Incremental Resource Usage Analysis

Elvira Albert and Jesús Correas
Complutense University of Madrid
{elvira, jcorreas}@fdi.ucm.es

Germán Puebla and Guillermo Román-Díez
Technical University of Madrid
{german,groman}@fi.upm.es

Abstract

The aim of *incremental* global analysis is, given a program, its analysis results and a series of changes to the program, to obtain the new analysis results as efficiently as possible and, ideally, without having to (re-)analyze fragments of code which are not affected by the changes. Incremental analysis can significantly reduce both the time and the memory requirements of analysis. This paper presents an incremental *resource usage* analysis for a sequential Java-like language. Our main contributions are (1) a *multi-domain* incremental fixed-point algorithm which can be used by all global pre-analyses required to infer the cost (including class, sharing, cyclicity, constancy, and size analyses), and which takes care of propagating dependencies among such domains, and (2) a novel form of *cost summaries* which allows us to incrementally reconstruct only those components of cost functions affected by the change. Experimental results in the COSTA system show that the proposed incremental analysis performs very efficiently in practice.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D.3 [*Programming Languages*]: [Formal Definitions and Theory]

General Terms Languages, Theory, Verification, Reliability

Keywords Static Analysis, Resource Guarantees, Incremental

1. Introduction

Cost analysis [26] (a.k.a. resource usage analysis) aims at automatically inferring the resource consumption of executing a program as a function of its input data *sizes*. In this work, we rely on a generic notion of resource, which can be instantiated to measure the amount of memory allocated, number of instructions executed, number of calls to methods, etc. Intuitively, the main steps in order to infer the cost of programs written in an object-oriented (OO) language are:

1. *OO Pre-analyses*. Almost for every property being analyzed, it is required to perform a global *class* analysis [23] which determines the set of reachable classes which must be considered by the subsequent steps. Besides, analyzers often perform non-nullness analysis which allows removing unfeasible nullity checks, and some form of *field-sensitive* analysis [1, 17] which allows reasoning on the values (or the shape of) data stored in the global memory (or heap).

2. *Cost relations*. Given the program and the pre-analyses information, this step consists in setting up cost *recurrence equations*, or cost relations for short (CRs), which define the cost of executing the program in terms of the input data sizes. The global analysis underlying this step is the inference of size relations which determine how the sizes of data change along program's execution [2]. In the presence of heap-allocated data structures, size analysis based on path-length [22] relies on a series of pre-analyses, namely, *sharing*, *acyclicity* and *constancy*.

3. *Cost functions*. In the last step, cost analyzers [5] try to solve CRs and obtain *cost functions* which are not in recursive form and hence are directly evaluable. Since a precise solution seldom exists, analyzers infer upper/lower bounds for the CRs. This is again a global process which starts by solving the CRs which do not depend on any other one and continues by replacing the computed cost functions on the equations which call such relations untill all CRs are solved.

Hence, cost is inferred by a sequence of *global* analyses (or whole-program analyses) which require to analyze the whole program in order to obtain sound and precise results. Despite the great progress made in static analysis, most global analyzers still read and analyze the entire program at once in a non-incremental way. In particular, all resource analyses to date are non-incremental [2, 11, 14]. During software development, programs are often modified, e.g., because a new implementation of an existing method is provided (which improves its efficiency or fixes its correctness) or because an existing code is extended with new functionality (typically by extending a class with further methods). In such cases, the existing analysis information for the program may no longer be correct and/or accurate. However, resource analysis is a costly task and starting analysis from scratch is inefficient in most cases. A key challenge for static analysis techniques is achieving a satisfactory combination of precision and scalability. Making precise (and hence expensive) static analysis incremental is a step forward in this direction.

In this paper, we present an incremental extension of resource usage analysis of an imperative and object-oriented programming language. The difficulty when devising an incremental analysis framework is to recompute the least possible information and do it in the most efficient way. In our setting, we achieve it by means of the following two steps wich are our main contributions:

- A *multi-domain* incremental analysis engine which can be used by all global pre-analyses required to infer the resource usage of a program (including the class analysis, sharing, cyclicity, constancy and size analysis as mentioned above). The algorithm is multi-domain in the sense that it interleaves the computation for the different domains and takes into account dependencies among them, in such a way that it is possible to invalidate only partial pre-computed information.
- Even a small change within a method (e.g., adding an instruction) can change the overall cost of the program. A fundamental

idea to minimize the amount of information which needs to be recomputed is to be able to distinguish within a *cost summary* the cost subcomponent associated to each method, so that the final cost functions can be recomputed by replacing only the affected subcomponents.

Our incremental analysis has been implemented in the COSTA system, a cost and termination analyzer for Java bytecode programs. Experimental results are performed on selected benchmarks from the standardized *JOlden* benchmark suite [24] and from the *Apache Commons* Project [18]. Our results show that the proposed incremental analysis achieves a significant speedup with respect to the non-incremental approach. To the best of our knowledge, this is the first incremental resource usage analysis framework.

2. Global Analysis of OO Languages

To formalize the analysis algorithms, we consider a *rule-based* intermediate representation of a today's programming language (e.g., Java, Java bytecode [2], X10 [3], Timber [15] can be compiled into this language. We then present a global analysis engine for such rule-based representation (RBR) which is parametric w.r.t. the analysis domain and that will later be extended to support incremental analysis.

2.1 The Rule-Based Language

A *program* in the RBR consists of a set of *procedure*s which are defined as a set of (recursive) rules. A procedure p with input arguments $\bar{x}$ and output arguments $\bar{y}$ is defined by a set of *guarded rules* which adhere to the following grammar:

$$\begin{array}{rcl}
rule & ::= & p(\bar{x},\bar{y}) \leftarrow g, b_1, \ldots, b_n \\
g & ::= & true \mid exp_1\ op\ exp_2 \mid \mathsf{type}(x, C) \\
exp & ::= & x \mid \mathsf{null} \mid n \mid x{-}y \mid x{+}y \mid x{*}y \\
op & ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq \\
b & ::= & x{:=}exp \mid x{:=}\mathsf{new}\ c \mid x{:=}y.f \mid x.f{:=}y \mid q(\bar{x},\bar{y})
\end{array}$$

where $p(\bar{x},\bar{y})$ is the *head* of the rule; $\bar{x}$ (resp. $\bar{y}$) are the input (resp. output) parameters; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\bar{x},\bar{y})$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, C)$, which succeeds if the runtime class of x is exactly C. A class C is a finite set of typed field names, where the type can be integer or a class name. The key features of this representation which simplify later the formalization of the analysis are: (1) input and output parameters are explicit variables of rules, (2) *recursion* is the only iterative mechanism, (3) *guards* are the only form of conditional and (4) objects can be regarded as records, and the behavior induced by dynamic dispatch is compiled into *dispatch* rules guarded by a type check.

A method m in a Java (bytecode) program is represented by a set of procedures in the RBR such that there is an entry procedure named m and the remaining ones are intermediate procedures invoked only within m. We use function *is_method(m)* to check if m is the entry procedure of a method. The translation of a program into the RBR works by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the RBR as a rule. As customary a single static assignment transformation is applied on the rules variables. The process is identical as [2, 3, 15], hence, we will not go into the technical details of the transformation but just show the intuition by means of an example.

EXAMPLE 2.1 (RBR). Our running example in Fig. 1 is the simplest one we could find to show the interaction among the different domains and the reanalysis required by the incremental extensions.

```
class C {
  void main (List l) {
    int s = len(l);
    if (s % 2 != 0 || s < 2)
      return;
    mod(l);
  }
  void mod (List l) {
    Inc o = get(l);
    dup(o, o.incr(l));
  }
  Inc get (List l) {
    return new Inc();
  }
  void dup (Inc o, List l) {
    while (l != null) {
      l.data = l.data * 2;
      l = o.incr(l);
    }
  }
  int len(List l) {
    int i = 0;
    for(; l != null; l = l.next) {
      i++;
    }
    return i;
  }
}
class Inc {
  List incr (List l) { return l.next.next; }
}
class Inc2 extends Inc{
  List incr (List l) { return l.next; }
}
class List {
  List next;
  int data;
}
```

