{f}_p : \mathbb{Z}^n \mapsto \mathbb{Q}$ is a safe UBF on the peak-cost of p, iff for any $\langle \bar{v}_1, c\rangle \in \hat{\pi}(p)$ it holds $\hat{f}_p(\bar{v}_1) \geq c$.*

Our approach for inferring UBFs on the peak-cost is done in two steps, verification and inference, similar to the case of net-cost.

Verification of UBFs on the peak-cost Let us start by explaining the basics of the verification step. Assume that we have a procedure p defined by the following single rule

$$p(\bar{x}, \bar{y}) \leftarrow q_1(\bar{x}_1, \bar{y}_1), q_2(\bar{x}_2, \bar{y}_2)$$

and assume that we have UBFs $\hat{f}_{q_1}$ and $\hat{f}_{q_2}$ on the peak-cost of q_1 and q_2 respectively. We are interested in verifying that a given function $\hat{f}_p(\bar{x})$ is indeed a safe UBF on the peak-cost of p. When executing p, the peak-cost might be reached while executing q_1 or q_2. If it is reached during q_1, then the peak-cost of p is like that of q_1, and if it is reached during q_2, then the peak-cost of p is like that of q_2 plus the amount of resources that p holds before calling q_2. Now note that this last amount is exactly the net-cost of q_1. Thus, in order to verify the correctness of $\hat{f}_p$ it is sufficient to check that the condition $\hat{f}_p(\bar{x}) \geq \hat{f}_{q_1}(\bar{x}_1) \wedge \hat{f}_p(\bar{x}) \geq \tilde{f}_{q_1}(\bar{x}_1, \bar{y}_1) + \hat{f}_{q_2}(\bar{x}_1)$ holds for any values of the program variables, where $\tilde{f}_{q_1}(\bar{x}_1, \bar{y}_1)$ is a safe UBF on the *net-cost* of q_1. Applying this principle to all rules of the program, it is possible to verify the correctness of several UBFs simultaneously.

Given a set $\hat{F}$ of candidate UBFs on the peak-cost, which includes a function $\hat{f}_p : \mathbb{Z}^n \mapsto \mathbb{Q}$ for each procedure $p \in P$, we want to build a VC whose validity

implies that each $\hat{f}_p$ is indeed a safe UBF. For this, we assume a given set $\tilde{F}$ of safe UBFs on the net-cost of each procedure (later we will see that $\hat{F}$ and $\tilde{F}$ can be verified or inferred simultaneously). The peak-cost VC, denoted by $\Phi(\tilde{F}, \hat{F})$, is generated from the program rules as we explain next.

Definition 6 (Peak-cost VC). *Let $\hat{F}$ be a set of candidate UBFs on the peak-cost, and $\tilde{F}$ be a set of safe UBFs on the net-cost. For each rule $r \equiv p(\bar{x}, \bar{y}) \leftarrow b_1, b_2, \ldots, b_n$, we generate a condition ϕ_r according to the following steps*

1. *let $b_{\ell_1}, \ldots, b_{\ell_k}$, with $1 \leq \ell_1 < \cdots < \ell_k \leq n$, be all elements of the body that are of the form $q_{\ell_i}(\bar{x}_{\ell_i}, \bar{y}_{\ell_i})$ or $\texttt{acq}(e)$. We assume there is at least one such element, otherwise we add $\texttt{acq}(0)$ at the end of r;*
2. *let φ_i be the conjunction of all constraints in r up to b_{ℓ_i};*
3. *the peak-cost $\hat{b}_{\ell_i}$ of an instruction b_{ℓ_i} is defined as follows: if $b_{\ell_i} \equiv q_{\ell_i}(\bar{x}_{\ell_i}, \bar{y}_{\ell_i})$ then $\hat{b}_{\ell_i} \equiv \hat{f}_q(\bar{x}_{\ell_i})$, and if $b_{\ell_i} \equiv \texttt{acq}(e)$ then $\hat{b}_{\ell_i} \equiv e$;*
4. *let ϕ_r be the formula below where $\bar{w} = vars(r)$ and $\tilde{b}_j$ are as in Def. 4.*

