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 F.3.2 [Logics and Meanings of Programs]: Program Analysis; F.2.9 [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 [2, 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 [3]. 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 [1] 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 until 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 [3, 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 which 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 [3], X10 [4], 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 procedures 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:

$$
\text{rule} ::= p(\bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n
$$

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 guard 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 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 in [3, 4, 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.

```java
class C {
    void main (List l) {
        int s = len(l);
        if (s % 2 != 0 || s < 2) return;
        mod(1);
    }
    void mod (List l) {
        Inc o = get(1);
        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:

- `mod(l, ⟨⟩) ← get(l, ⟨⟩), call_incr(o, l, (l'))`, `dup(l', ⟨⟩)`
- `get(l, ⟨⟩) ← r := new Inc`
- `dup(o, l, ⟨⟩) ← while(o, l, (l'))`
- `while(o, l, ⟨⟩) ← l = null`
- `while(o, l, (l')) ← l ≠ null, l'.data := l.data * 2, call_incr(o, l', (l''))`, `while(o, l', (l''))`.
- `call_incr(o, l, (l')) ← type(o, Inc)`
- `Inc.incr(l, (l')) ← l' := l.next, l'' := l'.next`

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-
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 $\Rightarrow$ are defined in a 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 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 $L$ where the answers for all procedures are stored and (2) the queue of events $Q$, 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 $L$ (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 $L$ (L35). When the instruction is a procedure call (L25), $get_proc_answer$ first checks if a previously computed answer exists in $L$ (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 $Q$ during its execution when (i) a method must be analyzed for a given calling pattern in $L$, 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 $Q$ (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 [2] 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 $CP_D \rightarrow AP_D$ for the following domains:

1. $D_{class}$, where $x \in \{C_1, \ldots, C_n\} \rightarrow AP_D$ (resp. $CP_D$) 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).

**Algorithm 1:** Fixed-point algorithm (operators $\sqcup$, $\sqcap$, $\sqsubseteq$ are parametric w.r.t. the analysis domain, $D$)
(5) Define size, where $AP[m]$ (resp. $CP[m]$) are linear constraints describing the relation between the size of $x$ and $y$ after (resp. before) executing $m$ [4].

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.

**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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>void mod(1)</td>
<td>$cl : {f : \text{List}} \mapsto \bot$</td>
</tr>
<tr>
<td></td>
<td>$sz : {l \geq 2} \mapsto {l \geq 2}$</td>
</tr>
<tr>
<td></td>
<td>Inc get(1)</td>
</tr>
<tr>
<td></td>
<td>$sz : {l \geq 2} \mapsto {l \geq r, r = 1}$</td>
</tr>
<tr>
<td></td>
<td>void dup(o,1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>List incr(1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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 $G$ which contains the set of summaries for all previously analyzed methods such that each entry stored in $G$ 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 in $D$...
m for p and domain D is valid. The call invalidate(p,D) sets up to invalidate the flags for the set of domains 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 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 R. Now, we need to take care of reanalyzing all those methods that relied on answers for methods in R. The list P is used for this purpose. P is formed by pairs of the form (m,D) where D is a list of domains for which m must be reanalyzed according to Fig. 2. Initially, P contained an entry for the changed method and all domains (L2). It is later updated in L14 as follows. For each element in R, we know that its summary for such domain has changed (L10). We compare the new answer APn with the one in the summary APn (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 M and a set of domains D and, for each m ∈ M, D ∈ D, if there exists an entry (m,Dm), it is updated to (m,add(Dm,D)), where the addition of D is ordered according to Fig. 2. If it does not exist, it adds (m, D). (L15). 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 P in L16. Finally, the fixed point is reached when there are no more methods to analyze in 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 G does not contain the newly generated answer pattern {r:m:Inc, Inc2}. Therefore, all direct ancestors of get must be reanalyzed, namely mod is added to 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 G has CP2 = {o:m:Inc, l:List} and thus it is not applicable to CP2 = {o:m:Inc, Inc2, l:List}. Both CPs are joined (resulting in CP2) and dup is reanalyzed for the class domain w.r.t. CP2. Entries for dup for dep(class) in G are invalidated and added to P. Due to polymorphism, during the analysis of dup, an invocation to Inc2.incr1 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 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inc.get(1)</td>
