A Multi-Domain Incremental Analysis Engine and its Application to Incremental Resource Analysis ☆

Elvira Albert[a], Jesús Correas[a], Germán Puebla[b], Guillermo Román-Díez[b]

[a] *DSIC, Complutense University of Madrid (UCM), Spain*
[b] *DLSIIS, Technical University of Madrid (UPM), Spain*

Abstract

The aim of *incremental* 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. The first contribution of this article is a *multi-domain* incremental fixed-point algorithm for a sequential Java-like language. The algorithm is multi-domain in the sense that it interleaves the (re-)analysis for multiple domains by taking into account dependencies among them. Importantly, this allows the incremental analyzer to invalidate only those analysis results previously inferred by certain *dependent* domains. The second contribution is an incremental *resource usage* analysis which, in its first phase, uses the multi-domain incremental fixed-point algorithm to carry out all global pre-analyses required to infer cost in an interleaved way. Such resource analysis is parametric on the cost metrics one wants to measure (e.g., number of executed instructions, number of objects created, etc.). Besides, we present a novel form of *cost summaries* which allows us to incrementally reconstruct only those components of cost functions affected by the changes. Experimental results in the COSTA system show that the proposed incremental analysis provides significant performance gains, ranging from a speedup of 1.48 up to 5.13 times faster than non-incremental analysis.

1. Introduction

Static cost analysis [39] (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*, i.e., without actually executing the program. 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

☆ This work is an extended and revised version of PEPM'12 [5].

Email addresses: elvira@sip.ucm.es (Elvira Albert), jcorreas@fdi.ucm.es (Jesús Correas), german@fi.upm.es (Germán Puebla), groman@fi.upm.es (Guillermo Román-Díez)

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 *class* (or *application extraction*) analysis [34] which determines the set of reachable classes which must be considered by subsequent global analyses. Besides, analyzers of OO languages often perform non-nullness analysis [22, 33] which allows removing unsatisfiable nullness checks.
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 [35] relies on a series of pre-analyses, namely, *sharing*, *acyclicity* and *constancy*.
3. *Cost functions.* In the last step, cost relation solvers [2] try to obtain *cost functions* which are not in recursive form and hence are directly evaluable. Since exact solutions seldom exist, 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 analysis is performed by a sequence of *global* analyses, i.e., 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, current state of the art resource analyses are non-incremental [3, 18, 21]. Incremental analysis has applications in the following two scenarios: (1) *Software development.* 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. (2) *SPLE.* One increasing trend in software engineering is to develop multiple, similar software products instead of just a single individual program. Software Product Line Engineering (SPLE) [10] offers a solution which is based on the explicit modelling of what is common and what differs between product variants, and on building a reuse infrastructure (product line asset) that can be instantiated and possibly extended to build the desired similar products. Building a product consists in incrementally assembling the product from the product line assets by applying the selected features.

Resource analysis is a compute-intensive task and, in scenarios as those mentioned above, starting analysis from scratch (instead of reusing previous

results) is inefficient in most cases. Consider a given program, its analysis results and a series of changes to the program, e.g., extensions to build a new product in the SPLE scenario or modifications to fix a bug in the software development scenario. Incremental resource usage analysis aims at obtaining the new analysis results more efficiently, without having to (re-)analyze fragments of code which are not affected by the changes.

1.1. Summary of Contributions

In this article, we present a generic incremental multi-domain analysis engine for an imperative object-oriented programming language, and study its application in the context of incremental resource usage analysis. The main challenge when devising an incremental analysis framework is to recompute the least possible information and do it in the most efficient way. Our main contributions can be summarized as follows:

- We introduce a *multi-domain* incremental analysis engine which interleaves the computation for multiple analysis domains. Dealing with a large number of pre-analyses, and threading the information about change through all of them is the main challenge we face here. Our algorithm takes into account the dependencies among them in such a way that it is possible to invalidate only part of the pre-computed dependent information.
- We describe how the previous algorithm can be used in order to incrementally compute all global pre-analyses required to infer the resource usage of a program (including class analysis, nullness, sharing, cyclicity, constancy and size analyses mentioned above). All such analysis information is included in the so-called *cost method summary* and used by the multi-domain incremental analysis engine.
- Even a small change within a method (e.g., adding an instruction) can change the overall cost of the program. Our contribution, in order to minimize the amount of information that needs to be recomputed, is on the notion of *upper bound summary* which allows us to distinguish the cost subcomponents associated to each method, so that the final cost functions can be recomputed by replacing only the affected subcomponents.
- The correctness of our approach has been proved within the article. In addition, complete details of some proofs can be found in an appendix of the electronic version of this article.
- We have implemented the incremental analysis 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 [36] and from the *Apache Commons* Project [28]. 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 approach to the incremental inference of resource usage bounds.

1.2. Organization

The rest of the article is organized as follows. In Section 2, we present our running example, which is written in Java using delta-oriented programming style [31] in order to define the changes to be applied on a core module. In Section 3, we first describe the simple intermediate language on which the analysis is developed. This language has been already used to formalize other resource analyses (e.g., [3, 24]). We then present a classical event-based global analysis algorithm which is generic on the abstract property of interest.

Our multi-domain incremental analysis algorithm is introduced in Section 4. Given a change on a method, it reconstructs the information which needs to be recomputed for a given set of domains. A unique feature of our incremental algorithm is that it is able to handle multiple domains such that the minimal amount of reanalysis is performed.

Section 5 presents first the application of the generic multi-domain algorithm in Section 4 in order to infer all OO global pre-analysis required to infer the cost (item 1 above) as well as the size analysis required to set up cost relations (item 2 above). Once such global information is recomputed, cost relations can be generated locally (and thus incrementally) for the code fragment that has been changed.

In Section 6, an incremental resource usage bound inference is presented. The algorithm recomputes the cost functions of the subcomponents affected by the change and relies on the notion of resource bound *skeleton* which annotates the cost subexpressions with an identifier of the method they originate from, and besides deletes the content of the subexpressions that must be recomputed, leaving only the skeleton.

In Section 7, the experimental results obtained by applying our approach over a set of realistic programs are presented. The experiments try to simulate the classical scenarios for software development and for SPLE in which an incremental approach can be of interest. Finally, related work is discussed in Section 8, and in Section 9 the conclusions are presented.

2. Running Example

Our running example, shown in Figure 1, represents a very simple case of a SPLE scenario. The example uses delta-oriented programming (DOP) [31] to model the changes to be applied to a program. DOP has been proposed as a novel programming language approach particularly designed for implementing Software Product Lines (SPL), based on the concept of program *deltas*. In DOP, the implementation of a SPL is divided into a core module and a set of delta modules. The core module comprises a set of classes that implement a complete program (or product) for a valid feature configuration. This allows developing the core module with well-established single-application engineering techniques to ensure its quality. Delta modules specify changes to be applied to the core module in order to implement other products. A delta module can add classes to a product implementation or remove classes from a product implementation.

Furthermore, existing classes can be modified by changing the super class, the constructor, and by additions, removals and renamings of fields and methods.

In our running example, the core module is composed of classes C, Mover, Mover2 and List and we also have a delta module called SingleOrDoubleStep which specifies the changes to be applied to the core module. Method main receives a list of integers, checks the length of the list and modifies some of its elements by invoking mod. In the core module, elements in odd positions of the list are modified, starting from the third element of the list. The delta module SingleOrDoubleStep modifies the implementation of class C by changing the implementation of method get such that now it allows selecting two alternative ways of traversing the list. In particular, depending on the value of the first element of the list, all elements in the list are modified, or only those at odd positions.

This modification of the program does not only change the resource consumption of method get, but also of methods that are ancestors or descendants of get. For instance, method mod is an ancestor of get (i.e. mod calls get) and its resource consumption includes that of get and hence will be affected by any change in get. Method dup is a descendant of mod and it uses the results obtained from the invocation to get, through its ancestor mod, and hence its resource consumption can be affected when the implementation of get changes. As we will see in the following sections, when performing the incremental analysis of a change specified by means of a delta, besides the method being modified, its *ancestors* and *descendants* must also be considered. Those methods must be reanalyzed if the previous analysis results are no longer valid (i.e. *correct*) after applying the delta modification.

3. Global Analysis of OO Languages

To formalize the analysis algorithms, we consider a *rule-based* intermediate representation of a Java-like programming language (e.g., Java, Java bytecode [3], and the sequential sublanguages of X10 [4] and ABS [23, 1] can be compiled into this language). Although this representation has been first defined by us in [3], some of its main features are similar to an intermediate form used in Soot [37] (namely a main common idea is that the stack is flattened into local variables). It should be noted also that our representation has been adopted by other authors [24]. 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.

3.1. 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:

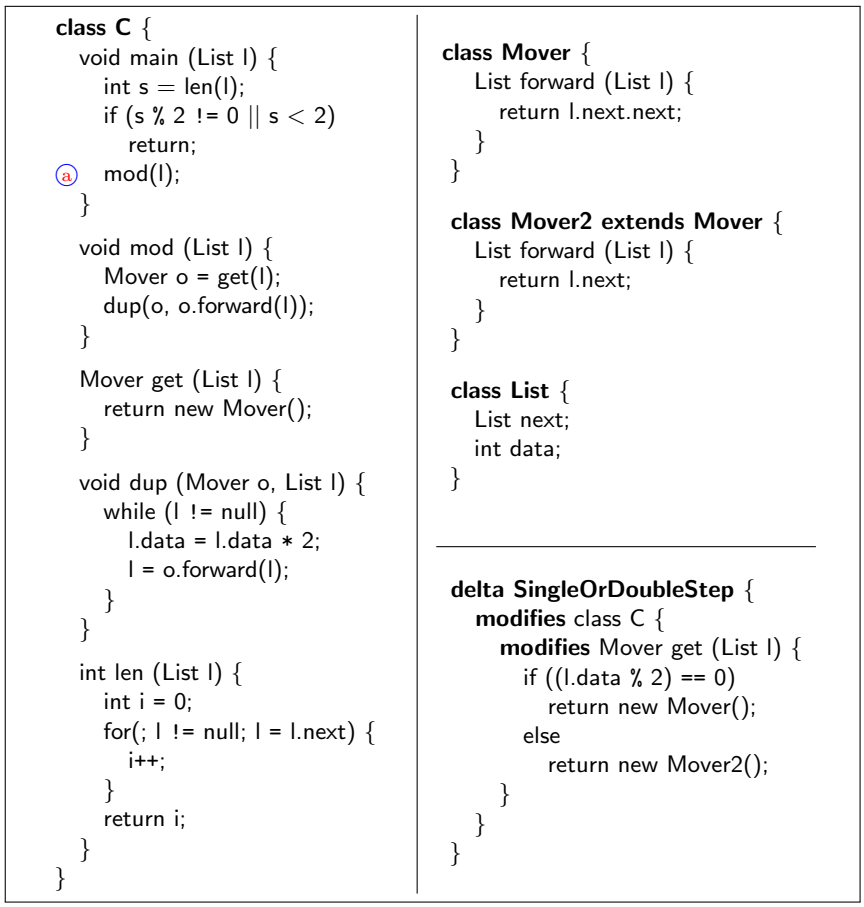


Figure 1: Running Example written in Delta-Oriented Programming Style.