$$\phi_r \equiv \underbrace{(\wedge_{i=1}^{k} \forall \bar{w} : \varphi_i \Rightarrow \hat{f}_p(\bar{x}) \geq (\Sigma_{j=1}^{\ell_i - 1} \tilde{b}_j) + \hat{b}_j))}_{\mathcal{A}} \wedge \underbrace{(\forall \bar{w} : \varphi_1 \Rightarrow \hat{f}_p(\bar{x}) \geq 0))}_{\mathcal{B}}$$

Then, the peak-cost VC is $\Phi(\tilde{F}, \hat{F}) = \wedge_{r \in P} \phi_r$.

Let us explain the parts of ϕ_r: ($\mathcal{A}$) this part generalises the intuition that we have explained before. Intuitively, the instructions $b_{\ell_1}, \ldots, b_{\ell_k}$ are those that might *increase* the resource consumption, thus, the peak-cost of p should be greater than or equal to the peak-cost $\hat{b}_{\ell_i}$ of each b_{ℓ_i} plus the resources $\sum_{j=1}^{\ell_i - 1} \tilde{b}_j$ that p holds before executing $\hat{b}_{\ell_i}$ (note the use of the net-cost $\tilde{b}_j$); and ($\mathcal{B}$) this part requires that the peak function is non-negative. Note that in principle we should require $\forall \bar{w} : \varphi_i \Rightarrow \hat{f}_p(\bar{x}) \geq 0$ for all $i \in [1 \ldots k]$, however, requiring $\mathcal{B}$ is enough since $\varphi_i \Rightarrow \varphi_1$ for all $i \in [2 \ldots k]$. In the examples below we sometimes omit the second part $\mathcal{B}$ when it is redundant.

Example 7. The peak-cost VC for the program of Fig. 2, w.r.t. (some generic) $\tilde{F}$ and $\hat{F}$, is $\Phi(\tilde{F}, \hat{F}) = \phi_{r_1} \wedge \cdots \wedge \phi_{r_5}$ where

$$\begin{aligned}
\phi_{r_1} &\equiv \forall \bar{w}_1 : n = 0 \Rightarrow \hat{f}_p(n) \geq 0 \\
\phi_{r_2} &\equiv (\forall \bar{w}_2 : n \geq 1 \Rightarrow \hat{f}_p(n) \geq \hat{f}_q(n)) \wedge \\
&\quad\ (\forall \bar{w}_2 : n \geq 1 \wedge n_1 = n - 1 \Rightarrow \hat{f}_p(n) \geq \tilde{f}_q(n, m) - m + \hat{f}_p(n_1)) \wedge \\
&\quad\ (\forall \bar{w}_2 : n \geq 1 \Rightarrow \hat{f}_p(n) \geq 0) \\
\phi_{r_3} &\equiv \forall \bar{w}_3 : i \geq 0 \Rightarrow \hat{f}_l(i) \geq 2 \\
\phi_{r_4} &\equiv (\forall \bar{w}_4 : i \geq 1 \Rightarrow \hat{f}_l(i) \geq 2) \wedge (\forall \bar{w}_4 : i \geq 1 \wedge i_2 = i - 1 \Rightarrow \hat{f}_l(i) \geq 2 + \hat{f}_l(i_2)) \\
\phi_{r_5} &\equiv (\forall \bar{w}_5 : n \geq 1 \wedge i = \tfrac{n}{2} \Rightarrow \hat{f}_q(n) \geq \hat{f}_l(i)) \wedge (\forall \bar{w}_5 : n \geq 1 \wedge i = \tfrac{n}{2} \Rightarrow \hat{f}_q(n) \geq 0)
\end{aligned}$$