Figure 1. Running Example

Method `main` receives a list of integers, checks the length of the list and modifies some of its elements by invoking `mod`. The following RBRs for some methods of the example illustrate all features of the intermediate representation mentioned above:

$$\begin{array}{rcl}
mod(\langle l\rangle, \langle\rangle) & \leftarrow & get(\langle l\rangle, \langle o\rangle), \\
 & & call\_incr(\langle o, l\rangle, \langle l'\rangle), \\
 & & dup(\langle o, l'\rangle, \langle\rangle) \\
get(\langle l\rangle, \langle r\rangle) & \leftarrow & r := new\ \mathtt{Inc} \\
dup(\langle o, l\rangle, \langle\rangle) & \leftarrow & while(\langle o, l\rangle, \langle l'\rangle) \\
while(\langle o, l\rangle, \langle l\rangle) & \leftarrow & \boldsymbol{l = \mathtt{null}} \\
while(\langle o, l\rangle, \langle l'''\rangle) & \leftarrow & \boldsymbol{l \neq \mathtt{null}}, \\
 & & l'.data := l.data * 2, \\
 & & call\_incr(\langle o, l'\rangle, \langle l''\rangle), \\
 & & while(\langle o, l''\rangle, \langle l'''\rangle) \\
call\_incr(\langle o, l\rangle, \langle l'\rangle) & \leftarrow & \mathbf{type}(\boldsymbol{o}, \mathtt{Inc}), \\
 & & Inc.incr(\langle l\rangle, \langle l'\rangle) \\
Inc.incr(\langle l\rangle, \langle l''\rangle) & \leftarrow & l' := l.next, \\
 & & l'' := l'.next
\end{array}$$

The RBR of the program is built along the execution of class analysis. Virtual invocations are statically resolved and simulated by means of *dispatch rules* (e.g., *call_incr*). Class analysis for this program determines that `o` is an instance of `Inc`, hence the corresponding dispatch rules do not include a call to `Inc2.incr`.

2.2 A Global Fixed-Point Analysis Engine

We assume the reader is familiar with *abstract interpretation* [6]. Algorithm 1 presents an event-based global fixed-point analysis engine for the RBR. The algorithm is parametric w.r.t. the abstract do-

```
 1: proc analysis($m$,$CP$,$D$)
 2:    $\mathcal{Q} = \emptyset$; $\mathcal{L}$=$\emptyset$; $\mathcal{Q}$.add($m$,$CP$);
 3:    while ($\neg\mathcal{Q}$.empty()) do
 4:       ($p$,$CP$)=$\mathcal{Q}$.extract_first()
 5:       process_analysis($p$,$CP$,$D$)

 6: function get_proc_answer($p$,$CP$,$D$)
 7:    $CP' = CP$
 8:    if ($\mathcal{L}$.exists($p$)) then
 9:       ($CP^{\mathcal{L}} \mapsto AP^{\mathcal{L}}$)= $\mathcal{L}$.get($p$)
10:       if ($CP \sqsubseteq CP^{\mathcal{L}}$) then
11:          return $AP^{\mathcal{L}}$
12:       $CP' = CP \sqcup CP^{\mathcal{L}}$
13:    $\mathcal{Q}$.add($p$,$CP'$)
14:    return $\bot$

15: proc invalidate_callers($q$)
16:    for all ($p'$ in callers($q$)) do
17:       if ($\mathcal{L}$.exists($p'$)) then
18:          ($CP' \mapsto AP'$)=$\mathcal{L}$.get($p'$)
19:          $\mathcal{L}$.remove($p'$); $\mathcal{Q}$.add($p'$,$CP'$)

20: proc process_analysis($p$,$CP$,$D$)
21:    $AP$=$\bot$
22:    for all ($R_i : p \leftarrow b_{i1}, \ldots, b_{in}$) do
23:       $ST$=extend($CP$,$vars(R_i)$,$D$)
24:       for each ($b_{ij}, 1 \leq j \leq n$) do
25:          if ($b_{ij} = q(\_,\_)$) then
26:             $CP'$=restrict($ST$,$vars(b_{ij})$,$D$)
27:             $ST'$=get_proc_answer($b_{ij}$,$CP'$,$D$)
28:             $ST'$=extend($ST'$,$vars(R_i)$,$D$)
29:             if ($ST' \not\sqsubseteq AP$) then
30:                invalidate_callers($b_{ij}$)
31:          else
32:             $ST'$=extend(alpha($b_{ij}$,$D$),$vars(R_i)$,$D$)
33:          $ST$= $ST \sqcap ST'$
34:       $AP = AP \sqcup ST$
35:    $\mathcal{L}$.update($CP \mapsto AP$)
```

Algorithm 1: Fixed-point algorithm (operators $\sqcup$, $\sqsubseteq$, $\sqcap$ are parametric w.r.t. the analysis domain, D)

main D that describes some property of interest. As usual, the domain D is defined for all possible descriptions, which form a complete lattice for which all ascending chains are finite. We assume that the operations of *least upper bound* (denoted $\sqcup$), *greatest lower bound* (denoted $\sqcap$) and $\sqsubseteq$ are defined in some precise sense that depends on the particular abstract domain. Function alpha returns the abstraction of an instruction in D. Function restrict(ST, V, D) projects an abstraction ST to the variables in the set V w.r.t. domain D, and extend(ST', V', D) extends the abstraction ST' to the variables in V' w.r.t. domain D. The analysis results of a procedure are computed with respect to a specific calling pattern CP to it, which is a description in the abstract domain. The analysis is *monovariant*, i.e., the goal of the analysis is to compute for each procedure p in the program at most one *answer* of the form $CP \mapsto AP$, where AP is the *answer pattern*, which is also description in the abstract domain, and the *call pattern* CP is general enough to cover all possible patterns for p that appear during the analysis of the program.

The algorithm uses two global data structures: (1) the *local answer table* $\mathcal{L}$ where the answers for all procedures are stored and (2) the *queue of events* $\mathcal{Q}$, which initially contains as single element the pair (m, CP) with the entry procedure m and a corresponding call pattern CP. The analysis of a method is carried out in *process_analysis*, where we analyze all rules defining a procedure in Line 22 (L22 for short) and traverse the instructions in its body from left to right (L24). When the instruction is not a procedure call, we abstract it according to the abstract domain (L32). As usual, the abstract description obtained from one instruction is conjoined with the previously computed one (L33). The analysis results obtained from the different rules which define a procedure are joined together (L34) and stored in the local answer table $\mathcal{L}$ (L35). When the instruction is a procedure call (L25), *get_proc_answer* first checks if a previously computed answer exists in $\mathcal{L}$ (L8). We assume that *get* returns the answer for a given call properly renamed w.r.t. the arguments in the call (L9). If the existing calling pattern is general enough (L10), we just use the previous answer (L11). Otherwise, since the algorithm is monovariant, we join the calling patterns (L12) and reanalyze the corresponding method (L13). Upon return from *get_proc_answer*, if the answer for q has changed (L29), we need to invalidate the information for all methods that invoke q, denoted *callers(q)* (L15-19).

Observe that the *analysis* engine adds entries to $\mathcal{Q}$ during its execution when (i) a method must be analyzed for a given calling pattern in L13, either because there was not answer for it or because it had been analyzed for a less general calling pattern and (ii) when the answer of a method invoked from it changes. The execution finishes when there are not more events to process in $\mathcal{Q}$ (L3).

The fact that the language is rule-based makes the design of the analysis simpler and similar to existing analysis drivers developed for constraint logic programming (CLP) [13]. A main difference between analyzing CLP and imperative programs is that we need to treat global data. Interestingly, Alg. 1 can handle global data by using the field-sensitive extension proposed in [1] without requiring any change in the algorithm. In particular, this extension allows treating object fields that meet certain locality conditions as local variables. Therefore, with such pre-processing, Alg. 1 becomes directly a field-sensitive analysis algorithm.

3. Incremental Inference of Cost Relations

The goal of this section is to support incremental analysis in all global pre-analyses required to infer cost relations (CRs) which corresponds to item (2) in Section 1. First, Sec. 3.1 describes the notion of *method summary* which comprises the analysis information that has been computed globally in a non-incremental way in order to set up CRs. Sec. 3.2 introduces a multi-domain incremental analysis which, given a change and the method summaries, is able to reconstruct the summaries for all domains.

3.1 Method Summary for Global Properties

All analysis information included within a method summary can be computed by using the generic fixed-point engine in Alg. 1 for each corresponding domain, one after another in the top down order established in Fig. 2, where the dependencies between the domains are shown. Given a domain D, we will refer to the set of domains reachable from D in the dependency graph (including D) as *dep(D)*.