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

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 out-

put 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, and function *callers(p)* to obtain the set of procedures that invoke p directly. 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. For the generation of the CFG, a rapid type analysis (RTA) [9] is used. RTA simply decides which are the classes to be loaded and generates accordingly all possible instantiation types when a virtual call is found. A standard single static assignment transformation is applied on the rules variables. The process is identical to that in [4, 3, 24], hence, we will not go into the technical details of the transformation, but just show the intuition by means of an example. In [2], it is formalized that traces in the RBR mimic those of bytecode and, hence, we can reason in the cost of the Java bytecode program by analyzing the RBR.

Example 3.1 (RBR). *The following RBR for some methods of the running example before applying the delta* SingleOrDoubleStep *illustrates all features of the intermediate representation mentioned above:*

$$
\begin{array}{lcl}
main(\langle l\rangle,\langle\rangle) & \leftarrow & [true], len(\langle l\rangle,\langle s\rangle), s' := s\%2, if(\langle l,s,s'\rangle,\langle\rangle)\\
if(\langle l,s,s'\rangle,\langle\rangle) & \leftarrow & [s'\neq 0]\\
if(\langle l,s,s'\rangle,\langle\rangle) & \leftarrow & [s'=0], if_2(\langle l,s\rangle,\langle\rangle)\\
if_2(\langle l,s\rangle,\langle\rangle) & \leftarrow & [s<2]\\
if_2(\langle l,s\rangle,\langle\rangle) & \leftarrow & [s\geq 2], mod(\langle l\rangle,\langle\rangle)\\
len(\langle l\rangle,\langle i'\rangle) & \leftarrow & [true], i := 0, for(\langle l,i\rangle,\langle l,i'\rangle)\\
for(\langle l,i\rangle,\langle l,i\rangle) & \leftarrow & [l=\mathtt{null}]\\
for(\langle l,i\rangle,\langle l'',i''\rangle) & \leftarrow & [l\neq\mathtt{null}], i' := i+1, l' := l.next, for(\langle l',i'\rangle,\langle l'',i''\rangle)\\
mod(\langle l\rangle,\langle\rangle) & \leftarrow & [true], get(\langle l\rangle,\langle o\rangle), call\_forward(\langle o,l\rangle,\langle l'\rangle),\\
 & & dup(\langle o,l'\rangle,\langle\rangle)\\
get(\langle l\rangle,\langle r\rangle) & \leftarrow & [true], r := new\ \mathsf{Mover}\\
dup(\langle o,l\rangle,\langle\rangle) & \leftarrow & [true], while(\langle o,l\rangle,\langle l'\rangle)\\
while(\langle o,l\rangle,\langle l\rangle) & \leftarrow & [l=\mathtt{null}]\\
while(\langle o,l\rangle,\langle l'''\rangle) & \leftarrow & [l\neq\mathtt{null}], l'.data := l.data*2,\\
 & & call\_forward(\langle o,l'\rangle,\langle l''\rangle), while(\langle o,l''\rangle,\langle l'''\rangle)\\
call\_forward(\langle o,l\rangle,\langle l'\rangle) & \leftarrow & [\mathsf{type}(o,\mathsf{Mover})], Mover.forward(\langle l\rangle,\langle l'\rangle)\\
Mover.forward(\langle l\rangle,\langle l''\rangle) & \leftarrow & [true], l' := l.next, l'' := l'.next
\end{array}
$$

The conditional statement in method main *is translated into 4 rules defined by procedures if and if$_2$. Since only simple comparisons are allowed as guards in the RBR representation, the additional rules defined in procedure* if$_2$ *are required to represent the disjunctive condition. The loops in methods* dup *and* len

```
1: proc analysis(m,CP,D)
2:    Q = ∅
3:    L^D=∅
4:    Q.add(m,CP)
5:    while (¬Q.empty()) do
6:       (p,CP)=Q.extract_first()
7:       process_analysis(p,CP,D)

8: function get_proc_answer(p,CP,D)
9:    CP' = CP
10:   AP' = ⊥
11:   if (L^D.exists(p)) then
12:      (CP^L ↦ AP^L)= L^D.get(p)
13:      if (CP ⊑ CP^L) then
14:         return AP^L
15:      CP' = CP ⊔ CP^L
16:      AP' = AP^L
17:   Q.add(p,CP')
18:   L^D.update(p,CP' ↦ AP')
19:   return AP'

20: proc invalidate_callers(p)
21:   for all (p' in callers(p)) do
22:      if (L^D.exists(p')) then
23:         (CP' ↦ AP')=L^D.get(p')
24:         Q.add(p',CP')

25: proc process_analysis(p,CP,D)
26:   AP=⊥
27:   for all (R_i : p ← [b_i0], b_i1, ..., b_in) do
28:      ST=extend(CP,vars(R_i),D)
29:      for each (b_ij, 0 ≤ j ≤ n) do
30:         if (b_ij = q(_,_)) then
31:            CP'=restrict(ST,vars(b_ij),D)
32:            ST'=get_proc_answer(b_ij,CP',D)
33:            ST'=extend(ST',vars(R_i),D)
34:         else
35:            ST'= τ(b_ij, ST, D)
36:         ST= ST ⊓ ST'
37:      AP = AP ⊔ ST
38:   (CP^L ↦ AP^L)=L^D.get(p)
39:   if (AP ⋢ AP^L) then
40:      invalidate_callers(p)
41:      L^D.update(p,CP ↦ AP)
```

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

are translated into recursive procedures, defined by a rule guarded by the loop condition and another one by its negation. The RBR of the program is built during the execution of the RTA analysis which allows resolving statically virtual invocations. In particular, we create dispatch *rules (e.g., call_forward) that are guarded by type conditions which cover all possible runtime types. Note that the dispatch rule defined by procedure call_forward does not include a call to* Mover2.forward *because the delta is not applied yet.*

3.2. A Global Fixed-Point Analysis Engine

Algorithm 1 presents an event-based global fixed-point analysis engine for the RBR, similar to other worklist algorithms [26]. The analysis is based on the theory of *abstract interpretation* [11], which we briefly summarize below. In abstract interpretation, the program execution is simulated using a *description (or abstract) domain* D which is simpler than its corresponding *concrete domain* C. The algorithm is parametric w.r.t. the abstract domain D that describes some property of interest. Description (or abstract) values and sets of concrete values are related by an *abstraction* function $\alpha : 2^C \to D$, and a *concretization* function $\gamma : D \to 2^C$. The pair $\langle \alpha, \gamma \rangle$ forms a Galois connection. The concrete and

abstract domains must be related in such a way that the following condition holds [11]:

$$\forall x{\in}2^C, \forall y{\in}D: \ (\alpha(x) \sqsubseteq y) \Longleftrightarrow (x \subseteq \gamma(y))$$

In general $\sqsubseteq$ is induced by $\subseteq$ and α. In order to guarantee termination of analysis, it is required that the elements in D be ascending chain finite under the partial order $\sqsubseteq$. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^C in a precise sense.

Each abstract domain comes equipped with a *transfer function* (denoted τ) which provides an abstraction of the behaviour of all basic instructions according to the corresponding abstract domain. The transfer function receives as input the next instruction to be executed, the current abstraction, and the abstract domain under consideration. It returns the new abstraction which corresponds to the state where the instruction has already been executed. Function $\mathsf{restrict}(ST, V, D)$ projects an abstraction ST onto the variables in the set V w.r.t. domain D, and $\mathsf{extend}(ST, V, D)$ extends the abstraction ST to the variables in V w.r.t. domain D.

The analysis results for a procedure are computed with respect to a specific calling pattern CP, which is a description of the input data (e.g., the input list is of type List) 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 a 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. Let us show these concepts by means of an example

Example 3.2 (restrict, extend)**.** *The abstract domain "*size*" infers* size constraints *which determine how the size of data is modified along the program's execution. For objects, this domain abstracts them to their maximal path length [35]. Integers are abstracted to their values. In general, elements in the abstract domain are constraints on the sizes of program variables. For example, if* b *and* s *are variables of type* List*, the constraint* $b \geq s+1$ *means that the length of list* b *is greater than the length of* s*. Let us consider method* foo *defined as follows, which calls method* Mover.forward *shown in Figure 1.*

```
void foo (Mover o, List s) {
  List b = new List(); b.next = s;
  List a = new List(); a.next = b;
ⓑ List c = o.forward(a);
  ...}
```

$CP_{\mathsf{foo}} \equiv \{o = 1, s \geq 0\}$

$ST_1 \equiv \{a \geq b+1, b \geq s+1, s \geq 0\}$

$CP_{\mathsf{forward}} \mapsto AP_{\mathsf{forward}} \equiv \{l \geq 2\} \mapsto \{l{\geq}r{+}2, r{\geq}0\}$

$ST_2 \equiv \{a \geq c+2, b \geq s+1, s \geq 0, c \geq 0\}$

Code for foo *has been annotated with the abstract state for* size *abstract domain at relevant program points. The analysis of* foo *starts from the call pattern* CP_{foo} *which states that the size of object* o *is 1 and that the length of the list is non-negative. At program point* ⓑ*, function* restrict *is used to project abstract information from* ST_1 *to the parameters of* forward*.* restrict *first projects* ST_1 *in terms of* a*, resulting in* $\{a \geq 2\}$*, and then renames* a *to* l*, the variable in the*

definition of forward*, generating* $CP_{\mathsf{forward}} \equiv \{l \geq 2\}$*. After analyzing* forward*, function* extend *projects and extends* AP_{forward}*, expressed in terms of* r*, back to the abstract state* ST_2*, renaming the result to variable* c*.*

The algorithm uses two global data structures: (1) the *local answer table* $\mathcal{L}^D$ for domain D, 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 given call pattern CP. The analysis of a method is carried out in *process_analysis*, where we analyze all rules defining a procedure in Line 27 (L27 for short) by traversing the instructions in its body from left to right (L29). Here, b_{i0} stands for the rule guard, which is handled together with the rest of instructions. When the instruction is not a procedure call, we abstract it according to the abstract domain (L35). As usual, the abstract description obtained from one instruction is conjoined with the previously computed one (L36). The analysis results obtained from the different rules which define a procedure are joined together (L37). When the instruction is a procedure call (L30), *get_proc_answer* first checks if a previously computed answer exists in $\mathcal{L}$ (L11). We assume that *get* returns the answer for a given call properly renamed w.r.t. the arguments in the call (L12). If the existing calling pattern is general enough (L13), we just use the previous answer (L14). Otherwise, since the algorithm is monovariant, we join the calling patterns (L15) and analyze again the corresponding method (L17). At the end of *process_analysis*, if the answer for p has changed (L39), i.e., a fixed point has not been reached, we need to invalidate the information for all rules that invoke p (calling *invalidate_callers* in L40), and update the local answer table $\mathcal{L}$ (L41). *invalidate_callers* adds to $\mathcal{Q}$ those methods that invoke p directly, denoted *callers(p)* (L21), and have an entry in $\mathcal{L}$ (L22).