Formula ϕ_{r_2}, for example, corresponds to the second rule of procedure p. It consists of 3 subformulas, the first two are the $\mathcal{A}$-part and the last is the $\mathcal{B}$-part. In the second, note the expression $\tilde{f}_q(n, m) - m$ which is the amount of resource that p holds before the recursive call to p. Using $\tilde{f}_q(n, m) = 2 \cdot m$, $\hat{f}_p(n) = n + 2$,

$\hat{f}_q(n) = n+2$, and $\hat{f}_l(i, i_1) = 2{\cdot}i_1+2$, it is possible to verify that $\Phi(\tilde{F}, \hat{F})$ is valid. However, using another safe UBF on the net-cost of q, e.g., $\tilde{f}_q(n, m) = n + 2$, then $\Phi(\tilde{F}, \hat{F})$ is not valid. Indeed, $2{\cdot}m$ is the most precise UBF on the net-cost of q, and is the one needed to verify the above UBF on the peak-cost of p.

Theorem 2. *Given a set $\tilde{F}$ of safe UBFs on the net-cost (Th. 1), and a set $\hat{F}$ of candidate UBFs on the peak-cost, if $\models \Phi(\tilde{F}, \hat{F})$, then $\hat{F}$ is safe.*

As in the case of $\Psi(\tilde{F})$ cost, checking the validity of $\Phi(\tilde{F}, \hat{F})$ reduces to a satisfiability problem of first order logic.

Inferring UBFs on the peak-cost Our main interest is in inferring $\hat{F}$ rather than verifying the correctness of a given one. This can be done using template UBFs as the case of net-cost. However, an important point is that instead of assuming a given set $\tilde{F}$ of UBFs on the net-cost, we can infer it at the same time as $\hat{F}$, simply by considering the VC $\Phi(\tilde{F}, \hat{F}) \wedge \Psi(\tilde{F})$. This is actually essential in practice, since as we have seen in Ex. 7 not any safe UBF on the net-cost can be used to infer the peak-cost. Inferring them simultaneously will force choosing the required one.

Example 8. Let $\tilde{F}$ and $\hat{F}$ be defined by the following linear UBF templates:

$$\begin{array}{lll} \tilde{f}_p(n) = \lambda_1{\cdot}n + \lambda_2 & \tilde{f}_q(n, m) = \lambda_3{\cdot}n + \lambda_4{\cdot}m + \lambda_5 & \tilde{f}_l(i, i_1) = \lambda_6{\cdot}i + \lambda_7{\cdot}i_1 + \lambda_8 \\ \hat{f}_p(n) = \mu_1{\cdot}n + \mu_2 & \hat{f}_q(n) = \mu_3{\cdot}n + \mu_4 & \hat{f}_l(i) = \mu_5{\cdot}i + \mu_6 \end{array}$$

and let $\Phi(\tilde{F}, \hat{F})$ be the VC of Ex. 7 using these $\tilde{F}$ and $\hat{F}$. Moreover, let $\Psi(\tilde{F}) = \psi_{r_1} \wedge \cdots \wedge \psi_{r_5}$ be the corresponding net-cost VC using the above $\tilde{F}$, where

$$\begin{array}{l} \psi_{r_1} \equiv \forall \bar{w}_1 : n = 0 \Rightarrow \tilde{f}_p(n) \geq 0 \\ \psi_{r_2} \equiv \forall \bar{w}_2 : n \geq 1 \wedge n_1 = n - 1 \Rightarrow \tilde{f}_p(n) \geq \tilde{f}_q(n, m) - m + \tilde{f}_p(n_1) + m \\ \psi_{r_3} \equiv \forall \bar{w}_3 : i \geq 0 \wedge i_1 = i - 1 \Rightarrow \tilde{f}_l(i, i_1) \geq 2 \\ \psi_{r_4} \equiv \forall \bar{w}_4 : i \geq 1 \wedge i_2 = i - 1 \Rightarrow \tilde{f}_l(i, i_1) \geq 2 + \tilde{f}_l(i_2, i_1) \\ \psi_{r_5} \equiv \forall \bar{w}_5 : n \geq 1 \wedge i = \frac{n}{2}, m = i - i_1 \Rightarrow \tilde{f}_q(n, m) \geq \tilde{f}_l(i, i_1) \end{array}$$