<td>cf {l:List} → r:Inc.incr2 {l ≥ 2 \rightarrow l ≥ 2, r = 1}</td>
</tr>
<tr>
<td>void dup(o,1)</td>
<td>cf {o:Inc, Inc2, l:List} → \⊥</td>
</tr>
<tr>
<td>List Inc2.incr(1)</td>
<td>cf {l:List} → r:List, l ≥ 2 \rightarrow l ≥ 2, r ≥ 0</td>
</tr>
</tbody>
</table>

**Theorem 3.4 (correctness).** Given a program P, its analysis results stored in G, and a method m which has been modified, incremental fixpoint(m) terminates and returns a correct 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. 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 ≥ l + 2, l’ ≥ 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:

```
mod(l) = 3 + get(l) + call.incr(l) + dup(l) \{l ≥ l + 2\}
get(l) = 1
while(l) = 3 + call.incr(l) + while(l) \{l’ ≥ 0, l ≥ l + 2\}
dup(l) = 1 + while(l) \{l ≥ 0\}
call.incr(l) = 0 + Inc.incr(l) \{l = 0\}
while(l) = 0 \{l ≥ 2\}
Inc.incr(l) = 2
```

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 \( P \)). 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 \( P \) contains \{get, mod, dup, Incr2, incr, get\}. Thus, the following CRs of while must be generated by using the standard CR generation as explained above:

\[
\begin{align*}
\text{while}(l) = 3 + \text{callincr}(l) + \text{while}'(l') & \quad \{l' \geq 0, l \geq l' + 1\} \\
\text{callincr}(l) = 0 + \text{Incrincr}(l) & \quad \{} \\
\text{Incrincr}(l) = 2 & \quad \{l \geq 2\} \\
\text{callincr}(l) = 0 + \text{Incr2incr}(l) & \quad \{} \\
\text{Incr2incr}(l) = 1 & \quad \{l \geq 1\}
\end{align*}
\]

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 to assume that the length of the list decreases (in the worst-case) by \( \Theta(l) \). To accomplish this, we need an upper bound for the size relations (which do not call any other relation) and consecutively replacing such CRs 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:

\[
(C(\pi)) = \exp + \sum_{i=1}^{k} D_i(\pi_i) + C(\pi_i, \varphi)
\]

In [1], 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 [1]).** An upper bound for \( C(\pi) \) is UB\(_C(\pi) = \#\text{iter} \times \text{mexp} \) (base cases are ignored for simplicity) where:

1. \(
\#\text{iter} \text{ is an upper bound on the number of recursive calls of } C,
\)
2. the size relations \( (\varphi) \) are linear constraints on variables \( \pi \) and \( \pi_i, \pi_i^\prime \)
3. the invariant \( (\psi) \) relates variables \( \pi, \pi_i^\prime \) to their initial values \( \pi_0 \) (we denote the initial value of a variable \( x \) as \( x_0 \)).
4. \( \text{ub} \) are upper bounds for each call \( D_i(\pi_i) \).
5. for each recursive equation, we have that:

\[
\text{mexp}' = \maximize(\exp, \pi, \psi, \varphi) + \sum_{i=1}^{k} \maximize(\text{ub}, \pi, \psi, \varphi)
\]

where function \( \maximize(\epsilon, \pi, \psi, \varphi) \) returns the maximization of \( \epsilon \) for \( \psi \) w.r.t. the equation entry variables \( \pi \).

**Example 5.2 (ub).** Let us apply the above definition to solve the CR \( \text{mod} \) in Ex. 4.1 which has obtained before applying the modification in Ex. 3.3. The standalone CRs \( \text{Incr} \) and \( \text{get} \) are already solved since they are not recursive. Next, we solve \( \text{while}(l) \) which is required to solve \( \text{dup} \). The following invariant holds for this CR

\[
\psi = \{l_0 > l' + 2\}
\]

which simply states that the length of the list in the recursive call is strictly smaller than the length of the initial list. An UB of \( \#\text{iter} \) for while is \( \text{nat}(l'/2) \). Function \( \text{nat}(v) = \max\{(v, 0)\} \) is used by the UB solver to avoid negative evaluations. By applying Def. 5.1, UB\(_{\text{dup}}(l') = 1 + (3 + 2) \times \text{nat}(l'/2) \). Finally, when computing an UB for \( \text{mod}(l) \), the UB for \( \text{dup} \) has to be maximized w.r.t. the entry \( l, \varphi = \{l' \geq l' + 2, l' \geq 0\} \) and \( \psi = \{l_0 = l_0, l_0 \geq 2\}. \) This results in UB\(_{\text{mod}}(l) = 6 + (1 + 5 \times \text{nat}(l/2) - 1) \), where we can observe that the maximum cost of \( \text{dup} \) within \( \text{mod} \) 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 originates 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 5.2 (upper bound summary).** In the same conditions of Def. 5.1, an UB summary for \( C(\pi) \) is a tuple UB\(_{\text{summary}}(\pi) = (\#\text{iter} \times \text{aexp}, \psi, \varphi) \), where \( \text{aexp} = \maximize(\exp, \psi, \varphi, \pi) + \sum_{i=1}^{k} \mathit{\text{maximize}}(\text{removeannotate}(\epsilon), \psi, \varphi) \). s.t.:

- \( \epsilon \mathit{\text{ub}} \) is the annotated cost expression in the upper bound summary of \( D_i \).