DEFINITION 3.1 (method summary). *Given a method $m(\bar{x}, \bar{y})$, a* method summary *for m is a tuple of five answers $CP_D \mapsto AP_D$ for the following domains:*

(1) D=class, *where x:$\{C_1, \ldots, C_n\} \in AP_{cl}$ (resp. CP_{cl}) represents the set of classes that variable x may be typed to after (resp. before) executing m [23];*
(2) D=sharing, *where $(x, y) \in AP_{sh}$ (resp. CP_{sh}) means that x and y might share after (resp. before) executing m [21];*
(3) D=acyclicity, *where $x \in AP_{ac}$ (resp. CP_{ac}) means that x may point to a cyclic data structure after (resp. before) executing m [20];*
(4) D=constancy, *where $x \in AP_{cn}$ if the structure of x may have changed during the execution of m [10] (this analysis is context-insensitive);*

(5) D=size*, where* AP_{sz} *(resp.* CP_{sz}*) are linear constraints describing the relation between the size of* $\bar{x}$ *and* $\bar{y}$ *after (resp. before) executing* m *[3].*

Let us mention the most relevant issues related to the five points in the above definition and which explain the domain dependencies in Fig. 2. (1) The class analysis information determines the overall code that must be analyzed in the next steps. This domain is finite since it only contains the classes available in the program at run-time. (2) In size analysis, we assume that the size of a heap-allocated data structure is its path-length (i.e., the length of the longest path reachable from it), if the data structure is not *cyclic*. Hence, acyclicity is a soundness requirement for size analysis. (3) Besides, if two variables x and y share and there is a reference field assignment $x.f = y$, then no safe information can be provided regarding the acyclicity of x nor y since cycles might be introduced. Hence, sharing is a soundness requirement for acyclicity and hence for size. (4) It is essential to know the arguments of m whose shape remains *constant* upon return because in such case their path-length is preserved on exit from m [10]. (5) Once all previous analyses have been performed, size relations can be inferred in order to determine how the size of data is modified along the program's execution.

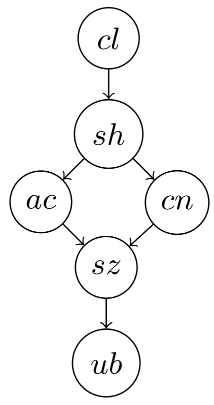


Figure 2. Domains dependencies

EXAMPLE 3.2 (summaries and CRs). The following method summaries are obtained by sequentially performing each of the global analyses above in a top down order (the results of sharing, acyclicity and constancy are not shown as they do not detect any sharing between data structures, cycles, nor changes in the data structures shapes, resp.). Note that *r* stands for the return value:

Method		Summaries
`void mod(l)`	*cl*	$\{l{:}\{\texttt{List}\}\} \mapsto \bot$
	sz	$\{l \geq 2\} \mapsto \{l \geq 2\}$
`Inc get(l)`	*cl*	$\{l{:}\{\texttt{List}\}\} \mapsto \{r{:}\{\texttt{Inc}\}\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq 2, r = 1\}$
`void dup(o,l)`	*cl*	$\{o{:}\{\texttt{Inc}\}, l{:}\{\texttt{List}\}\} \mapsto \bot$
	sz	$\{o = 1, l \geq 0\} \mapsto \{o = 1, l \geq 0\}$
`List incr(l)`	*cl*	$\{l{:}\{\texttt{List}\}\} \mapsto \{r{:}\{\texttt{List}\}\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq r + 2, r \geq 0\}$

Observe that after creating the object in `get`, the type of the object is instantiated to `Inc` and its size is 1. Also, notice that, after analyzing `len`, the **size** analysis of `main` learns from the conditional statement that the size of the list is greater than or equal to two. This size constraint is used as precondition of the next methods. It is important to understand the size relation obtained for method `incr` which allows us to know that the size of the output list is the size of the input list decreased by two. This piece of information is essential to bound the cost.

3.2 A Multi-Domain Incremental Fixed-Point Analyzer

When performing incremental analysis, after a modification in a program, a change in the analysis results associated to one domain must *invalidate* the results previously inferred by subsequent dependent domains. Algorithm 2 presents the extensions

```
 1: proc incremental_fixpoint($m$)
 2:   $\mathcal{P}$ =[($m$,$\mathcal{D}$)]
 3:   $\mathcal{G}$.invalidate($m$,$\mathcal{D}$)
 4:   while ($\neg\mathcal{P}$.empty()) do
 5:     $\mathcal{R}$=$\emptyset$
 6:     ($m'$,$\mathcal{D}_{m'}$) =$\mathcal{P}$.extract_first()
 7:     for each $D$ in $\mathcal{D}_{m'}$ do
 8:       ($CP^{\mathcal{G}}_{m'} \mapsto AP^{\mathcal{G}}_{m'}$)=$\mathcal{G}$.get($m'$,$D$)
 9:       noincr:analysis($m'$,$CP^{\mathcal{G}}_{m'}$,$D$)
10:     for all (($n$,$D$) in $\mathcal{R}$) do
11:       ($CP_n \mapsto AP_n$)=$\mathcal{G}$.get($n$,$D$)
12:       ($CP'_n \mapsto AP'_n$)=$\mathcal{L}$.get($n$,$D$)
13:       if ($AP'_n \not\sqsubseteq AP_n$)) then
14:         $\mathcal{P}$.add_dom(callers($n$),dep($D$))
15:       $\mathcal{G}$.update($CP'_n \mapsto AP'_n \sqcup AP_n$)
16:     $\mathcal{P}$.remove($\mathcal{R}$)
17: function get_proc_answer($p$,$CP$,$D$)
18:   $CP'$=$CP$
19:   if (is_method($p$) $\wedge$ $\mathcal{G}$.exists($p$,$D$)) then
20:     ($CP^{\mathcal{G}} \mapsto AP^{\mathcal{G}}$)=$\mathcal{G}$.get($p$,$D$)
21:     if ($\mathcal{G}$.valid($p$,$D$) $\wedge$ $CP \sqsubseteq CP^{\mathcal{G}}$) then
22:       return $AP^{\mathcal{G}}$
23:     else
24:       $CP'$=$CP \sqcup CP^{\mathcal{G}}$
25:       $\mathcal{G}$.invalidate($p$,dep($D$))
26:       $\mathcal{P}$.add_dom({$p$},dep($D$))
27:       $\mathcal{R}$.add(($p$,$D$))
28:   $AP$=noincr:get_proc_answer($p$,$CP'$,$D$)
29:   return $AP$
```

Algorithm 2: Generic incremental fixed-point algorithm.

required to make Alg. 1 incremental by relying on the summaries in Def. 3.1 and the dependencies in Fig. 2. We use the notation *noincr:m* to refer a procedure *m* defined in Alg. 1 (see L9 and L28 in Alg. 2). The incremental algorithm uses method *noincr:analysis* of Alg. 1 (and its data structures) and two implementations of *get_proc_answer*, *noincr:get_proc_answer* and *incr:get_proc_answer*. In L27 of *noincr:process_analysis* of Alg. 1, we invoke *incr:get_proc_answer* instead of *noincr:get_proc_answer*.

In contrast to other approaches [13], the granularity of our analysis is set at the level of methods, i.e., we establish the method as the smallest piece of code whose analysis information will be stored and reanalyzed in case of changes. Procedure *incremental_fixpoint* receives the signature of the method m which has been changed. If there are multiple methods changed, changes will be handled one after another. Note that class analysis determines the CFG of the program being analyzed. Hence, after a change, the callers information used in L14 by Alg. 2 is recomputed. Observe that the objective of Alg. 2 is to recompute invalidated information, but not to improve the precision of previously computed results. Trying to improve the precision would require further recomputation in some cases. The three main aspects of the algorithm are described as follows.

Multi-domain. Our aim is to handle multiple domains in the context of incremental analysis such that the minimal amount of reanalysis is performed. Our approach consists in interleaving the computation of the incremental fixed point for all domains by means of validity flags. The idea is that a change in a summary for a specific domain invalidates only the entries for those dependent domains according to their dependencies (in our case those in Fig. 2). This is handled in the algorithm by means of a *global answer table* $\mathcal{G}$ which contains the set of summaries for all previously analyzed methods such that each entry stored in $\mathcal{G}$ has a flag to indicate whether the entry for this particular domain is valid. Initially all entries are valid. We use function *valid(p,D)* to check if the summary for p and domain D is valid. The call *invalidate(p,$\mathcal{D}$)* sets up to invalid the flags for the set of domains $\mathcal{D}$ for entry p.