Then, applying quantifier elimination on $\Phi(\tilde{F}, \hat{F}) \wedge \Psi(\tilde{F})$ to eliminate the universally quantified variables, we get a formula ξ over the template parameters that is a conjunction of the following equalities and inequalities

$$\begin{array}{l|l|l|l|l} \lambda_2 \geq 0 & \lambda_3 = 0 & \lambda_8 \leq \lambda_5 \leq 0 & \lambda_1 = \lambda_4 - 2 & \lambda_6 + \lambda_8 \geq 2 \\ \lambda_6 \geq 2 & \mu_6 \geq 2 & \mu_1 = \lambda_1 + 1 & \lambda_4 = \lambda_6 = -\lambda_7 & 2{\cdot}\mu_6 + \mu_5 \leq 2{\cdot}\mu_4 + 2{\cdot}\mu_3 \\ \mu_5 \geq 2 & \mu_2 \geq 0 & 2{\cdot}\mu_3 \geq \mu_5 & \lambda_6 \geq \mu_3 + 1 & \mu_1 + \mu_2 \geq \mu_3 + \mu_4 \end{array}$$

The models of ξ define possible instantiations $\tilde{F}$ and $\hat{F}$ such that they are safe UBFs. E.g., there is a model of ξ with $\mu_1 = 1$ and $\mu_2 = 2$ which defines the UBF $n + 2$ on the peak-cost of p. Note the constraint $\lambda_3 = 0$, which means that the UBF on the net-cost of q must not depend on the input n (in Ex. 7 we failed with $\tilde{f}_q(n, m) = n + 2$). This demonstrates how the peak-cost VC affects the net-cost one. Note that, for p, we have inferred the UBF $n + 2$ and not the optimal one $n + 1$ because the quantifier elimination is done over $\mathbb{R}$ and not over $\mathbb{Z}$.

Example 9. Let us finish with an example of a non-terminating program. Consider the following (contrived) program, which is defined by a single rule

$$p([n],[y_1]) \leftarrow n \geq m \geq 0, \mathtt{acq}(m), n_1 = n - m, p([n_1],[y_1]).$$

Procedure p receives a non-negative integer n, non-deterministically chooses a non-negative value $m \leq n$, acquires m resources, and then calls p recursively with $n - m$. The peak-cost of this program is exactly n, since any infinite trace cannot acquire more than n resources and there are infinite traces that acquire exactly n. The peak-cost VC for this program is

$$(\forall n, m : n \geq m \geq 0 \Rightarrow \hat{f}_p(n) \geq m) \wedge (\forall n, m : n \geq m \geq 0 \Rightarrow \hat{f}_p(n) \geq m + \hat{f}_p(n_1)$$

Assuming the template UBF $\hat{f}_p(n) = \lambda_1 \cdot n + \lambda_2$, the elimination of the universally quantified variables result in the formula $\xi = \lambda_1 \geq 1 \wedge \lambda_2 \geq 0$. Since $\lambda_1 = 1$ and $\lambda_2 = 0$ is a model of ξ, then $\hat{f}_p(n) = n$ is a safe UBF.

6 Relation to Amortised Cost Analysis

In this section we discuss an interesting relation that we have observed between UBFs that are defined in terms of both input and output parameters, and the notion of potential functions used in the context of amortised cost analysis. This may provide a semantics-based explanation to why amortised analysis can obtain more precise UBFs.