Observe that the *analysis* engine adds entries to $\mathcal{Q}$ during its execution when (i) a rule must be analyzed for a given calling pattern in L17, either because there was not an answer for it or because it had been analyzed for a less general calling pattern and (ii) when the answer of a rule invoked from it changes (L24). The execution finishes when there are no more events to process in $\mathcal{Q}$ (L5).

Example 3.3 (algorithm 1). *The following table shows some relevant states of the execution of Algorithm 1 when analyzing method* main *of the running example w.r.t. the calling pattern* $CP \equiv \{l$:{List}$\}$ *for the domain "*class*". In the* class *domain, an abstract value represents the set of classes that the corresponding variable can be typed to. Thus, the CP indicates that the type of the input list* l *is* List*.*

(1)	$\mathcal{Q}$: *(*main,{l:{List}}*)* $\mathcal{L}$: *(*main,{l:{List}} $\mapsto \bot$*)*
(2)	$\mathcal{Q}$: *(*len,{l:{List}}*),(*mod,{l:{List}}*)* $\mathcal{L}$: *(*main,{l:{List}} $\mapsto \bot$*), (*len,{l:{List}} $\mapsto \bot$*), (*mod,{l:{List}} $\mapsto \bot$*)*
(3)	$\mathcal{Q}$: *(*mod,{l:{List}}*)* $\mathcal{L}$: *(*main,{l:{List}} $\mapsto \bot$*), (*len,{l:{List}} $\mapsto \bot$*), (*mod,{l:{List}} $\mapsto \bot$*),* *(*get,{l:{List}} $\mapsto \bot$*), (*dup,{o:$\bot$, l:{List}} $\mapsto \bot$*)*
(4)	$\mathcal{Q}$: *(*get,{l:{List}}*), (*dup,{o:$\bot$,l:{List}}*)* $\mathcal{L}$: *(*main,{l:{List}} $\mapsto \bot$*), (*len,{l:{List}} $\mapsto \bot$*), (*mod,{l:{List}} $\mapsto \bot$*),* *(*get,{l:{List}} $\mapsto$ {r:{Mover}}*), (*dup,{o:$\bot$, l:{List}} $\mapsto \bot$*)*
(5)	$\mathcal{Q}$: *(*dup,{o:$\bot$,l:{List}}*), (*mod,{l:{List}}*)* $\mathcal{L}$: *(*main,{l:{List}} $\mapsto \bot$*), (*len,{l:{List}} $\mapsto \bot$*), (*mod,{l:{List}} $\mapsto \bot$*),* *(*get,{l:{List}} $\mapsto$ {r:{Mover}}*), (*dup,{o:$\bot$, l:{List}} $\mapsto \bot$*)*
(6)	$\mathcal{Q}$: *(*mod,{l:{List}}*), (*Mover.forward,{l:{List}}*),* *(*dup,{o:{Mover.forward}, l:{List}}*)* $\mathcal{L}$: *(*main,{l:{List}} $\mapsto \bot$*), (*len,{l:{List}} $\mapsto \bot$*), (*mod,{l:{List}} $\mapsto \bot$*),* *(*get,{l:{List}} $\mapsto$ {r:{Mover}}*), (*dup,{o:$\bot$, l:{List}} $\mapsto \bot$*),* *(*Mover.forward,{l:{List}} $\mapsto \bot$*)*
⋮	⋮

When explaining the example, internal rules are ignored because they do not add any relevant information to the class *analysis. Relevant iterations proceed as follows: At iteration (1), method* main *is analyzed using the CP* {l:{List}}*. Method* main *calls* len *and* mod *which are added to* $\mathcal{Q}$ *and to* $\mathcal{L}$ *at iteration (2), see L17 and L18 of Algorithm 1, with the default AP,* $\bot$*. At (3), method* mod *is analyzed, adding to* $\mathcal{L}$ *methods* get *and* dup*. The call to* forward *is not resolved at (4) and (5) because the callgraph is being built during class analysis and the type of* o *is not known yet. Iteration (4) corresponds to the analysis of method* get *where a greater AP for* get *is found (before it was* $\bot$*). Thus, according to* invalidate_callers*,* mod *is added again to* $\mathcal{Q}$*. At iteration (5),* dup *is analyzed. Iteration (6) analyzes* mod *and the dynamic dispatching of* forward *can be solved, thus the calls to* Mover.forward *and to* dup *are added to* $\mathcal{Q}$*. The algorithm iterates until a fixpoint is reached (the result is shown later in Example 4.2).*

4. A Generic Multi-Domain Incremental Fixed-Point Analyzer

This section introduces a generic multi-domain incremental fixed-point engine which, given a change and the previous analysis results, is able to reconstruct the new analysis results for all domains. The algorithm relies in the notion of *method summary* which contains the analysis information that has been computed globally in a non-incremental way.

4.1. Method Summary for Global Properties

We now provide a generic definition for the notion of method summary on which our fixed-point algorithm relies.

Definition 4.1 (method summary). *Given a method $m(\bar{x},\bar{y})$ and a set of domains $\mathcal{D}$ applied over $m(\bar{x},\bar{y})$, a* method summary *for m is defined as a set of pairs $CP_D \mapsto AP_D$ for all $D \in \mathcal{D}$, where CP_D is the calling pattern used for analyzing m and AP_D is the answer pattern obtained by the analysis of m.*

Therefore, the method summary simply comprises the final analysis results obtained for a number of domains.

Example 4.2 (method summaries). *For the running example, we will consider two domains: (1)* class*, which determines the instantiation types of objects and hence the code that must be analyzed in the next steps, refining the information obtained by RTA. This domain is finite since it only contains the classes available in the program at run-time; and, (2)* size*, as defined in Example 3.2. The following method summaries are obtained by applying Algorithm 1 on the RBR in Example 3.1 by first performing the class and then the size analysis. We use* r *to represent the return value:*

Method		Summaries
void main (List l)	*class*	$\{l$:{List}$\} \mapsto \bot$
	size	$\{l \geq 0\} \mapsto \{l \geq 0\}$
void mod (List l)	*class*	$\{l$:{List}$\} \mapsto \bot$
	size	$\{l \geq 2\} \mapsto \{l \geq 2\}$
Mover get (List l)	*class*	$\{l$:{List}$\} \mapsto \{r$:{Mover}$\}$
	size	$\{l \geq 2\} \mapsto \{l \geq 2, r = 1\}$
void dup (Mover o, List l)	*class*	$\{o$:{Mover}$, l$:{List}$\} \mapsto \bot$
	size	$\{o = 1, l \geq 0\} \mapsto \{o = 1, l \geq 0\}$
int len (List l)	*class*	$\{l$:{List}$\} \mapsto \bot$
	size	$\{l \geq 0\} \mapsto \{l \geq 0\}$
List forward (List l)	*class*	$\{l$:{List}$\} \mapsto \{r$:{List}$\}$
	size	$\{l \geq 2\} \mapsto \{l \geq r + 2, r \geq 0\}$

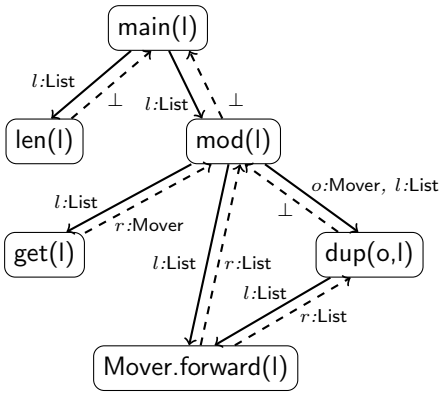
In the summary of dup*, observe that after creating the object in* get*, the type of the object is instantiated to* Mover *and its size is 1 (this means that it is not null). The inferred answer pattern for* main *states that the size of the returned list is greater than or equal to zero because of the* return *in the if statement. Note that although the size of the list in the summary of* main *is greater than or equal to zero, 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 at program point* ⓐ *in Figure 1. This size constraint is used as precondition of the (descendant) methods,* mod*,* get *and* forward*. It is important to understand the size relation obtained for method* forward *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, as we will see later.*

The summaries obtained for a program P and a particular domain D can also be represented as a *summaries graph*.

Definition 4.3 (summaries graph). *Given a program P and a global answer table $\mathcal{G}$ with entries of the form $(m, D) : CP \mapsto AP$, a* summaries graph *G_P is a directed graph represented by the pair $\langle N_s, E_s \rangle$ where:*

- *N_s is the set of nodes which consists of the set of methods m for which there is an entry $(m, D) : CP \mapsto AP$ stored in $\mathcal{G}$*
- *$E_s \subseteq N \times N$ is the set of edges, generated from $\mathcal{G}$ according to the following rules. For every entry $(m, D) : CP \mapsto AP \in \mathcal{G}$ and every node $n \in callers(m)$,*
  - *there is an edge from n to m labeled with CP, and*
  - *there is an edge from m to n labeled with AP*

Example 4.4 (summaries graph). *The summaries shown in Example 4.2 can be presented as a summaries graph. In the graph we only show the class analysis information to avoid cluttering the figure, but size information can be added in the same way. Normal lines represent CP's and dashed lines represent the AP's returned by methods.*



4.2. A Multi-Domain Incremental Analysis Engine

When performing incremental analysis, after a modification in the program, a change in the analysis results associated to one domain must *invalidate* the results previously inferred by subsequent *dependent domains*. We say that domain B depends on domain A when B uses information inferred by A. Dependencies can be represented by means of a *dependencies graph*.