Descendants. When a method changes, its code must be reanalyzed with respect to all domains (L3). L5-9 take care of reanalyzing the descendants. In particular, the call *noincr:analysis* (L9) reanalyzes m for a particular domain as well as all those methods reachable from m (descendants) whose answer for such domain must be recomputed. Observe that the new function *get_proc_answer* differs from the one in Alg. 1 in that, for method calls, it tries to reuse an existing answer from the method summary (L19) if its calling pattern is general enough and the entry has not been invalidated (L21). Otherwise, both calling patterns are joined (L24) and the pair of method signature and domain is added to the list $\mathcal{R}$ (which contains the list of methods to be reanalyzed for specific domains). Only those dependent entries of the method summary are invalidated (L25). Note that the call to *noincr:invalidate_callers* only invalidates methods which are descendants of m. If there is no summary for the method, or if it is an intermediate procedure of a method, the function *noincr:get_proc_answer* of Alg. 1 is invoked, and it analyzes m in the non-incremental way (L28).

Ancestors. When the call *noincr:analysis* finishes (L9), a series of pairs have been added to the list $\mathcal{R}$. Now, we need to take care of reanalyzing all those methods that relied on answers for methods in $\mathcal{R}$. The list $\mathcal{P}$ is used for this purpose. $\mathcal{P}$ is formed by pairs of the form $(m, \mathcal{D})$ where $\mathcal{D}$ is a list of domains for which m must be reanalyzed ordered according to Fig. 2. Initially, $\mathcal{P}$ contained an entry for the changed method and all domains (L2). It is later updated in L14 as follows. For each element in $\mathcal{R}$, we know that its summary for such domain has changed (L10). We compare the new answer AP'_n with the one in the summary AP_n (L11-12). If the new one is not contained in the previous one (L13), we need to reanalyze all methods that invoke m (L14) for such domain and its dependent domains. We assume that *add_dom* receives a set of methods $\mathcal{M}$ and a set of domains $\mathcal{D}$ and, for each $m \in \mathcal{M}$, $D \in \mathcal{D}$, if there exists an entry $(m, \mathcal{D}_m)$, it is updated to $(m, add(\mathcal{D}_m, D))$, where the addition of D is ordered according to Fig. 2. If it does not exist, it adds $(m, \{D\})$. However, some methods that invoke m may have been already reanalyzed in this iteration and, therefore, they do not need to be reanalyzed again. They are removed from $\mathcal{P}$ in L16. Finally, the fixed point is reached when there are no more methods to analyze in $\mathcal{P}$.

EXAMPLE 3.3 (alg 2). Let us consider this new implementation of method `get`:

```
Inc get(List l) {
  if ((l.data % 2)==0)
    return new Inc();
  else
    return new Inc2();
}
```

The following iterations of Alg. 2 are performed due to this change:

(1) *analysis(*`get`*,_,D)* is executed for all domains. The class analysis info for `get` stored in $\mathcal{G}$ does not contain the newly generated answer pattern $\{r{:}\{\texttt{Inc}, \texttt{Inc2}\}\}$. Therefore, all direct ancestors of `get` must be reanalyzed, namely `mod` is added to $\mathcal{P}$ for *dep(class)*.

(2) *analysis(*`mod`*,_,D)* is launched for *dep(class)*. During class analysis, a call to `dup` is found. The entry for `dup` in $\mathcal{G}$ has $CP_1 \equiv \{o{:}\{\texttt{Inc}\}, l{:}\{\texttt{List}\}\}$ and thus it is not applicable to $CP_2 \equiv \{o{:}\{\texttt{Inc}, \texttt{Inc2}\}, l{:}\{\texttt{List}\}\}$. Both *CP*s are joined (resulting in CP_2) and `dup` is reanalyzed for the class domain w.r.t. CP_2. Entries for `dup` for *dep(class)* in $\mathcal{G}$ are invalidated and added to $\mathcal{P}$. Due to polymorphism, during the analysis of `dup`, an invocation to `Inc2.incr`[1] is found. Since there are no entries for `Inc2.incr`, it is analyzed for all domains. This forces that the size relations for procedure *while* change. Now, we must join the analysis results of `Inc.incr` and `Inc2.incr`, since any of the two methods can be executed within the loop.

At the end of (2) a fixed-point is reached since there are no further changes in the answer pattern for any domain. Importantly, only affected methods have been reanalyzed and their information in $\mathcal{G}$ is up-to-date. Methods `main`, `len` and `Inc.incr` have not required reanalysis for any domain. The following table shows those summaries that have changed w.r.t. the ones in Ex. 3.2:

Method	Summaries	
`Inc get(l)`	*cl*	$\{l{:}\{\texttt{List}\}\} \mapsto \{r{:}\{\texttt{Inc}, \boldsymbol{Inc2}\}\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq 2, r = 1\}$
`void dup(o,l)`	*cl*	$\{o{:}\{\texttt{Inc}, \boldsymbol{Inc2}\}, l{:}\{\texttt{List}\}\} \mapsto \perp$
	sz	$\{o = 1, l \geq 0\} \mapsto \{o = 1, l \geq 0\}$
`List Inc2.incr(l)`	***cl***	$\{l{:}\{\texttt{List}\}\} \mapsto \{r{:}\{\texttt{List}\}\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq r + 1, r \geq 0\}$

THEOREM 3.4 (correctness). *Given a program P, its analysis results stored in $\mathcal{G}$, and a method m which has been modified,* incremental_fixpoint(m) *terminates and returns a correct $\mathcal{G}$ for all methods in P.*

4. Generation of Cost Relations

Given the size relations, the second phase of cost analysis (item 2 of Sec. 1) is the generation of CRs. This step is performed locally to each rule and hence it is already "incremental" (or local). Nevertheless, in order to link with the next phase of cost analysis (in Sec. 5), we need to explain what CRs are and how the incremental analysis has to treat them. Intuitively, the generation of an equation for a rule consists of the next steps: (1) apply the selected cost model to each instruction in the rule (a cost model maps an instruction into its corresponding cost), (2) abstract each basic instruction by a size constraint and, (3) when we find a call to a method, the size constraint is the size relation in the summary above and the cost of the call is defined by a corresponding equation. Output variables of methods do not appear in the equations, as the cost is given in terms of the input arguments only.

EXAMPLE 4.1. Consider the original program before the change in Ex. 2.1. By using the cost model that counts number of instructions, the first rule of Ex. 2.1 is transformed into the equation: $mod(l) = 3 + get(l) + call\_incr(l) + dup(l')\ \{l \geq l' + 2, l' \geq 0\}$ where the 3 stands for the three calls in the instruction. The cost of the calls to the methods will be defined by corresponding equations. For the running example, we get:

$$\begin{array}{lr}
mod(l) = 3 + get(l) + call\_incr(l') + dup(l') & \{l \geq l' + 2\} \\
get(l) = 1 & \{\} \\
while(l) = 3 + call\_incr(l) + while(l') & \{l' \geq 0, l \geq l' + 2\} \\
dup(l) = 1 + while(l) & \{l \geq 0\} \\
call\_incr(l) = 0 + Inc.incr(l) & \{\} \\
while(l) = 0 & \{l = 0\} \\
Inc.incr(l) = 2 & \{l \geq 2\}
\end{array}$$

The resulting constraints to the right define the applicability conditions of the equations and the size relations between the variables. We omit in the equations the variables that are not involved in the equation guards and because they are useless for solving the equations.

[1] Method references include the class they belong to when disambiguation is needed.

After a modification in a program, when Alg. 2 finishes, we need to generate new CRs for all methods which have been reanalyzed (i.e., those that have belonged to $\mathcal{P}$). This is because their size relations may have changed; and, besides, for the changed method its accumulated cost can change as well.

EXAMPLE 4.2. Consider now the modification in Ex. 3.3. When Alg 2 finishes $\mathcal{P}$ contains {`get`, `mod`, `dup`, `Incr2.inc`, `get`}. Thus, the following CRs of *while* must be generated by using the standard CR generation as explained above:

$$\begin{array}{ll}
while(l) = 3 + call\_incr(l) + while(l') & \{l' \geq 0, l \geq l' + 1\} \\
call\_incr(l) = 0 + Inc.incr(l) & \{\} \\
Inc.incr(l) = 2 & \{l \geq 2\} \\
call\_incr(l) = 0 + Inc2.incr(l) & \{\} \\
Inc2.incr(l) = 1 & \{l \geq 1\}
\end{array}$$

When comparing the equations with the ones in Ex. 4.1, we observe that after merging the size information gathered for the two implementations of `incr`, in the new CR, we lose information and have to assume that the length of the list decreases (in the worst-case) by one.