A potential function, in the context of an ACR, is a function that maps a given state to a non-negative rational number, which is called the potential of the state. This potential can be interpreted as the amount of resources available in the given state. An automatic amortised cost analysis [15] assigns to each procedure $p(\bar{x}, \bar{y})$ two potential functions: *input* $P_p(\bar{x})$, and *output* $Q_p(\bar{y})$. Intuitively, the input potential $P_p(\bar{x})$ must be large enough to pay for the cost of executing $p(\bar{x}, \bar{y})$, and, upon exit, leaving at least $Q_p(\bar{y})$ resources to be consumed later. Thus, if c is the net-cost of p, then $P_p(\bar{x}) \geq c + Q_p(\bar{y})$ must hold. This later expression can be rewritten as $P_p(\bar{x}) - Q_p(\bar{y}) \geq c$, which means that $P_p(\bar{x}) - Q_p(\bar{y})$ is an UBF on the net-cost of p, but also is an UBF that uses input and the output parameters. Thus, the above potential functions are in principle UBFs as defined in Def. 3, however, they are just a special case since $P_p(\bar{x}) - Q_p(\bar{y})$ does not allow using, for example, expressions like $s_1 \cdot m$.

We have tried to analyse (a functional version of) the Stack example using the amortised analysis of [15], which uses the above notion of potential functions. The analysis failed to obtain the expected quadratic UBF, and instead, it reported a cubic UBF. This failure confirms that it is essential to define the output potential for *rpop* as $s_1 \cdot m$, which cannot be defined using the above kind of potential functions. Note that this should not be interpreted as a fundamental limitation of [15], since their underlying machinery can be easily adapted to support potential functions of this form. In addition, the above discussion should be considered only in the context of the ACR language, since amortised analysis has many other features that goes beyond the ACR language.

7 Implementation and Experiments

A prototype implementation of our analysis is available at `http://costa.ls.fi.upm.es/acrp`. It receives as input an ACR program and a set of template UBFs. Then, it generates the VCs described in Secs. 4 and 5 as a REDUCE script [20], executes the script to eliminate the universally quantified variables, and finally outputs the template parameters constraints in SMT2-LIB format, which can be then solved using off-the-shelf SMT solvers.

For the quantifier elimination, the REDUCE script uses the REDLOG package [11] with the theory of *real closed fields*. This theory allows using a wide range of template UBFs, such as multivariate polynomial, max and min operations, etc. As done in [19], REDLOG can be switched to use SLFQ [7], which is a formula simplifier for the theory of real closed fields. Using SLFQ significantly reduces the size of the template parameters constraints, and thus improves the overall performance. For solving the template parameters constraints we have used Z3 [21], employing the logic of *non linear real arithmetic* (QF_NRA). Currently, we only ask the SMT solver for a satisfying assignment, which in turn instantiate the templates to safe UBFs. Looking for an assignment that gives the tightest UBFs is left for future work.

We have applied the analyser on small examples collected from cost analysis literature. All are available in the above address. For these examples we obtained the expected precise UBFs. Unfortunately, being based on real quantifier elimination, our procedure does not yet scale for large programs. In a future work we plan to explore patterns of ACR programs for which (a variation of) the analysis scales, e.g., for the case of the multivariate polynomials of [15].

8 Related Work

Static cost analysis dates back to the seminal work of Wegbreit [22]. Recently it has received a considerable attention which resulted in several cost analysers for different programming languages [2,10,12,15]. The research in this paper is mostly related to [2] and [15], in the sense that our research was motivated by the limitations of [2], and our solution turned to have common ideas with of [15] as we have explained in Sec. 6. When comparing [15], the advantage of our analysis is in that it has a more general notion of potential functions, it is not limited to polynomial templates, and can handle variables with negative values. However, unlike ours, their techniques can handle data-structures by assigning potentials to its parts, and their tool is reasonably scalable and performs very well in practice.