- function \( \text{removeannotate} \) removes the annotations of an expression, and
- \( e_D \) is a constant value with the name of the relation \( D \) it originates from.

The notation \( \text{aexp.setexpr}(\epsilon, m, \exp) \) is used to rewrite in \( \text{aexp} \) the annotated subexpression \( \epsilon \) 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:

\[
\text{UB}_{\text{while}}(l) = (3 + 2 \times \text{incr}(l')) \times \text{nat}(l/2), \\
\{l_0 > l + 2, l \geq 0\}, \\
\{l = l'\}
\]
$UB^a_{dup}(l) = (1 + [(3 + 2) \cdot nat(l/2)])_{\text{while}(l'),}$
\[\{l_0 = l, l_0 \geq 0\},\]
\[\{l = l', l \geq 0\}\]

$UB^a_{mod}(l) = (3 + (l_0)_{\text{get}(l')}) + (2)_{\text{incr}(l')} + (1 + 5 \cdot nat(l/2 - 1))_{\text{dup}(l')},$
\[\{l_0 = l, l_0 \geq 2\},\]
\[\{l = l', l = l', l \geq l'' + 2, l'' \geq 0\}\]

$UB^a_{min}(l) = (6 + 2 + 3 + \text{nat}(l))_{\text{set}(l')} + (7 + 5 \cdot nat(l/2 - 1))_{\text{mod}(l')},$
\[\{l_0 = l\},\]
\[\{l = l', l = l', l \geq 2\}\]

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 $U$, which were previously computed, and the list of methods that have been reanalyzed in Alg. 2, named $C$ (i.e., those elements that have belonged to $P$ along the execution of Alg. 2). The first important point to notice is that (1) the summaries for all methods in $C$ must be recomputed, as well as (2) those fragments of the summaries of the ancestors of methods in $C$ that correspond to the cost of the reanalyzed methods. However, the actions to perform in each case are different: (1) while the summaries of $C$ must be fully recomputed, as a change in the size relations might affect all components of an UB ($\#\text{iter}$, invariants, size relations and maximized expressions can be different), (2) in the summaries of the ancestors of a method $m$ in $C$, 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 $U$ to indicate if the content of an entry is valid. An important idea is to first process all methods in $C$ 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 $C(\mathcal{E})$ is a tuple $SK_C(\mathcal{E}) = (\#\text{iter} \cdot aexp, \psi, \varphi)$, where $aexp$ is the annotated expression $\sum_{i=1}^k \exp_i$ and $\_\_\_$ denotes any value. In what follows, function $\text{do_skeleton}(C(\mathcal{E}))$ 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 $C$, the algorithm can treat in the same way $C$ 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 $\text{do_summary}(\text{Algorithm 3})$.