5. Incremental Inference of Upper Bounds

The third phase in cost analysis (see item 3 in Sec. 1) consists in transforming the CRs obtained in Sec. 3 into *cost functions* [5], i.e., cost expressions without recurrences. Since a precise solution often does not exist, cost analyzers infer upper bounds/lower bounds (UBs/LBs) from them which are, resp., over/under-approximations of the worst/best-case cost. For the sake of concreteness, we focus on UBs (the problem of LBs is dual). In Sec. 5.1, we first introduce the notion of UB summary which specifies the information which needs to be stored in order to recompute UBs after a change in a program (and hence in its CR). Sec. 5.2 presents an algorithm to support incremental inference in this step.

5.1 The Notion of Cost Summary

Our starting point is the technique of [5] which proposes an automatic approach to obtaining UBs from CRs by (1) first, transforming all relations into direct recursion (this process leaves one relation per SCC) and (2) then, obtaining an UB for the standalone CRs (which do not call any other relation) and consecutively replacing such UBs in the equations which call such relations until all CRs are solved. W.l.o.g., we consider polynomial CRs defined by a set of equations, each of them containing at most one recursive call:

$$\langle C(\overline{x}) = \texttt{exp} + \sum_{i=1}^{k} D_i(\overline{y}_i) + C(\overline{y}), \varphi \rangle \qquad (1)$$

In [5], automatic techniques to solve C (as well exponential and logarithmic relations) are proposed. The following definition summarizes such solving process:

DEFINITION 5.1 (upper bound [5]). *An upper bound for $C(\overline{x})$ is* $\text{UB}_C(\overline{x}) = \#iter * \texttt{mexp}$ *(base cases are ignored for simplicity) where:*

1. *$\#iter$ is an upper bound on the number of recursive calls of C,*
2. *the* size relations *(φ) are linear constraints on variables $\overline{x}$ and $\overline{y}_i \cup \overline{y}$,*
3. *the* invariant *(ψ) relates variables $\overline{y}_i \cup \overline{y}$ to their initial values $\overline{x}$ (we denote the initial value of a variable x as x_0),*
4. *$\texttt{ub}_i$ are upper bounds for each call $D_i(\overline{y}_i)$,*
5. *for each recursive equation, we have that:*
   $\texttt{mexp}'$=*maximize*($\texttt{exp}$,$\overline{x}$,ψ,φ)+ $\sum_{i=1}^{k}$ *maximize*$(\texttt{ub}_i, \overline{x}, \psi, \varphi)$
   where function maximize$(e, \overline{x}, \psi, \varphi)$ returns the maximization of e for ψ and φ w.r.t. the equation entry variables $\overline{x}$,
6. *if C has k recursive equations,* $\texttt{mexp}$=max($\texttt{mexp}'_1$, ..., $\texttt{mexp}'_k$), *where $\texttt{mexp}'_j$ is the maximized cost of equation j obtained in point 5, with $j = 1, \ldots, k$.*

Observe that the process of obtaining `mexp` requires an invariant generation phase and a maximization of expressions. We cannot make any incrementalization of these two parts because they are already locally obtained from the CRs (see the details in [5]).

EXAMPLE 5.2 (ub). Let us apply the above definition to solve the CR `mod` in Ex. 4.1 which has obtained before applying the modification in Ex. 3.3. The standalone CRs `Inc.incr` and `get` are already solved since they are not recursive. Next, we solve *while*(l') which is required to solve `dup`. The following invariant holds for this CR $\psi = \{l'_0 \geq l' + 2\}$, which simply stats that the length of the list in the recursive call is strictly smaller than the length of the initial list. An UB of $\#iter$ for while is $\mathsf{nat}(l'/2)$. Function $\mathsf{nat}(v) = \max(\{v, 0\})$ is used by the UB solver to avoid negative evaluations. By applying Def. 5.1, $\text{UB}_{\texttt{dup}}(l')=1+(3+2)*\mathsf{nat}(l'/2)$. Finally, when computing an UB for `mod`(l), the UB for `dup` has to be maximized w.r.t. the entry variable l, $\varphi = \{l \geq l'+2, l' \geq 0\}$ and $\psi = \{l_0 = l, l_0 \geq 2\}$. This results in $\text{UB}_{\texttt{mod}}(l)=6+(1+5*\mathsf{nat}(l/2-1))$, where we can observe that the maximum cost of `dup` within `mod` occurs when $l' = l - 2$.

In the above example, it can be seen that an UB is a global expression which includes the UBs of the relations it calls. If the CR associated to one method m changes, since it is not possible to distinguish within an UB which part of the cost is associated to m, the whole expression must be recomputed. This affects the UBs of all methods from which m is reachable and often forces recomputation of all cost functions upwards in the program call graph until reaching the `main` method. A fundamental idea to support incremental inference of UBs is to *annotate* each cost subexpression with the name of the relation it comes from. If additionally we keep the invariants and the size relations, given an annotated UB for a method m, it is possible to replace the cost subexpressions associated to those methods invoked from m whose UB has changed by the new (maximized) UBs, without having to recompute the whole UB for m. Thus, instead of using the UBs in Def. 5.1, we use the notion of *UB summary*.

DEFINITION 5.3 (upper bound summary). *In the same conditions of Def. 5.1, an* UB summary *for $C(\overline{x})$ is a tuple* $\text{UB}^{\mathsf{S}}_{\mathsf{C}}(\overline{x}) = \langle \#iter \cdot \texttt{aexp}, \psi, \varphi \rangle$, *where* $\texttt{aexp} = maximize(\texttt{exp}, \psi, \varphi, \overline{x}) + \sum_{i=1}^{k} [maximize(\mathsf{remove\_annot}(\texttt{aub}_i), \psi, \varphi, \overline{x})]_{D_i}$ *s.t.:*

- $\texttt{aub}_i$ *is the annotated cost expression in the upper bound summary of D_i,*
- *function* $\mathsf{remove\_annot}$ *removes the annotations of an expression, and*
- *$[e]_D$ is an annotation of e with the name of the relation D it originates.*

The notation `aexp`.*set_expr*($[e]_m$,*exp*) *is used to rewrite in* `aexp` *the annotated subexpression e with exp keeping the same annotations.*

An important observation in the above definition is that the annotations refer only to direct calls from the relations.

EXAMPLE 5.4 (UB summary). UB summaries for some selected CRs are:

$$\begin{array}{rl}
UB_{\texttt{while}}(l) = & \langle (3 + [2]_{\texttt{Inc.incr}(l')}) * \mathsf{nat}(l/2), \\
& \{l_0 \geq l + 2, l \geq 0\}, \\
& \{l = l'\} \rangle
\end{array}$$

$$\begin{aligned}
UB^{\mathsf{S}}_{\texttt{dup}}(l) = \langle & 1 + [(3+2) * \mathsf{nat}(l/2)]_{\texttt{while}(l')}, \\
& \{l_0 = l, l_0 \geq 0\}, \\
& \{l = l', l \geq 0\}\rangle \\
UB^{\mathsf{S}}_{\texttt{mod}}(l) = \langle & 3+[1]_{\texttt{get}(l')}+[2]_{\texttt{incr}(l'')}+[(1+5*\mathsf{nat}(l/2-1))]_{\texttt{dup}(l''')}, \\
& \{l_0 = l, l_0 \geq 2\}, \\
& \{l = l', l = l'', l \geq l''' + 2, l''' \geq 0\}\rangle \\
UB^{\mathsf{S}}_{\texttt{main}}(l) = \langle & 6+[2+3 * \mathsf{nat}(l)]_{\texttt{len}(l')}+[7+5*\mathsf{nat}(l/2-1)]_{\texttt{mod}(l'')}, \\
& \{l_0 = l\}, \\
& \{l = l', l = l'', l \geq 2\}\rangle
\end{aligned}$$

Observe that the only difference with the UB of Ex. 5.2 is in the annotations.

5.2 Incremental Inference of Summaries

The input to the algorithm for reconstructing UB summaries is the table of summaries, named $\mathcal{U}$, which were previously computed, and the list of methods that have been reanalyzed in Alg. 2, named $\mathcal{C}$ (i.e., those elements that have belonged to $\mathcal{P}$ along the execution of Alg. 2.) The first important point to notice is that (1) the summaries for all methods in $\mathcal{C}$ must be recomputed, as well as (2) those fragments of the summaries of the ancestors of methods in $\mathcal{C}$ that correspond to the cost of the reanalyzed methods. However, the actions to perform in each case are different: (1) while the summaries of $\mathcal{C}$ must be fully recomputed, as a change in the size relations might affect all components of an UB (#iter, invariants, size relations and maximized expressions can be different), (2) in the summaries of the ancestors of a method m in $\mathcal{C}$, we just need to replace those subexpressions annotated as m with the new maximized UB for m. As in Alg. 2, we use a flag in the summaries table $\mathcal{U}$ to indicate if the content of an entry is valid. An important idea is to first process all methods in $\mathcal{C}$ and, since full summaries for them might not be yet produced (as information about relations invoked from them might not be valid), generate only UB *skeletons*.