Definition 4.5 (dependencies graph). *Given a set of domains $\mathcal{D}$, a* dependencies graph *is a directed acyclic graph represented by the pair $\langle N_d, E_d \rangle$ where N_d is the set of nodes and $E_d \subseteq N \times N$ is the set of edges, such that: N_d is the set of domains in $\mathcal{D}$, and, E_d has one edge when, for each pair $\langle A, B \rangle$ taken from the elements in $\mathcal{D}$, such that B uses the information inferred by A.*
Given a domain D, we will refer to the set of reachable domains starting from D (including D) as $\mathrm{dep}(D)$.

```
1:  proc incremental_fixpoint(m)
2:    P =[(m,D)]; C = ∅;
3:    G.invalidate(m,D)
4:    while (¬P.empty()) do
5:      R=∅
6:      (m', D_m') =P.extract_first()
7:      for each D in D_m' do
8:        R.add(m',D)
9:        (CP_m' ↦ AP_m')=G.get(m',D)
10:       noincr:analysis(m',CP_m',D)
11:     for all ((n,D) in R) do
12:       (CP_n^G ↦ AP_n^G)=G.get(n,D)
13:       (CP_n^L ↦ AP_n^L)=L^D.get(n,D)
14:       if (AP_n^L ⋤ AP_n^G)) then
15:         P.add_dom(callers(n),dep(D))
16:       G.update(n,CP_n^L↦AP_n^L⊔AP_n^G)
17:     C.add(R);
18:     P.remove(R)
19: function get_proc_answer(p,CP,D)
20:   CP'=CP
21:   if (is_method(p) ∧ G.exists(p,D))
      then
22:     (CP^G ↦ AP^G)=G.get(p,D)
23:     if (G.valid(p,D)) then
24:       if (CP ⊑ CP^G) then
25:         return AP^G
26:       else
27:         G.invalidate(p,dep(D))
28:         P.add_dom({p},dep(D))
29:     CP'=CP ⊔ CP^G
30:   R.add((p,D))
31:   AP=noincr:get_proc_answer(p,CP',D)
32:   return AP
```

Algorithm 2: Generic multi-domain incremental fixed-point algorithm.

Example 4.6. *Given the two domains described in Example 4.2, the dependencies graph contains two nodes, class and size, and one edge from class to size because the class analysis determines the code to be analyzed by the size analysis.*

Note that the algorithm we will propose is able to handle multiple pre-analyses as long as the dependencies among analyses form a directed acyclic graph, which is generally the case. Note that if the dependencies form cycles, the same pre-analysis may need to be recomputed multiple times even in the non-incremental scenario. This is most often not the case since analyses are typically performed in a fixed order which avoids the need of recomputing analyses.

Algorithm 2 presents the extensions required to make Algorithm 1 multi-domain and incremental by relying on the summaries in Definition 4.1 and the dependencies between domains defined in the dependencies graph. We use the notation *noincr:m* to refer to a procedure m defined in Algorithm 1 (see L10 and L31 in Algorithm 2) and *incr:m* to refer to a procedure m defined in Algorithm 2. The incremental algorithm uses method *noincr:analysis* of Algorithm 1 (and its data structures) and two implementations of *get_proc_answer*, *noincr:get_proc_answer* and *incr:get_proc_answer*. In L32 of *noincr:process_analysis* of Algorithm 1, we invoke *incr:get_proc_answer* instead of *noincr:get_proc_answer*. The remaining of this section explains how the algorithm works and its soundness.

In contrast to other approaches [20], 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. Algorithm 2 uses three global data structures: (1) the *global answer table* $\mathcal{G}$, which contains the set of summaries for all previously analyzed methods; (2) the *queue of pending events* $\mathcal{P}$, which is formed by pairs of the form $(m, \mathcal{D})$ where $\mathcal{D}$ is the list of domains for which m must be reanalyzed in the order stated in Figure 2; (3) a list, $\mathcal{R}$, to store all methods that have been reanalyzed in the current iteration of the while loop in L4; and (4) another list, $\mathcal{C}$, to store all methods that have belonged to $\mathcal{P}$ along the execution of the algorithm, i.e., the set of methods that have been reanalyzed at least for one domain. The latter data structure is built only for its use in the second phase of cost analysis (Section 6). The main goal of the incremental algorithm is, starting from the modified method, to propagate the new information obtained from the modification.

Let us explain how the new information propagation occurs by using the notion of *affected edge*. We start by providing some notation. Let P_0 be the initial program, and P_1 the program after the modification of method m. We will use G_{P_0} and G_{P_1} to represent the summaries graphs of P_0 and P_1 and $\mathcal{G}_{P_0}$ and $\mathcal{G}_{P_1}$ to refer to the global answer tables generated by the non-incremental algorithm (Algorithm 1) for P_0 and P_1, respectively.

Definition 4.7 (affected edge). *Let G_{P_0} and G_{P_1} be the summaries graphs for programs P_0 and P_1, respectively. We say that an edge from node n to node m in G_{P_1}, labeled with abstract state ST_{P_1}, is* affected *by the change when either:*

- *Node m exists in G_{P_1} but not in G_{P_0}, or*
- *The corresponding edge from P_0, labeled ST_{P_0}, fulfills $ST_{P_0} \not\supseteq ST_{P_1}$*

Definition 4.8 (affected method). *Let G_{P_0} and G_{P_1} be the summaries graphs for programs P_0 and P_1, respectively. We say that a method m is* affected *by the change when there is an affected edge that goes to m.*

Example 4.9 (affected edge, method). *Let us consider the application of the delta* SingleOrDoubleStep *to the core module defined by class* C*. After applying the delta, we have a new implementation of method* get*. The answer pattern obtained by the analysis of the new version of* get *is $AP_1 \equiv \{r : \{\mathsf{Mover}, \mathsf{Mover2}\}\}$, which is not contained in the previous answer pattern $AP_0 \equiv \{r : \{\mathsf{Mover}\}\}$, i.e. $AP_1 \not\sqsubseteq AP_0$. This new answer pattern returned by* get *creates an affected edge from* get *to* mod*, and, consequently,* mod *is an affected method that will have to be reanalyzed.*

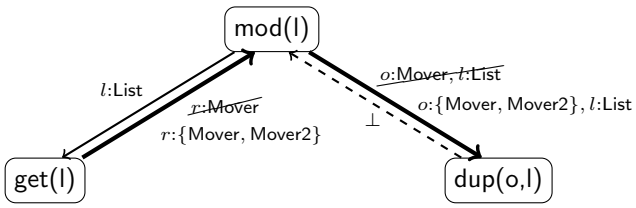
The goal of the algorithm is to start the analysis of P_1 from the changed method and reanalyze only those methods that have been affected by the change. Note that the aim is to build $\mathcal{G}_{P_1}$ (we do not have it a priori). Affected methods are either descendants or ancestors (transitively) of the changed method and information if propagated as follows:

Descendants. L7-10 take care of reanalyzing those methods that are pending to be reanalyzed. We use the *CP* stored in $\mathcal{G}$ to initiate the analysis (L9-L10). Besides, all reanalyzed methods are added to $\mathcal{R}$ in L8 in order to later (L11) decide the recomputation that must be done (see the previous ancestors paragraph). The call *noincr:analysis* (L10) reanalyzes m for a particular domain by calling Algorithm 1. During the execution of *noincr:analysis* (Algorithm 1) those methods reachable from m (descendants) whose answer for such domain must be recomputed will also be reanalyzed. Observe that the execution of *noincr:analysis* uses *incr:get_proc_answer* instead of *noincr:get_proc_answer*. The new function *get_proc_answer* in Algorithm 2 differs from the one in Algorithm 1 in that, for method calls, it tries to reuse an existing answer from the method summary (stored in $\mathcal{G}$) (L21) if its calling pattern is general enough and the entry has not been invalidated (L23-24). Otherwise, both calling patterns are joined (L29) and the pair of method signature and domain is added in L30 to the list $\mathcal{R}$. Entries of the method summary for dependent domains are invalidated (L27) and added to $\mathcal{P}$ (L28). If there is no summary for the method, or if it is an intermediate procedure of a method (L21), the function *noincr:get_proc_answer* of Algorithm 1 is invoked, and it analyzes m in the non-incremental way (L30). As we have seen in the non-incremental analysis, the analysis of one method may produce a new answer that must be propagated to its callers, by means of *noincr:invalidate_callers* (L40 of Algorithm 1). Observe that, in the incremental case, $\mathcal{L}$ only contains information regarding descendants of m which have been reanalyzed. Therefore, *noincr:invalidate_callers* only invalidates methods which have been reanalyzed in the current call to *noincr:analysis* (L22 of Algorithm 1). The execution of *noincr:analysis* finishes when no new information is propagated and a fixpoint for the reanalyzed methods is reached. This process is repeated for all domains for the considered method (L7).

Ancestors. When the call *noincr:analysis* finishes (L10), reanalyzed methods 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. Initially, $\mathcal{P}$ contains an entry for the changed method and all domains (L2). It is later updated in L15 as follows. For each element in $\mathcal{R}$, we know that it is reanalyzed for such domain (L11). We compare the new answer $AP_n^{\mathcal{L}}$ with the one in the summary $AP_n^{\mathcal{G}}$ (L12-14). If the new one is not contained in the previous one (L14), we have an affected method and thus we need to reanalyze all methods that invoke m (L15) for such domain and its dependent domains. However, some methods that invoke m may have been already reanalyzed in this iteration (during L7-10) and, therefore, they do not need to be reanalyzed again. All methods in $\mathcal{R}$ are removed from $\mathcal{P}$ in L18. Finally, the fixed point is reached when there are no more methods to analyze in $\mathcal{P}$.

Example 4.10 (descendants and ancestors). *The modification described in delta* SingleOrDoubleStep *launches the execution of* incremental_fixpoint*(*get*). Let us see how the new information for the class domain is propagated by Al-*

gorithm 1. The execution of the algorithm starts by adding get *to* $\mathcal{P}$ *and invalidating its results (L2-3) and L8-10 analyze the new version of* get*. The class analysis result for get stored in* $\mathcal{G}$ *does not cover the newly generated answer pattern, detailed in Example 4.9, thus all direct ancestors of* get*, that is* mod*, are affected methods and must be reanalyzed (L14-15). During the reanalysis of* mod*, a call to* dup *with a new call pattern to method is found. Observe that variable* o *can now be of types* {Mover, Mover2}*. This new call pattern forces the reanalysis of* dup *(L24-28), which is a descendant of* mod*, also affected by the change. The propagation of the new class information can be graphically seen in the portion of the summaries graph that corresponds to the affected methods. Thick edges denote the propagation of the new information.*



Now let us see that all affected methods are reachable from the modified method by an *affected path*.