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

$SK_{\text{while}}(l) = (3 + \max([\exp_{\text{incr}(l')}, \exp_{\text{incr}(l')}) \cdot \text{nat}(l),$
\[\{l_0 = l, l_0 \geq 0\},\]
\[\{l = l'\}\]

$SK_{\text{dup}}(l) = (1 + [\exp_{\text{while}(l')},$
\[\{l_0 = l, l_0 \geq 0\},\]
\[\{l = l', l \geq 0\}\]

$SK_{\text{mod}}(l) = (3 + [\exp_{\text{get}(l')} + \max([\exp_{\text{incr}(l')}, \exp_{\text{incr}(l')}) + \ldots$
\[\ldots\]
\[\{l_0 = l, l_0 \geq 2\},\]
\[\{l = l', l = l', l \geq l'' + 1, l'' \geq 0\}\]

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

Intuitively, Algorithm 3 works as follows. Procedure $\text{reconstruct_summaries}$ updates the entries for all methods in $C$ with their skeletons and activates the invalid flag for all relations that require reprocessing (i.e., $C$ and their ancestors). It also builds the list $P$ made up of such relations. Function $\text{do_summary}$ 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 $P$ (either because its recomputation was not needed or because it has already been recomputed). We remove its annotations because $U$ 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 $aexp_m$ when (i) the expression is in $C$ (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 $aexp_m$ have been treated (L20), the relation $m$ is removed from the list of pending relations to process $P$ 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 $\text{do_skeleton}$ for all methods in $C = \{\text{mod}, \text{dup}, \text{get}, \text{Inc2}, \text{incr}\}$. All those methods and their ancestors ($\text{main}$) are added to $P$. Let us assume that $\text{mod}$ is the first summary computed. Since $\text{mod}$ is in $C$, it needs the summary of dup. Hence, $\text{do_summary}(\text{dup})$ is invoked and a new summary of dup is produced by using its skeleton in Ex. 5.6. Then, dup is removed from $P$ and we have: $UB^a_{\text{while}}(l) = (3 + \max([\exp_{\text{incr}(l')}, [\exp_{\text{incr}(l')}) \cdot \text{nat}(l),$
\[\{l_0 = l, l_0 \geq 0\},\]
\[\{l = l'\}\]

$UB^a_{\text{dup}}(l) = (1 + [(3 + 2) \cdot \text{nat}(l)]_{\text{while}(l')},$
\[\{l_0 = l, l_0 \geq 0\},\]
\[\{l = l', l \geq 0\}\]

In order to obtain $UB_{\text{mod}}$, we need to maximize $UB_{\text{dup}}$, which leads to:
summaries terminates and correctly updates the upper
1200 2590 3790
150 93 62
4020 240 4260
460 140 600
419 750 1169
250 120 82
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 re-
fer to the non-incremental analysis of a program $P$ starting from
method $m$ as $A(P, m)$, while $A^\Delta(P, m)$ is used to represent
the incremental analysis of $P$, starting from $m$, with respect to the
information computed and stored in $G_{i-1}$ and $U_{i-1}$ in the previous
analysis. In what follows, as abbreviation, we use $T_i$ to denote both
$G_i$ and its associated $U_i$. Given a sequence of changes in a program,
$P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_l$, the successive incremental analyses can
be denoted as:
$$A(P_0, m_0) \xrightarrow{T_0} A^\Delta(P_1, m_1) \xrightarrow{T_1} \cdots \xrightarrow{T_{l-1}} A^\Delta(P_l, m_l)$$
We have performed a series of experiments which aim at evalu-
ating the benefits of using our proposed incremental approach. The
first two experiments capture the most common development sce-
ario 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 refactor-
ing 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 $m_1$ in $P_0$
and using the previously computed $T_0$:
$$A(P_0, m_0) \xrightarrow{T_0} A^\Delta(P_0, m_1)$$
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 $P (m_1)$:
$$S = \frac{n \times \text{time}(A(P_0, m_0))}{\sum_{i=1}^{n} \text{time}(A^\Delta(P_0, m_i))}$$
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 meth-
ods that return an integer, etc.
(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 \( P_0 \), each one missing the implementation of one method. Each \( P_0 \) is then analyzed using the non-incremental algorithm, \( A(P_0, m_0) \). Subsequently, the code of \( m_i \) is restored producing the version \( P_i \) (the final version of the program), and incrementally reanalyzed, \( A^\Delta(P_i, m_i) \), using \( T_0 \):

\[
A(P_0, m_0) \xrightarrow{T_0} A^\Delta(P_i, m_i)
\]

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.

![Figure 3. Modification experiment scheme](image)

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:

\[
S = \frac{n \times \text{time}(A(P_i, m_0))}{\sum_{i=1}^{n} \text{time}(A^\Delta(P_i, m_i))}
\]

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 experiments: 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 reanalyses.

![Figure 4. Top-down development experiment scheme](image)

The scenario is illustrated in Figure 4. The experiment starts from the analysis of an initial version \( A(P_0, m_0) \) 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 \( A^\Delta(P_1, m_1) \) is applied by using \( T_0 \). 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

\[
S = \frac{\sum_{i=1}^{n} \text{time}(A(P_i, m_i))}{\sum_{i=1}^{n} \text{time}(A^\Delta(P_i, m_i))}
\]

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), within 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