DEFINITION 5.5 (upper bound skeleton). *In the same conditions of Def. 5.1, an upper bound skeleton for $C(\overline{x})$ is a tuple $SK_C(\overline{x}) = \langle \#iter \cdot \texttt{sexp}, \psi, \varphi \rangle$, where* sexp *is the annotated expression* $\texttt{exp}+\sum_{i=1}^{k}[\_]_{D_i}$ *and _ denotes any value. In what follows, function* $\mathsf{do\_skeleton}(C(\overline{x}))$ *generates the upper bound skeleton for C.*

The difference between summaries and skeletons is that the UBs of the invoked relations are not filled (we write _) and maximization is not yet performed. Once the skeletons have been computed for all summaries in $\mathcal{C}$, the algorithm can treat in the same way $\mathcal{C}$ and their ancestors (i.e., actions 1 and 2 above must not be distinguished anymore). In particular, given a relation m, all we need to do is replace the UBs of the relations invoked from m by their new maximized expressions (when needed). This is done by function *do_summary* of Alg. 3.

EXAMPLE 5.6 (skeleton). The change of Ex. 3.3 leads to the following skeletons for `mod` and `dup`:

$$\begin{aligned}
SK_{\texttt{while}}(l) = \langle & (3+max([\_]_{\texttt{Inc.incr}(l')}, [\_]_{\texttt{Inc2.incr}(l')}))*\mathsf{nat}(l), \\
& \{l_0 \geq l+1, l \geq 0\}, \\
& \{l = l'\}\rangle \\
SK_{\texttt{dup}}(l) = \langle & 1 + [\_]_{\texttt{while}(l')}, \\
& \{l = l_0, l_0 \geq 0\}, \\
& \{l = l', l \geq 0\}\rangle \\
SK_{\texttt{mod}}(l) = \langle & 3+[\_]_{\texttt{get}(l')}+max([\_]_{\texttt{Inc.incr}(l'')},[\_]_{\texttt{Inc2.incr}(l'')})+\ldots \\
& [\_]_{\texttt{dup}(l''')}, \\
& \{l_0 = l, l_0 \geq 2\}, \\
& \{l = l', l = l'', l \geq l''' + 1, l''' \geq 0\}\rangle
\end{aligned}$$

Note that the skeleton of `dup` differs from the initial one because of the new implementation of `incr`. This leads to a different $\#iter$ and introduces the `max` expression which includes the worst-case costs of both implementations of `incr`. As `main` had not been reanalyzed, its skeleton will not be computed again.

```
1:  proc reconstruct_summaries()
2:    $\mathcal{P} = \emptyset$
3:    for all $m(\overline{x})$ in $\mathcal{C}$ do
4:      $\langle \texttt{aexp}_m, \psi, \varphi \rangle$=do_skeleton($m(\overline{x})$)
5:      $\mathcal{U}$.update($m$,$\langle \texttt{aexp}_m, \psi, \varphi \rangle$)
6:      $\mathcal{U}$.invalidate($m(\overline{x})$ $\cup$ ancestors($m$))
7:      $\mathcal{P}$.add($m(\overline{x})$ $\cup$ ancestors($m$))
8:    for all $m(\overline{x})$ in $\mathcal{P}$ do
9:      do_summary($m(\overline{x})$)
10:   $\mathcal{U}$.validate_all()

11: function do_summary($m(\overline{x})$)
12:   $\langle \texttt{aexp}_m, \psi, \varphi \rangle$ = $\mathcal{U}$.get($m$)
13:   if $m \notin \mathcal{P}$ then
14:     return remove_annot($\texttt{aexp}_m$)
15:   for all $[\_]_{p(\overline{y})}$ in $\texttt{aexp}_m$ do
16:     if ($m \in \mathcal{C}$) $\vee$ $\mathcal{U}$.is_invalid($p$) then
17:       $\texttt{exp}_p$ = do_summary($p(\overline{y})$)
18:       $\texttt{exp}_p$ = maximize($\texttt{exp}_p$,$\overline{x}$,$\psi$,$\varphi$)
19:       $\texttt{aexp}_m$.set_expr($[\_]_p$,$\texttt{exp}_p$)
20:   $\mathcal{P}$.remove($m$);
21:   $\mathcal{U}$.update($m$,$\langle \texttt{aexp}_m, \psi, \varphi \rangle$)
22:   return remove_annot($\texttt{aexp}_m$)
```

Algorithm 3: Incremental Upper Bounds Algorithm

Intuitively, Algorithm 3 works as follows. Procedure *reconstruct_summaries* updates the entries for all methods in $\mathcal{C}$ with their skeletons and activates the invalid flag for all relations that require reprocessing (i.e., $\mathcal{C}$ and their ancestors). It also builds the list $\mathcal{P}$ made up of such relations. Function *do_summary* takes care of replacing the affected components by the new maximized UBs. The base case of the recursion (L13) is when the relation is not in $\mathcal{P}$ (either because its recomputation was not needed or because it has already been recomputed). We remove its annotations because $\mathcal{U}$ only keeps the outer level of annotations, as seen in Def. 5.3. If it is not a base case (L15-19), we need to obtain new maximized expressions for those subexpressions of $\texttt{aexp}_m$ when (i) the expression is in $\mathcal{C}$ (this is because its size relations might have changed and we need to maximize again all components) or (ii) because the invoked relation is invalid. Lines 17-19 take care of recursively obtaining the summary for the subexpression, maximizing it and placing it inside the summary. Once all components of $\texttt{aexp}_m$ have been treated (L20), the relation m is removed from the list of pending relations to process $\mathcal{P}$ and its summary is updated (L21). The result is returned without annotations in order to use it from the calling site.

EXAMPLE 5.7 (alg 3). The change in Ex. 3.3 forces the execution of *do_skeleton* for all methods in $\mathcal{C}$ = {`mod`, `dup`, `get`, `Inc2.incr`}. All those methods and their ancestors (`main`) are added to $\mathcal{P}$. Let us assume that `mod` is the first summary computed. Since `mod` is in $\mathcal{C}$, it needs the summary of `dup`. Hence, *do_summary(*`dup`*)* is invoked and a new summary of `dup` is produced by using its skeleton in Ex. 5.6. Then, `dup` is removed from $\mathcal{P}$ and we have:

$$\begin{aligned}
UB^{\mathsf{S}}_{\texttt{while}}(l) = \langle & (3 + max([2]_{\texttt{Inc.incr}(l')}, [1]_{\texttt{Inc2.incr}(l')})) * \mathsf{nat}(l), \\
& \{l_0 \geq l+1, l \geq 0\}, \\
& \{l = l'\}\rangle \\
UB^{\mathsf{S}}_{\texttt{dup}}(l) = \langle & 1 + [(3+2) * \mathsf{nat}(l)]_{\texttt{while}(l')}, \\
& \{l = l_0, l_0 \geq 0\}, \\
& \{l = l', l \geq 0\}\rangle
\end{aligned}$$

In order to obtain $\text{UB}_{\texttt{mod}}$, we need to maximize $\text{UB}_{\texttt{dup}}$, which leads to:

$$\begin{aligned}UB^{\mathsf{S}}_{\texttt{mod}}(l) = \langle & 3+[3]_{\texttt{get}(l')}+max([2]_{\texttt{Inc.incr}(l'')},[1]_{\texttt{Inc2.incr}(l'')})+\ldots \\ & [1+5*\mathsf{nat}(l-1)]_{dup(l''')}, \\ & \{l_0=l, l_0\geq 2\}, \\ & \{l=l',l=l'',l\geq l'''+1,l'''\geq 0\}\rangle\end{aligned}$$

For brevity, the recomputation of the UBs for `incr` and `get` is not described. Next, the summary of `main` is recomputed. Since `main` was invalidated but not reanalyzed, its summary can be reused as a skeleton, maximizing again only its invalidated subexpressions. Only $\text{UB}_{\texttt{mod}}$ must be maximized, and $\text{UB}_{\texttt{len}}$ can be reused:

$$\begin{aligned}UB^{\mathsf{S}}_{\texttt{main}}(l) = \langle & 5+[2+3*\mathsf{nat}(l)]_{\texttt{len}(l')}+[9+5*\mathsf{nat}(l-1)]_{\texttt{mod}(l'')}, \\ & \{l = l_0\}, \\ & \{l = l', l = l'', l \geq 2\}\rangle\end{aligned}$$

All in all, the incremental extension has avoided computing the skeleton (#iter, invariant) and one maximization for `main`, and summaries of `len` and `Inc.incr` remain the same.