Our peak-cost constraints are similar to those of [3], they were used for inferring memory consumption in the presence of garbage collection. The limitations of CRs have been considered also in [4], but from a different perspective. Solving CRs using template function and real quantifier elimination has been considered before in [5]. However, it cannot handle the limitations we pointed out in this paper, and cannot handle non-terminating programs. Also [13,23] deal with

similar problems, however, they cannot handle the limitation described in this paper, and cannot handle non-terminating programs. Real quantifier elimination has been used for program verification in [8,16,18].

9 Conclusions

In this paper we have studied well known limitations of cost analysis approaches that are based on the use of CRs. We have shown that, unlike it was assumed so far, the reason for these limitations is that CRs ignore the output values of procedures. In particular, we have shown that there are programs whose cost cannot be modeled precisely using CRs. In order to overcome these limitations, we have defined the notion of UBFs that use both input and output parameters, and developed a novel approach for cost analysis that is based on this kind of UBFs. Interestingly, we have found a relation between this kind of UBFs and potential functions that are used in automatic amortised cost analysis [15], which might give an alternative explanation to why amortised analysis (of ACR programs) can be more precise than the classical approach.

Starting from template UBFs, our analysis generates a verification condition over these templates in which the program variables are universally quantified. Eliminating these variables using quantifier elimination tools results in a (possibly non-linear constraint) whose models define possible instantiations for the templates such that they are safe UBFs. An important feature of approach is that it can be used for inferring lower-bounds (for terminating programs) with minimal changes: just replacing $\geq$ by $\leq$ in the VC, and, in addition, $\wedge$ by $\vee$ in each peak-cost condition ϕ_r. Due to lack of space we skipped the details. We have also reported on a preliminary implementation and its evaluation on small examples. For future work, we would like to find some special cases of ACR program for which the analysis can scale to large programs.

Acknowledgements This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project. Diego Alonso is supported by the UCM PhD scholarship program.

References

1. Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

2. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142–159, 2012.
3. Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *ISMM*, pages 121–130, New York, NY, USA, 2010. ACM.
4. Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In *VMCAI*, volume 6538 of *LNCS*, pages 38–53. Springer, January 2011.
5. Hugh Anderson, Siau-Cheng Khoo, Stefan Andrei, and Beatrice Luca. Calculating polynomial runtime properties. In *APLAS*, volume 3780 of *LNCS*, pages 230–246. Springer, 2005.
6. Florence Benoy and Andy King. Inferring Argument Size Relationships with CLP(R). In *LOPSTR'97*, volume 1207 of *LNCS*, pages 204–223. Springer, 1997.
7. Christopher W. Brown and Christian Gross. Efficient preprocessing methods for quantifier elimination. In *CASC*, volume 4194 of *LNCS*, pages 89–100, 2006.
8. Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. Discovering non-linear ranking functions by solving semi-algebraic systems. In *ICTAC*, volume 4711 of *LNCS*, pages 34–49. Springer, 2007.
9. Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
10. Saumya K. Debray and Nai-Wei Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.
11. Andreas Dolzmann and Thomas Sturm. REDLOG: Computer Algebra meets Computer Logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.
12. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
13. Sumit Gulwani and Florian Zuleger. The Reachability-Bound Problem. In *PLDI*, pages 292–304. ACM, 2010.
14. Timothy J. Hickey and Jacques Cohen. Automating Program Analysis. *J. ACM*, 35(1):185–220, 1988.
15. Martin Hofmann Jan Hoffmann, Klaus Aehlig. Multivariate Amortized Resource Analysis. In *POPL'11*, pages 357–370. ACM, 2011.
16. Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Deduction and Applications*, volume 05431, 2006.
17. Daniel Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.
18. David Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Computer Science*, 6(3), 2010.
19. Thomas Sturm and Ashish Tiwari. Verification and Synthesis using Real Quantifier Elimination. In *ISSAC 2011*, pages 329–336. ACM, 2011.
20. REDUCE Computer Algebra System. REDUCE home page.
21. Z3 Theorem Prover . Z3 home page.
22. Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.
23. Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.