Definition 4.11 (affected path). *An* affected path *in a summaries graph is a path formed by affected edges.*

Example 4.12. *The affected path in Example 4.10 is made of the thick edges.*

Proposition 4.13. *Given a program* P*, a domain* D *and* G_{P_1}*, all affected methods of* G_{P_1} *are reachable from the modified method* m *by an affected path.*

Proof. We use the correctness result of Algorithm 1, which is similar to other worklist algorithms ([26], Section 6.1). Algorithm 1 returns the least fixed point of the analysis regardless of the order in which events are processed. Given a method n, Algorithm 1 uses as information for analyzing n the code of n plus the labels of the incoming edges to n in the summaries graph. The result of the analysis of n is the set of outgoing edges from node n in the summaries graph. Therefore, (1) if the code of n has not changed and the incoming edges to n in the summaries graph are not affected, then, due to the monotonicity of the analysis, Algorithm 1 will not produce outgoing affected edges from n. Furthermore, this can be generalized to a set of nodes N. Let I be the set of those edges from nodes not in N to nodes in N. (2) If none of the nodes in N has changed and the edges in I are not affected, then Algorithm 1 will produce no affected outgoing edges from any node in N.

Now, let us prove the proposition by contradicting the following:

> *(*) "There exists an affected node* n *not reachable from the node corresponding to the modified method* m *by an affected path."*

If n is an affected node, then, by Definition 4.7, there is at least one affected edge from another method n' to n. n' must be an affected node, by (1). Furthermore, n' cannot be the modified method m, because it would contradict (*). Let N be the set of affected nodes connected to n by affected paths, including n, and M the rest of nodes in G_{P_1}. Also, $n' \in N$. Clearly, there are no affected edges from methods in M to methods in N, and $m \in M$. Thus, by (2), there are no affected outgoing edges from any node in N. This contradicts the fact that there is an affected edge from n' to n. Thus, node n is not reachable from m by an affected path cannot exist, contradicting (*).

□

Besides ensuring that all affected methods are reanalyzed we need to prove that their analysis gives sound result.

Proposition 4.14. *The loop in L4-18 of Algorithm 2 obtains correct analysis information for affected methods.*

Proof (sketch). We consider one analysis domain D. The extension to several domains will be studied in Proposition 4.16. First, it must be proved that one iteration of the loop in L4-18 of Algorithm 2 correctly propagates analysis results to both $callers(m)$ and methods called by m. The latter is handled by calling *noincr:get_proc_answer* in L32 of Algorithm 1, reanalyzing methods called by m if either they where not analyzed before, or the new call pattern for them is not contained in the results stored in $\mathcal{G}$. On the other hand, methods in $callers(m)$ are added to $\mathcal{P}$ in L15 of Algorithm 2 if the analysis information obtained for m is not included in the information already stored in $\mathcal{G}$. The loop in L4-18 transitively processes affected methods, which are reachable from m by affected paths (Proposition 4.13). When the loop terminates, all affected methods are reanalyzed and $\mathcal{G}$ is updated with correct analysis information. The complete proof can be found in the online version. □

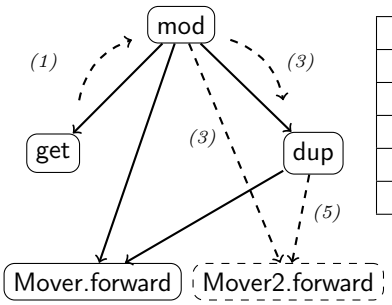
The extension of Proposition 4.13 and Proposition 4.14 is straightforward for global answer tables which are obtained after executing Algorithm 2, in order to handle consecutive changes.

Up to now, we have seen how the algorithm works incrementally for one domain. Now, let us see how the algorithm is able to interleave the computation of multiple domains.

Multi-domain. When a method changes, its code must be reanalyzed with respect to all domains. This is done by the algorithm in L2-3 where we add to $\mathcal{P}$ the modified method for all domains and invalidate its entries in $\mathcal{G}$. 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 (see Definition 4.5). This is handled in the algorithm by means of $\mathcal{G}$ 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 method p. Invalidated entries must be reanalyzed. This occurs in the algorithm when the entries to be reanalyzed are added to $\mathcal{P}$ in L15 and L28. Note that class analysis determines the CFG of the program being analyzed. Hence, after a change, the callers information used in L15 by Algorithm 2 is recomputed. To that end, *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 as determined by the domains dependencies graph. If an entry does not exist, it adds $(m, \{D\})$.

Example 4.15 (algorithm 2). *At this point, let us explain how Algorithm 2 handles the modification of delta* SingleOrDoubleStep *and interleaves the computation of class and size analysis. The application of the delta launches the following iterations of the loop in L7 of Algorithm 2:*



Iter	$\mathcal{P}$
(1)	$\{(\mathsf{get}, class), (\mathsf{get}, size)\}$
(2)	$\{(\mathsf{get}, size), (\mathsf{mod}, class), (\mathsf{mod}, size)\}$
(3)	$\{(\mathsf{mod}, class), (\mathsf{mod}, size)\}$
(4)	$\{(\mathsf{mod}, size), (\mathsf{dup}, size)\}$
(5)	$\{(\mathsf{dup}, size)\}$

The left figure graphically represents the iterations of the incremental analysis algorithm. Arrows represent parental relations among methods. Dashed lines (arrows and boxes) indicate the recomputed information due to the application of the delta. Some arrows are labeled with the iteration number that computes it. The table to the right shows the contents of $\mathcal{P}$ at each iteration. Intuitively, the algorithm proceeds as follows:

(1) analysis(get,_,*class*) *of Algorithm 1 is executed. As it is explained in Example 4.9 and Example 4.10, the new answer pattern for* get *forces the reanalysis of its direct ancestors, namely* mod *is added to $\mathcal{P}$ for domains* dep(class). *This is shown graphically by a dashed arrow from* get *to* mod *labeled with (1).*

(2) Afterwards, size *analysis for* get *does not propagate new information to its callers, hence no arrow is labeled with (2).*

(3) analysis(mod,_,*class*) *is launched. During class analysis, calls to* dup *and* forward *are found. According to the new results obtained for* get*, variable* o *can now be of types* {Mover, Mover2}*. Due to this polymorphism,*

an invocation to Mover2.forward[1] *is found in addition to the previous* Mover.forward. *Since there are no entries for* Mover2.forward *in* $\mathcal{G}$, *it is analyzed during the execution of* analysis *and added to* $\mathcal{G}$ *for the domain* class. *Besides, the entry for* dup *in* $\mathcal{G}$ *has* $CP_1 \equiv \{o{:}\{\mathsf{Mover}\}, l{:}\{\mathsf{List}\}\}$ *and thus it does not cover the new* $CP_2 \equiv \{o{:}\{\mathsf{Mover}, \mathsf{Mover2}\}, l{:}\{\mathsf{List}\}\}$. *Both CPs are joined (resulting in* CP_2*) and* dup *is reanalyzed for the class domain w.r.t.* CP_2. *During the class analysis of* dup, *a new call to* Mover2.forward *is found, but now* $\mathcal{G}$ *contains valid information for* Mover2.forward, *and thus the AP stored in* $\mathcal{G}$ *can be directly used. As* dup *has been reanalyzed for domain* class, *entries for* dup *for* dep(class) *in* $\mathcal{G}$ *are invalidated and added to* $\mathcal{P}$. *Note that (*dup,class*) is not added to* $\mathcal{P}$ *because it has been handled as a descendant of* mod.

(4) Similarly to (3), during the analysis of mod *for the* size *domain, the entry for (*Mover2.forward,*size) is added to* $\mathcal{G}$.

(5) Polymorphism of the call to forward *forces that the size relations for procedure* while *change. Now, we combine the size results of* Mover.forward *and* Mover2.forward, *since any of the two methods can be executed within the loop.*

At the end of (5) 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* Mover.forward *have not required reanalysis for any domain. The following table shows the summaries that have changed w.r.t. the ones in Ex. 4.2:*

Method		Summaries
Mover get(l)	*class*	$\{l{:}\{\mathsf{List}\}\} \mapsto \{r{:}\{\mathsf{Mover}, \mathsf{Mover2}\}\}$
	size	$\{l \geq 2\} \mapsto \{l \geq 2, r = 1\}$
void dup(o,l)	*class*	$\{o{:}\{\mathsf{Mover}, \mathsf{Mover2}\}, l{:}\{\mathsf{List}\}\} \mapsto \bot$
	size	$\{o = 1, l \geq 0\} \mapsto \{o = 1, l \geq 0\}$
List Mover2.forward(l)	*class*	$\{l{:}\{\mathsf{List}\}\} \mapsto \{r{:}\{\mathsf{List}\}\}$
	size	$\{l \geq 2\} \mapsto \{l \geq r + 1, r \geq 0\}$

Proposition 4.16. *The loop in L4-18 of Algorithm 2 obtains correct analysis information for affected methods.*

Proof. Proving the correctness of the algorithm to handle multiple domains is a straightforward extension of the proof of Proposition 4.14. All methods that are reanalyzed must be invalidated and reanalyzed not only for the analyzed domain, but also for all dependent domains. $\mathcal{G}$ entries are invalidated by the Algorithm *alg:noincr-algorithm* in L27 and L15, which add the method to the

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

queue of pending events $\mathcal{P}$, not only the current domain (D), but also all dependant domains using function *dep(D)*. The invalidation of one entry in $\mathcal{G}$ forces its reanalysis for all dependant domains and the behaviour of the algorithm is analogous to handling only one domain. □

Theorem 4.17 (correctness). Incremental_fixpoint(m) *terminates and returns a correct* $\mathcal{G}$ *for all methods.*

Proof (sketch). Correctness is an immediate consequence of Propositions 4.13 and 4.16. Termination is guaranteed by the ascending chain condition (ACC) of the abstract domains and the $\sqcup$ domain operator. L29 and L16 of Algorithm 2 and L15 of Algorithm 1 guarantee that the abstract states always ascend or stabilize in the analyzed domain. More details can be found in the online version. □

Final Remarks. For simplicity of the presentation, when there are multiple methods changed, changes will be handled one after another. The algorithm can be extended in order to handle sets of methods changed simultaneously, which may result in improved efficiency of the incremental analysis in some situations.