THEOREM 5.8 (correctness). *Given a set of relations $\mathcal{C}$ whose CRs have changed and a table of upper bounds summaries $\mathcal{U}$, reconstruct_summaries terminates and correctly updates the upper bound summaries for all entries in $\mathcal{U}$.*

6. Experiments

We have integrated our techniques within the COSTA system [2], a resource usage analyzer for Java bytecode. COSTA implements all global analyses described in the paper and, in addition, it performs non-nullness, sign and field-sensitive value analyses. The incremental multi-domain algorithm has been implemented and applied to all domains. Our experimental evaluation has been performed on slightly modified versions of programs `Voronoi`, `Health`, `TSP`, and `MST`, from the JOlden benchmark suite [24], available at `http://costa.ls.fi.upm.es`. Such modifications (described in [4] in detail) are performed in order to overcome some limitations inherent to the size analysis and the UB solver of COSTA and are not related to the incremental extensions presented in this paper. Furthermore, we have used as benchmarks the following programs borrowed from the Apache-Commons Project [18]: StringEncrypt and ParseTarHeader from Apache-Commons-Math, and TestOrthogonal and TestDistance from Apache-Commons-Compress. The source code of all of them is available at the Apache-Commons web site. The cost model used in our experiments is the number of bytecode instructions required for executing the corresponding programs.

	Program Info.			Analysis Times		
Experiment	$\#_{\mathsf{BY}}$	$\#_{\mathsf{RU}}$	$\#_{\mathsf{EQ}}$	$\mathbf{T}_{\mathsf{CRs}}$	$\mathbf{T}_{\mathsf{UB}}$	$\mathbf{T}_{\mathsf{T}}$
MST	250	120	82	9870	570	10440
TSP	189	78	55	760	10940	11700
Health	209	73	47	4020	240	4260
Voronoi	202	66	40	460	140	600
StringEncrypt	204	136	101	419	750	1169
ParseTarHeader	341	164	115	1200	2590	3790
TestOrthogonal	221	88	56	500	180	680
TestDistance	150	93	62	550	90	640

Table 1. Benchmarks information

Table 1 shows some information about the size and complexity of the benchmark programs. For each program, the column $\#_{\mathsf{BY}}$ shows the number of bytecode instructions, $\#_{\mathsf{RU}}$ indicates the number of RBR rules, and $\#_{\mathsf{EQ}}$ the number of relations in the CR. The table also contains information about the analysis times (in ms) taken by the non-incremental analysis. The total analysis time (T_{T}) is split into the time taken to build the CRs ($\mathbf{T}_{\mathsf{CRs}}$) and the time to obtain a closed-form UB from the CRs ($\mathbf{T}_{\mathsf{UB}}$).

Our experiments are based on making a series of systematic modifications to the benchmark programs and comparing the time taken by the incremental approach with the time taken by the non-incremental one. We use the notation $P_{i-1} \rightarrow P_i$ to represent a program change, where P_{i-1} and P_i correspond to the versions of the program before and after the change, respectively. We refer to the non-incremental analysis of a program P starting from method m as $\mathcal{A}(P, m)$, while $\mathcal{A}^{\Delta}(P_i, m_i)$ is used to represent the incremental analysis of P_i starting from m_i with respect to the information computed and stored in $\mathcal{G}_{i-1}$ and $\mathcal{U}_{i-1}$ in the previous analysis. In what follows, as abbreviation, we use $\mathcal{T}_i$ to denote both $\mathcal{G}_i$ and its associated $\mathcal{U}_i$. Given a sequence of changes in a program, $P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_i$, the successive incremental analyses can be denoted as:

$$\mathcal{A}(P_0, m_0) \xrightarrow{\mathcal{T}_0} \mathcal{A}^{\Delta}(P_1, m_1) \xrightarrow{\mathcal{T}_1} \ldots \xrightarrow{\mathcal{T}_{i-1}} \mathcal{A}^{\Delta}(P_i, m_i)$$

We have performed a series of experiments which aim at evaluating the benefits of using our proposed incremental approach. The first two experiments capture the most common development scenario where (most of) the program is available and the programmer is making relatively small changes which affect one method at a time. In our expected usage scenario, analysis is triggered whenever the user saves the class file he/she is editing. Depending on the case, the change can be minimal (for example, after refactoring the code) or important, where the analysis results for the new version of the method drastically differ from the previous ones and they have to be propagated to calling methods. This first extreme case is captured by our first experiment (*Touch experiment*), where we simply replace the code of a method with a new version which in reality is identical to the previous one. In this case, analysis will reanalyze the updated method, but no change has to be propagated. The second extreme case is captured by our second experiment (*Adding experiment*), where we replace a missing implementation of a method, which simply returns the default value of the return type[2], with the final implementation.

We use the term *unweighted speedup* (S) to denote the ratio between the time required to perform the non-incremental analysis of a program, and the time required by the incremental analysis. In addition to S, experiments tables also contain the *weighted speedup* (W), weighting formula S with respect to the number of bytecode instructions of the modified method. Larger methods are more likely to be changed, thus W provides a more realistic estimate.

(1) Touch experiment: In this experiment we just have one version of the program, P_0, on which the incremental and non-incremental analyses are performed. The incremental analysis is systematically performed by starting from each method m_i in P_0 and using the previously computed $\mathcal{T}_0$:

$$\mathcal{A}(P_0, m_0) \xrightarrow{\mathcal{T}_0} \mathcal{A}^{\Delta}(P_0, m_i)$$

The speedup (S) is computed as the ratio between n times the time taken by the non-incremental analysis of P_0, and the addition of the times of the incremental analysis starting the analysis from different methods in P (m_i):

$$\mathsf{S} = \frac{n \times time(\mathcal{A}(P_0, m_0))}{\displaystyle\sum_{i=1}^{n} time(\mathcal{A}^{\Delta}(P_0, m_i))}$$

The above results (Table 2) clearly show that the incremental approach is much more efficient than the non-incremental one when handling small changes in the program, with a weighted speedup of over 5.

[2] Methods that return a non-void type keep a default `return` statement: `return null` for methods that return an object type, `return 0` for methods that return an integer, etc.

	Unweighted Speedup			Weighted Speedup		
Benchmark	S_{CRs}	S_{UB}	S_T	W_{CRs}	W_{UB}	W_T
MST	29.05	6.97	24.77	18.89	4.10	15.78
TSP	2.72	3.97	3.85	1.77	2.09	2.07
Health	7.39	2.35	6.59	5.35	1.61	4.73
Voronoi	4.18	6.22	4.53	2.45	2.49	2.46
StringEncrypt	10.90	7.09	8.11	7.56	3.92	4.74
ParseTarHeader	5.52	2.03	2.54	8.09	3.31	4.07
TestOrthogonal	3.25	10.00	3.96	2.64	4.27	2.94
TestDistance	3.09	4.95	3.26	4.05	6.09	4.25
Arith. Mean	**8.26**	**5.45**	**7.20**	**6.35**	**3.49**	**5.13**

Table 2. Touch experiment results

	Unweighted Speedup			Weighted Speedup		
Benchmark	S_{CRs}	S_{UB}	S_T	W_{CRs}	W_{UB}	W_T
MST	2.17	1.14	2.07	2.03	1.31	1.97
TSP	1.08	1.33	1.31	1.14	1.46	1.43
Health	2.57	1.17	2.41	1.75	1.23	1.71
Voronoi	1.55	2.07	1.64	1.31	1.86	1.40
StringEncrypt	1.26	1.30	1.28	2.04	2.40	2.26
ParseTarHeader	1.54	1.30	1.37	2.46	2.35	2.39
TestOrthogonal	1.14	1.79	1.26	1.22	1.55	1.30
TestDistance	1.38	1.80	1.43	1.90	2.44	1.96
Arith. Mean	**1.59**	**1.49**	**1.60**	**1.73**	**1.82**	**1.80**

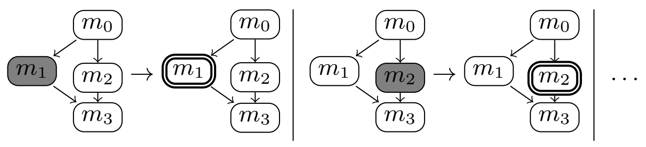
Table 3. Adding experiment results

(2) Adding experiment: This experiment captures the second extreme case in which we replace a missing implementation of a method with its final code. In this situation, the analysis of the new version will require triggering analysis of possibly multiple (transitively) calling methods.