Regarding accuracy of the incremental versus non-incremental analyses, it must be mentioned that one of our design decisions when proposing Algorithm 2 was to recompute analysis information as efficiently as possible by reusing all information which is guaranteed to remain correct, and only recompute the analysis information which is potentially no longer correct. This results in improved efficiency but the results are potentially less accurate than those which may be obtained by analyzing the program from scratch. Though in our experience, the precision loss does not happen very often, the situations where it can occur are easy to identify during incremental analysis and can be used to inform the user that the analysis results may be improved by performing analysis from scratch. In particular, this occurs when the new call pattern for a method is strictly less general than the one for which analysis is stored. If that is the case, an analysis from scratch can be performed when requested by the user or at the end of the development phase.

5. Incremental Inference of Cost Relations

We now illustrate how the generic incremental fixed-point analysis engine is used for the generation of cost relations in the first phase of cost analysis (see Section 1).

5.1. Method Summary for Global Properties

All analysis required for cost analysis (class, sharing, acyclicity, ...) can be computed by using the generic fixed-point engine in Algorithm 1 for each domain. These pre-analyses are executed consecutively and the information inferred for one domain is used for analyzing subsequent domains. Figure 2

shows the *dependencies graph* between domains, where the order of execution and the dependencies between domains are shown. The following definition of *cost method summary* together with the dependencies is used by Algorithm 2 to perform incremental analysis.

Definition 5.1 (cost 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 *[34];*

(2) D=sharing*, where* $(x, y) \in AP_{sh}$ *(resp.* CP_{sh}*) means that* x *and* y *might share after (resp. before) executing* m *[32];*

(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 *[30];*

(4) D=constancy*, where* $x \in AP_{cn}$ *if the structure of* x *may have changed during the execution of* m *[17] (this analysis is context-insensitive);*

(5) D=size*, where* AP_{sz} *(resp.* CP_{sz}*) are a set of linear constraints describing the relation between the size of the variables* $\bar{x}$ *and* $\bar{y}$ *after (resp. before) executing* m *[4].*

Let us mention the most relevant issues related to the five points in the above definition and explain the domain dependencies in Figure 2. The class analysis information determines the types of objects and thus allows us to refine the information obtained by the RTA analysis for generating the CFG. For instance, assume that class B extends A, if we have a code like the leftmost two lines:

```
A o = new B();      call ← type(o,A),A.m()
o.m();              call ← type(o,B),B.m()
```

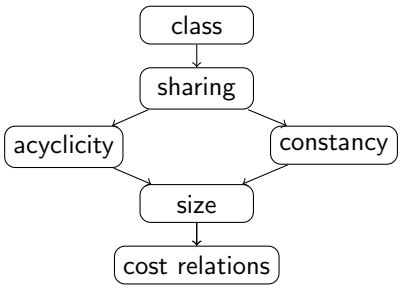


Figure 2: Domain dependencies

The RTA analysis had decided to load classes A and B and the virtual invocation had been decompiled into the two rules that appear to the right. Then, class analysis determines that o is of type A and the analysis will only infer information from the second rule. Regarding 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; 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. 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 [17]. 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. Finally, size relations are used to generate the *cost relations* whose definition is given in Section 5.2.

Example 5.2. *After applying the analysis using the domains defined in 5.1 to the running example, the only relevant information obtained is related to class and size domains. As it is usual in resource analysis, we assume that the input list for method* main *is not cyclic, and analyses for sharing, constancy and acyclicity guarantee that, after program execution, the list remains acyclic. Example 4.15 has showed the detailed execution of our incremental analysis engine for the domains used in cost analysis.*

5.2. Generation of Cost Relations

Given the size relations, the second phase of cost analysis (item 2 of Section 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 Section 6 and make the article self-contained, we recall 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 $\mathcal{M}$ maps an instruction b to a corresponding cost, denoted $\mathcal{M}(b)$), (2) abstract each basic instruction by a size constraint [3] and, (3) when we find a call to a method, the size constraint is the size relation for the answer pattern of the size summary.[2] The actual cost of executing 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. Given a RBR rule $r \equiv p(\bar{x},\bar{y})\leftarrow B$, we use the following functions: $calls(B)$ to obtain the set of elements $q(\bar{w},\bar{z})$ which are calls to procedures in B, and $instr(B)$ to refer to the set of elements of B which are other instructions. Given a rule r, its size relation φ_r is the conjunction of the size constraints of its instructions in $instr(B)$ (point 2 above) and of the size relations in the answer patterns of the summaries for $calls(B)$ (point 3 above).

Definition 5.3 (cost relation system [3]). *Given a cost model $\mathcal{M}$, a rule $r \equiv p(\bar{x},\bar{y})\leftarrow B$ in the program and its size relation φ_r, we generate the cost equation:*

$$\langle\ p(\bar{x}) = \sum_{b\in instr(B)} \mathcal{M}(b) + \sum_{q(\bar{w},\bar{z})\in calls(B)} q(\bar{w}),\quad \varphi_r\rangle$$

Given a program P, its cost relation system is obtained by applying the above transformation to all rules.

[2] Note that in order to obtain such size summaries we need all pre-analyses discussed above.

The above kind of CRs can be generated for any kind of cumulative cost that always increases along the execution of the program (e.g., number of executed instructions, number of objects created). However, certain cost criteria are non-cumulative (e.g., the peak of the memory consumption in the presence of garbage collection) and require non-standard CRs that capture such cost peaks (see [7]).

Example 5.4. *Consider the original program before applying the delta of the running example and the summaries in Example 4.2. In the example we will use a cost model which counts the number of instructions executed by the program, i.e.* $\mathcal{M}(b) = 1$ *(guards are not counted as instructions in the cost model). As an example, by applying the cost model to the rule* $mod(l)$ *we obtain that the number of instructions is 3. By applying Definition 5.3, the rule for method* mod *is transformed into the equation:* $mod(l) = 3 + get(l) + call\_forward(l) + dup(l')\ \{l' \geq 0, l \geq l' + 2\}$ *where the* 3 *stands for the three calls in the instruction and the size relations are obtained from the size summaries for the methods in Example 4.2. Irrelevant constraints on the input are omitted in the example, e.g. in* $get(l)$*, the constraint* $l \geq 2$ *is not shown in the CRs. The cost of the calls to the methods is defined by corresponding equations. For the running example, we get the following set of equations:*

$$
\begin{array}{ll}
mod(l) = 3 + get(l) + call\_forward(l) + dup(l') & \{l' \geq 0, l \geq l' + 2\} \\
get(l) = 1 & \{\} \\
dup(l) = 1 + while(l) & \{l \geq 0\} \\
while(l) = 3 + call\_forward(l) + while(l') & \{l' \geq 0, l \geq l' + 2\} \\
while(l) = 0 & \{l = 0\} \\
call\_forward(l) = 0 + Mover.forward(l) & \{l \geq 2\} \\
Mover.forward(l) = 2 & \{\}
\end{array}
$$

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 (e.g., the first argument o *in rule dup) because they are useless for solving the equations.*

After a modification in a program, when Algorithm 2 finishes, we need to generate new CRs for all methods which have been reanalyzed (i.e., those that belong to $\mathcal{C}$). This is because their size relations may have changed; and, besides, for the changed method its accumulated cost may have changed as well. The remaining ones do not require any change.

Example 5.5. *Considering the application of the delta* SingleOrDoubleStep *and the summaries in Example 4.15. When Algorithm 2 finishes* $\mathcal{C}$ *contains* {get, mod, dup, Mover2.forward}*. Thus, CRs for all those methods must be recomputed by using the standard CR generation as explained in Definition 5.3:*

$$
\begin{array}{ll}
mod(l) = 3 + get(l) + call\_forward(l) + dup(l') & \{l' \geq 0, \underline{l \geq l' + 1}\} \\
get(l) = 3 & \{\} \\
dup(l) = 1 + while(l) & \{l \geq 0\} \\
while(l) = 3 + call\_forward(l) + while(l') & \{l' \geq 0, \underline{l \geq l' + 1}\} \\
while(l) = 0 & \{l = 0\} \\
call\_forward(l) = 0 + Mover.forward(l) & \{l \geq 2\} \\
call\_forward(l) = 0 + Mover2.forward(l) & \{l \geq 1\} \\
Mover2.forward(l) = 1 & \{\}
\end{array}
$$

When comparing the equations with the ones in Example 5.4, we observe that after merging the size information gathered for the two implementations of forward*, in the new CR, we have to assume that the length of the list decreases (in the worst-case) by one. This corresponds to the underlined constraints. The cost relations for* main*,* len *and* Mover.forward *do not need to be updated.*

6. Incremental Inference of Upper Bounds

The third phase in cost analysis (see item 3 in Section 1) consists in transforming the CRs obtained in Section 4 into *cost functions* [2], 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. This is also the reason why computer algebra systems (CAS) cannot be used in general to transform CRs into cost functions (see [2]). In other words, CRs do not define functions, but rather relations and thus CAS cannot be used to solve them. Very shortly, the reasons why they are relations are (1) that CRs are highly non-deterministic: equations for the same relation are not required to be mutually exclusive. Even if the programming language is deterministic, size abstractions introduce a loss of precision: some guards which make the original program deterministic may not be observable when using the size of arguments instead of their actual values. (2) Besides, CRs may have constraints other than equalities, such as $l > l'$. When dealing with realistic programming languages which contain non-linear data structures, such as trees, it is often the case that size analysis does not produce exact results.

W.l.o.g., we make three simplifications for the sake of clarity of the presentation:

1) We focus on *upper bounds*, since the problem of *lower bounds* is dual [8].
2) We consider polynomial CRs. Note that the technique in [2] can solve also exponential and logarithmic relations. In the same way, our incremental analysis can be applied to these kinds of relations without requiring any modification other than the upper bound formula (UB_C in Definition 6.1) which gathers all components together.
3) We ignore base cases, because they are handled in a strictly simpler way than recursive equations. Thus, the same reasoning applies.

A precondition in order to apply the method of [2] (and thus also for our approach) is that the CR form an acyclic graph. Thus, after a change in the program, the CRs are generated and the precondition is checked.

In Section 6.1, 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). Section 6.2 presents an algorithm to support incremental inference in this step.