In this case, the experiment considers different initial versions of the program P_0^i, each one missing the implementation of one method. Each P_0^i is then analyzed using the non-incremental algorithm, $\mathcal{A}(P_0^i, m_0)$. Subsequently, the code of m_i is restored producing the version P_1 (the final version of the program), and incrementally reanalyzed, $\mathcal{A}^\Delta(P_1, m_i)$, using $\mathcal{T}_0^i$:

$$\mathcal{A}(P_0^i, m_0) \xrightarrow{\mathcal{T}_0^i} \mathcal{A}^\Delta(P_1, m_i)$$

This procedure is illustrated in Figure 3, using as example a small call graph composed by four methods. Colored nodes represent empty methods, and white nodes represent implemented methods.



$$\mathcal{A}(P_0^1, m_0) \xrightarrow{\mathcal{T}_0^1} \mathcal{A}^\Delta(P_1, m_1) \quad \mathcal{A}(P_0^2, m_0) \xrightarrow{\mathcal{T}_0^2} \mathcal{A}^\Delta(P_1, m_2)$$

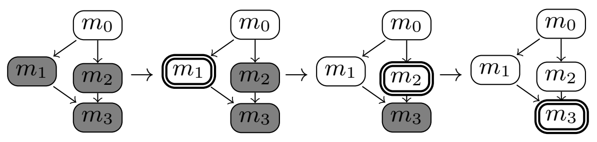
Figure 3. Modification experiment scheme

This procedure is systematically applied in order to modify all methods of the benchmark program. The speedup of the incremental approach is calculated as follows:

$$\mathsf{S} = \frac{n \times time(\mathcal{A}(P_1, m_0))}{\sum_{i=1}^{n} time(\mathcal{A}^\Delta(P_1, m_i))}$$

Experimental results (Table 3) clearly show that the incremental approach efficiently handles method modifications in a program. It is efficient in both parts of the resource usage analysis, in the generation of `CRs` and the `UB` solving. Altogether it achieves a significant improvement over non-incremental analysis, being almost two times faster.

(3) Top-down development experiment: A question which remains to be answered is whether the incremental approach can be less efficient than analyzing the whole program. This question is important since there is no formal guarantee that the incremental analysis will be more efficient than the analysis from scratch. In fact, it is possible to find situations where global analysis can be more efficient. To assess this situation, our third experiment tries to perform a stress test of the worst possible situation that can arise. This occurs when we analyze in an incremental fashion a program, by adding a method at a time, following a top-down order in the call graph. In the experiment, we start with empty implementations that lack the content of the method for all methods. We progressively add the implementations one by one starting from the root of the call graph. This scenario will require the largest possible number of reanalysis.



$$\mathcal{A}(P_0, m_0) \xrightarrow{\mathcal{T}_0} \mathcal{A}^\Delta(P_1, m_1) \xrightarrow{\mathcal{T}_1} \mathcal{A}^\Delta(P_2, m_2) \xrightarrow{\mathcal{T}_2} \mathcal{A}^\Delta(P_3, m_3)$$

Figure 4. Top-down development experiment scheme

The scenario is illustrated in Figure 4. The experiment starts from the analysis of an initial version $\mathcal{A}(P_0, m_0)$ where all methods except the main method m_0 are empty. In the second step, the code of a method m_1, directly invoked by m_0, is added generating a new version of the program P_1, and the incremental analysis $\mathcal{A}^\Delta(P_1, m_1)$ is applied by using $\mathcal{T}_0$. In the following steps, the contents of the remaining methods are added one by one (m_i), producing different versions of the program (P_i). The speedup (S) of the incremental analysis is computed as

$$\mathsf{S} = \frac{\sum_{i=1}^{n} time(\mathcal{A}(P_i, m_i))}{\sum_{i=1}^{n} time(\mathcal{A}^\Delta(P_i, m_i))}$$

	Unweighted Speedup			Weighted Speedup		
Benchmark	S_{CRs}	S_{UB}	S_T	W_{CRs}	W_{UB}	W_T
MST	3.42	1.43	2.96	3.42	1.48	2.94
TSP	1.04	1.34	1.32	1.05	1.40	1.38
Health	1.18	1.09	1.17	1.05	0.96	1.04
Voronoi	1.33	1.74	1.40	1.10	1.52	1.17
StringEncrypt	1.35	1.31	1.32	1.51	1.63	1.58
ParseTarHeader	1.29	1.26	1.27	1.29	1.41	1.37
TestOrthogonal	1.09	1.85	1.23	0.84	1.16	0.91
TestDistance	1.36	2.26	1.44	1.39	2.16	1.46
Arith. Mean	**1.51**	**1.53**	**1.51**	**1.46**	**1.46**	**1.48**

Table 4. Top-down development experiments results

Table 4 shows that, even in the extreme case of having the reanalyze a large number of methods, the use of our incremental analysis is not worse than the global one. Only in one example (`TestOrthogonal`) there is a small slowdown. While, there is an overall gain of 1.48. This, together with the first two experiments, indicates that incremental analysis will provide important gains in the most common and realistic scenarios while not introduce overhead in the less optimal scenarios.

7. Conclusions and Related Work

The traditional global analysis scheme in which all the program code is analyzed from scratch and no previous analysis information is available is unsatisfactory in many situations. This paper shows that incremental analysis of a complex property –the resource consumption of executing a program– is feasible and much more efficient in certain contexts than traditional (non-incremental) global analysis. The most related approach to ours is [13], which develops a generic incremental analysis algorithm for constraint logic programs. In addition to the language differences, and the fact that we consider a global memory, their incremental algorithm does not handle domain dependencies like ours, which is fundamental for an application such as resource usage which relies on multiple pre-analyses with dependencies among them. Besides, our work provides novel definitions for cost summaries which enable the incremental reconstruction of cost functions, a problem that has not been considered before.

Other approaches to incremental analysis are developed for other purposes, e.g., [25] proposes an efficient incremental parser for general context-free grammars which allows generating incremental tools. The work in [12] develops an approach to incremental static semantic analysis for object-oriented languages using door attribute grammars as a way to maintain incremental information, while our work is mostly focused on the reconstruction of the analysis information and the *cost summaries*. An incremental analysis based on incremental specifications such as those found in formal models is presented in [9], while we do not rely on specifications. The notion of summary has been previously used in other contexts [8, 19] different from incremental analysis. Modular analysis [7, 16] is related to incremental analysis in that it aims at reducing the time and memory required to perform analysis by splitting the program into smaller parts and storing analysis results, either automatically or by using user-provided summaries. Our technique is modular in the sense that it automatically stores summaries, though it does not split the program into smaller parts. On the other hand, modularity per se does not handle the efficient recomputation of analysis results after a program change.

Acknowledgments

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 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *Proc. of SAS'10*, volume 6337 of *LNCS*, pages 100–116. Springer, 2010.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[3] E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *Proc. of LCTES'11*, pages 21–30. ACM Press, 2011.

[4] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM Press, 2010.

[5] 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.

[6] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.

[7] P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Compiler Construction*, 2002.

[8] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, pages 270–280. ACM, 2008.

[9] C. A. Lakos G. Lewis. Towards incremental analysis. In *Workshop on Formal Methods for Dependable Systems (FMDS)*, 1998.

[10] Samir Genaim and Fausto Spoto. Constancy analysis. In *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.

[11] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.

[12] G. Hedin. An object-oriented notation for attribute grammars. In *Proc. of ECOOP'89*, pages 329–345, 1989.

[13] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.

[14] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.

[15] M. Kero, P. Pietrzak, and Nordlander J. Live Heap Space Bounds for Real-Time Systems. In *Proc. of APLAS'10*, volume 6461 of *LNCS*, pages 287–303. Springer, 2010.

[16] Francesco Logozzo. Practical verification for the working programmer with codecontracts and abstract interpretation - (invited talk). In *Proc. of VMCAI'11*, volume 6538 of *LNCS*, pages 19–22. Springer, 2011.

[17] A. Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *Proc. of LCTES'06*, pages 54–63. ACM, 2006.

[18] Apache Commons Project. `http://commons.apache.org/`.

[19] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL'95*, pages 49–61, 1995.

[20] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 95–110. Springer, 2006.

[21] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 320–335. Springer, 2005.

[22] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode based on Path-Length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[23] Fausto Spoto and Thomas P. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.

[24] JOlden Suite. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.

[25] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *Proc. of PLDI'97*, pages 31–43, 1997.

[26] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.