6.1. The Notion of Upper Bound Summary

Our starting point is the technique of [2] which proposes an automatic approach to obtaining UBs from CRs by 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. The following definition summarizes such solving process:

Definition 6.1 (upper bound [2]). *Consider a CR defined by n equations of the form:*

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

where the size relations *(φ) are linear constraints on variables $\overline{x}$ and $\overline{y}_i \cup \overline{y}$, An upper bound for $C(\overline{x})$ is $\mathrm{UB}_C(\overline{x}) = \#iter * \texttt{mexp}$ such that:*

1. *$\#iter$ is an upper bound on the number of recursive calls of C,*
2. *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),*
3. *$\texttt{ub}_i$ are upper bounds for each call $Y_i(\overline{y}_i)$,*
4. *for each recursive equation, we have that:*
   $\texttt{mexp}' = maximize(\texttt{exp}, \overline{x}, \psi, \varphi) + \sum_{i=1}^{k} maximize(\texttt{ub}_i, \overline{x}, \psi, \varphi)$
   where function $maximize(e, \overline{x}, \psi, \varphi)$ returns the maximization of e for ψ and φ w.r.t. the equation entry variables $\overline{x}$,
5. *if C has n recursive equations, $\texttt{mexp} = max(\texttt{mexp}'_1, ..., \texttt{mexp}'_n)$, where $\texttt{mexp}'_j$ is the maximized cost of equation j obtained in point 4, with $j = 1, \ldots, n$.*

Example 6.2. *Let us see the application of Definition 6.1 to the following CRs:*

$$\begin{array}{ll}
while_1(i) = 3 + while_2(j) + while_1(i') & \{i' > 0, i \geq i' + 1, i = j\} \\
while_1(i) = 0 & \{i = 0\} \\
while_2(j) = 5 + while(j') & \{j' < 10, j \geq j' - 1\} \\
while_2(j) = 0 & \{j = 10\}
\end{array}$$

In order to solve $while_2$, we obtain that an upper bound on the number of iterations is $\#iter = \mathsf{nat}(10-j)$ (point 1). In $while_2$, $\texttt{mexp}$ is simply the cost of executing one iteration of the loop, that is, 5, resulting in $UB_{while_2}(j) = 5*\mathsf{nat}(10-j)$. Let us continue with the UB of $while_1$. The maximum number of iterations of $while_1$ is determined by $\mathsf{nat}(i)$. Regarding point 5, $while_1$ has only

one recursive call, thus $\texttt{mexp} = \texttt{mexp}'$. *Now, so as to obtain* $\texttt{mexp}'$ *we have the cost executed by* $while_1$ *in one iteration plus the worst case of calling* $while_2$, *i.e.,* $UB_{while_1}(i) = \mathsf{nat}(i) * (3 + \mathtt{ub}_{while_2})$. *This maximization is done by using the size relations* $\varphi = \{i' > 0, i \geq i' + 1, i = j\}$ *and the invariant* $\psi = \{i_0 \geq i\}$ *(i_0 represents the value of i before entering the loop) to maximize the expression* $5{*}\mathsf{nat}(10{-}j)$ *with respect to i (points 2, 3, 4). The worst possible case for this expression is obtained when* $while_2$ *is called with* $i = 0$, *thus* $\mathtt{ub}_{while_2}$ *can be replaced by* $5{*}10$, *resulting in* $UB_{while_1}(i) = \mathsf{nat}(i) * (3 + 50)$.

Intuitively, given a recursive cost relation defined by several equations, a safe upper bound is obtained by inferring an upper bound on the number of iterations $\#iter$ (point 1 above) and then taking the maximal cost of all possible executions of the equations $\texttt{mexp}$ (point 5). Clearly, $\#iter * \texttt{mexp}$ is a sound upper bound. We use the techniques in [2] to infer $\#iter$ automatically. Such approach in complete when $\#iter$ is a linear expression and it is based on inferring a *ranking function* for each loop which bounds its number of iterations. Other techniques can adapted for non-linear polynomials [14]. The maximal cost $\texttt{mexp}$ is the maximum of the worst-case cost of executing each of the equations that define the relation in isolation (point 4). Observe that the process of obtaining such worst-case cost requires an invariant (ψ) generation phase (point 2) and then a maximization of expressions (point 3). We cannot make any incrementalization of these two parts because they are already locally obtained from the CRs, see the details of the techniques used in [2]. This step is again not complete (as inferring cost is undecidable) and thus it is not always computable (see [2]).

Example 6.3 (UB). *Let us apply the Definition 6.1 to solve the CR* mod *in Example 5.4 which has been obtained before applying the delta of Figure 1. The standalone CRs* Mover.forward *and* get *are already solved since they are not recursive. Next, we solve* while(l') *which is required in order to solve* dup. *The CR* while *contains one call to* Mover.forward *with constant cost. Its UB can be computed by multiplying the cost of one iteration by the maximum number of iterations. The cost of one iteration is 3 plus the cost of the call to get, 2. 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. Thus, the UB of the CR* while *is*

$$UB_{\mathsf{while}}(l'){=}3{+}2{*}\mathsf{nat}(l'/2).$$

The UB for dup *is directly obtained from* UB_{while} *resulting in the UB*

$$UB_{\mathsf{dup}}(l'){=}1{+}(3{+}2){*}\mathsf{nat}(l'/2).$$

Finally, mod *is not a recursive CR, so we do not need to infer number of iterations. The CR for* mod *calls* dup. *Thus, when computing an UB for* $\mathsf{mod}(l)$, *the UB for* dup *has to be maximized w.r.t. the entry variable* l, $\varphi = \{l \geq l'{+}2, l' \geq 0\}$ *and the invariant* $\psi = \{l_0{=}l, l_0{\geq}2\}$. *This results in*

$$UB_{\mathsf{mod}}(l){=}6{+}(1{+}5{*}\mathsf{nat}(l/2{-}1)),$$

where we can observe that the maximum cost of dup *within* mod *occurs when* $l' = l - 2$. *Finally, the cost of the* main *method is computed by adding* UB_{mod} *and* $UB_{\mathsf{len}}(l) = 2{+}3{*}\mathsf{nat}(l)$ *and its own cost, resulting in the expression:*

$$UB_{\mathsf{main}}(l) = 5{+}2{+}3 * \mathsf{nat}(l){+}7{+}5{*}\mathsf{nat}(l/2-1)$$

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, it is not possible to distinguish within an UB which part of the cost is associated to m. Thus, 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. In order to support incremental inference of UBs, we propose 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 Definition 6.1, we introduce the notion of *UB summary*.

Definition 6.4 (upper bound summary). *In the same conditions of Definition 6.1, an* UB summary *for* $C(\overline{x})$ *is a tuple* $\mathrm{UB}^{\mathsf{S}}_{\mathsf{C}}(\overline{x}) = \langle \#iter \cdot \mathtt{aexp}, \psi, \varphi \rangle$, *where* $\mathtt{aexp} = maximize(\mathtt{exp}, \psi, \varphi, \overline{x}){+}\sum_{i=1}^{k}[maximize(\mathsf{remove\_annot}(\mathtt{aub}_i), \psi, \varphi, \overline{x})]_{Y_i}$ *s.t.:*

- $\mathtt{aub}_i$ *is the annotated cost expression in the upper bound summary of* Y_i,
- *function* remove_annot *removes the annotations of an expression, and*
- $[e]_Y$ *is an annotation of* e *with the name of the relation* Y.

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, all nested annotations are deleted by function *remove_annot*.

Example 6.5 (UB summary). *The UB summaries for the CRs of Example 5.5 are:*

$$UB^{\mathsf{S}}_{\mathsf{while}}(l) = \langle (3 + [2]_{\mathsf{Mover.forward}(l')}) * \mathsf{nat}(l/2), \{l_0 \geq l + 2, l \geq 0\}, \{l = l'\} \rangle$$

$$UB^{\mathsf{S}}_{\mathsf{dup}}(l) = \langle 1 + [(3 + 2) * \mathsf{nat}(l/2)]_{\mathsf{while}(l')}, \{l_0 = l, l_0 \geq 0\}, \{l = l', l \geq 0\} \rangle$$

$$UB^{\mathsf{S}}_{\mathsf{mod}}(l) = \langle 3{+}[1]_{\mathsf{get}(l')}{+}[2]_{\mathsf{forward}(l'')}{+}[(1{+}5{*}\mathsf{nat}(l/2{-}1))]_{\mathsf{dup}(l''')}, \{l_0 = l, l_0 \geq 2\}, \{l = l', l = l'', l \geq l''' + 2, l''' \geq 0\} \rangle$$

$$UB^{\mathsf{S}}_{\mathsf{main}}(l) = \langle 5 + [2 + 3 * \mathsf{nat}(l)]_{\mathsf{len}(l')} + [7 + 5 * \mathsf{nat}(l/2 - 1)]_{\mathsf{mod}(l'')}, \{l_0 = l\}, \{l = l', l = l'', l \geq 2\} \rangle$$

Observe that the only difference with the UB of Example 6.3 is in the annotations.

6.2. 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 along the execution of Algorithm 2, named $\mathcal{C}$. 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 with m by the new maximized UB for m. Similarly to Algorithm 2, we use a flag in the summaries table $\mathcal{U}$ to indicate whether the content of an entry is valid or not. The intuitive 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 6.6 (upper bound skeleton). *In the same conditions of Definition 6.1, an upper bound skeleton for* $C(\overline{x})$ *is a tuple* $SK_C(\overline{x}) = \langle \#iter \cdot \mathtt{sexp}, \psi, \varphi \rangle$, *where* sexp *is the annotated expression* $\mathtt{exp} + \sum_{i=1}^{k} [\_]_{D_i}$ *and* _ *denotes any value. In what follows, function* 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 replacing the UBs of the relations invoked from m by their new maximized expressions (when needed). This is done by function *do_summary* of Algorithm 3.

Example 6.7 (skeleton). *The application of the delta in Figure 1 leads to the following skeletons for* mod *and* dup*:*

$$\begin{aligned}
SK_{\mathsf{while}}(l) &= \langle (3 + max([_]_{\mathsf{Mover.forward}(l')}, [_]_{\mathsf{Mover2.forward}(l')})) * \mathsf{nat}(l), \\
&\quad \{l_0 \geq l + 1, l \geq 0\}, \{l = l'\} \rangle \\
SK_{\mathsf{dup}}(l) &= \langle 1 + [_]_{\mathsf{while}(l')}, \{l = l_0, l_0 \geq 0\}, \{l = l', l \geq 0\} \rangle \\
SK_{\mathsf{mod}}(l) &= \langle 3 + [_]_{\mathsf{get}(l'')} + max([_]_{\mathsf{Mover.forward}(l'')}, [_]_{\mathsf{Mover2.forward}(l'')}) + [_]_{\mathsf{dup}(l''')}, \\
&\quad \{l_0 = l, l_0 \geq 2\}, \{l = l', l = l'', l \geq l''' + 1, l''' \geq 0\} \rangle
\end{aligned}$$

Note that the skeleton of dup *differs from the initial one because of the new implementation of* forward*. This leads to a different number of iterations of the loop (named #iter in Definition 6.1) which now is* $\mathsf{nat}(l)$ *instead of* $\mathsf{nat}(l/2)$*. Also we have introduced the* max *expression in the skeletons for* while *and* mod

```
1:  proc reconstruct_summaries()
2:    P = ∅
3:    for all m(x̄) in C do
4:      ⟨aexp_m, ψ, φ⟩=do_skeleton(m(x̄))
5:      U.update(m,⟨aexp_m, ψ, φ⟩)
6:      U.invalidate(m(x̄) ∪ ancestors(m))
7:      P.add(m(x̄) ∪ ancestors(m))
8:    for all m(x̄) in P do
9:      do_summary(m(x̄))
10:   U.validate_all()
11: function do_summary(m(x̄))
12:   ⟨aexp_m, ψ, φ⟩ = U.get(m)
13:   if m ∉ P then
14:     return remove_annot(aexp_m)
15:   for all [_]_p(ȳ) in aexp_m do
16:     if (m∈ C) ∨ U.is_invalid(p) then
17:       exp_p = do_summary(p(ȳ))
18:       exp_p = maximize(exp_p,x̄,ψ,φ)
19:       aexp_m.set_expr([_]_p,exp_p)
20:   P.remove(m);
21:   U.update(m,⟨aexp_m, ψ, φ⟩)
22:   return remove_annot(aexp_m)
```

Algorithm 3: Incremental Upper Bounds Algorithm

which accounts for the worst-case costs of both implementations of forward*. As* main *has not been reanalyzed, its skeleton will not be computed again.*

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 Definition 6.4. 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. We use aexp to refer to $\#iter \cdot \texttt{sexp}$. 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 6.8 (algorithm 3). *The application of the delta in Figure 1 forces the execution of* do_skeleton *for all methods in* $\mathcal{C} = \{$mod, dup, get, Mover2.forward$\}$. *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 (see Example 6.7). Then,* dup *is removed from* $\mathcal{P}$ *and we obtain:*

$$UB^{\mathsf{S}}_{\mathsf{while}}(l) = \langle (3 + max([2]_{\mathsf{Mover.forward}(l')}, [1]_{\mathsf{Mover2.forward}(l')})) * \mathsf{nat}(l), \{l_0 \geq l+1, l \geq 0\}, \{l = l'\}\rangle$$

$$UB^{\mathsf{S}}_{\mathsf{dup}}(l) = \langle 1 + [(3+2) * \mathsf{nat}(l)]_{\mathsf{while}(l')}, \{l = l_0, l_0 \geq 0\}, \{l = l', l \geq 0\}\rangle$$

$$UB^{\mathsf{S}}_{\mathsf{Mover2.forward}}(l) = \langle 1, \{l = l_0, l_0 \geq 1\}, \{\}\rangle$$

$$UB^{\mathsf{S}}_{\mathsf{get}}(l) = \langle 3, \{\}, \{\}\rangle$$

In order to obtain UB_{mod}*, we need to maximize* UB_{dup}*, which leads to:*

$$UB^{\mathsf{S}}_{\mathsf{mod}}(l) = \langle 3 + [3]_{\mathsf{get}(l')} + max([2]_{\mathsf{Mover.forward}(l'')}, [1]_{\mathsf{Mover2.forward}(l'')})) + [1 + 5 * \mathsf{nat}(l-1)]_{dup(l''')}, \{l_0 = l, l_0 \geq 2\}, \{l = l', l = l'', l \geq l''' + 1, l''' \geq 0\}\rangle$$

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* UB_{mod} *must be maximized, and* UB_{len} *can be reused:*

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

All in all, the incremental recomputation has avoided computing the skeleton (#iter, invariant) and one maximization for main*, and the summaries of* len *and* Mover.forward *remain the same.*

Theorem 6.9 (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* $\mathcal{U}$*.*

Proof (sketch). Correctness follows from the next two facts. Firstly, it is trivial to prove that Algorithm 3 adds all reanalyzed methods and their ancestors to $\mathcal{P}$. Observe that in L7, not only is m added to $\mathcal{P}$, but also the ancestors of all reanalyzed methods ($ancestors(\mathcal{C})$). Secondly, CRs relations can be represented as a directed acyclic graph where the nodes are the cost relations and the edges the dependencies between the cost relations. Using this graph, we can formally prove by induction on the depth of the graph, that all methods in $\mathcal{P}$ are processed and, at the end of Algorithm 3, all invalidated upper bounds are computed. Details of this proof can be found in the online version. □

7. Experiments

We have integrated our techniques within the COSTA system [3], a resource usage analyzer for Java bytecode. COSTA implements all global analyses described in the paper and, in addition, it performs non-nullness and sign analyses. The incremental multi-domain algorithm has been implemented and applied to all domains. We have taken the design and implementation decisions proposed along the paper, namely: (1) the granularity of the analysis is at the level of methods, so that we add, modify or delete methods (and not rules); (2) we use validity flags to invalidate information which must be recomputed; (3) we

prioritize efficiency over accuracy such that incremental analysis does not try to improve the analysis information already inferred by non-incremental analysis.

Our experimental evaluation has been performed on slightly modified versions of programs `Voronoi`, `Health`, `TSP`, and `MST`, from the JOlden benchmark suite [36], available at `http://costa.ls.fi.upm.es`. The modifications (described in [6] 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 algorithm presented in this paper. Furthermore, we have used as benchmarks the following programs borrowed from the Apache-Commons Project [28]: StringEncrypt and ParseTarHeader from Apache-Commons-Compress, and TestOrthogonal and TestDistance from Apache-Commons-Math. 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 a classical SPLE scenario where the core module is available and the generation of a specialized product requires a series of changes. It also simulates a common development scenario where the programmer makes relatively small changes to fix a bug, overrides a method or develops a program in top-down order. Depending on the case, the change can be minimal (for example, after refactoring the code) or larger when the analysis results for the new version of the method drastically differ from the previous ones and they have to be propagated to its 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 is identical to the previous one. In this case, the incremental analysis will reanalyze the updated method, but no other method is reanalyzed. This case simulates the typical change that fixes a minor bug, adapts a method to a concrete environment or prints extra debugging information. 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[3], with the final implementation. Finally, the last scenario simulates a top-down development process, where the programmer starts from the main method and develops the program in descendant order.

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 $\mathcal{T}_0$:

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

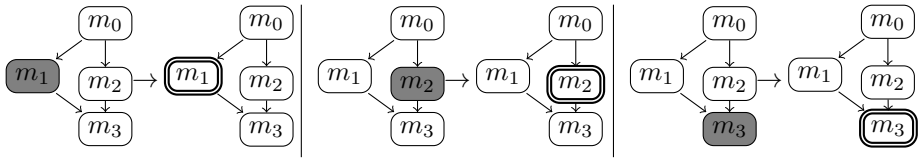
The speedup (S) is computed as the ratio between n times the time taken by the non-incremental analysis of P_0, and the addition of the times of the incremental analysis starting the analysis from different methods in P (m_i):

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

[3]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



$$\mathcal{A}(P_0^1, m_0) \xrightarrow{\mathcal{T}_0^1} \mathcal{A}^\Delta(P_1, m_1) \quad \mathcal{A}(P_0^2, m_0) \xrightarrow{\mathcal{T}_0^2} \mathcal{A}^\Delta(P_1, m_2) \quad \mathcal{A}(P_0^3, m_0) \xrightarrow{\mathcal{T}_0^3} \mathcal{A}^\Delta(P_1, m_3)$$

Figure 3: Adding experiment scheme

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) 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 to trigger 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. Shaded nodes represent empty methods, and white nodes represent implemented methods. 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.

	Unweighted Speedup			**Weighted Speedup**		
Benchmark	$\mathsf{S}_{\mathsf{CRs}}$	S_{UB}	S_{T}	$\mathsf{W}_{\mathsf{CRs}}$	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

(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. It will thus require the largest possible number of reanalysis.

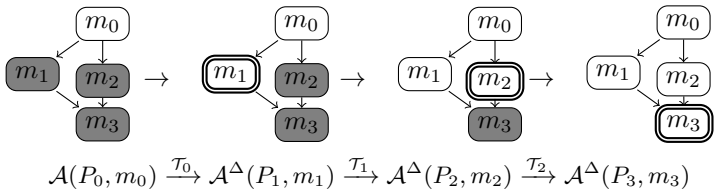


Figure 4: Top-down development experiment scheme

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	$\mathsf{S}_{\mathsf{CRs}}$	S_{UB}	S_{T}	$\mathsf{W}_{\mathsf{CRs}}$	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 in the total *weighted speedup*, 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.

Finally, let us explain in more detail the results obtained for TestOrthogonal as it presents some peculiarities with respect to the rest of the examples. Firstly, in scenarios (1) and (2), it is the most noteworthy example for which $\mathsf{W}_{\mathsf{CRs}} < \mathsf{W}_{\mathsf{UB}}$. It happens because the UB computation for this example is much more expensive in one method which is located at the bottom of the callgraph. Therefore, the upper bound of this method is only recomputed when this is the modified method. This results in significant speedups in the UB recomputation. Besides, in (3), TestOrthogonal has $\mathsf{W}_{\mathsf{CRs}} < 1$. In this example, all modified methods propagate new information to their ancestors, and this propagation also forces the reanalysis of some descendants of the reanalyzed methods, even multiple times. In general, this behaviour could end up in a slowdown of the incremental approach in comparison with the non incremental approach.

8. Related Work

The most related approach to ours is that in [20], which develops a generic incremental analysis algorithm for static analysis of constraint logic programs (CLP). In addition to the language differences, their incremental algorithm does not handle domain dependencies like ours, which is fundamental for an application such as resource usage analysis which relies on multiple pre-analyses with dependencies among them. Besides, our work provides novel definitions for upper bound summaries which enable the incremental reconstruction of cost functions, a problem that has not been considered before. Another difference is that the granularity of the analysis in our case is at the level of methods, whereas [20] considers modifications at the level of *rules*. Rather than methods, CLP programs are structured into procedures (a.k.a. predicates), and are not object oriented. In turn, each procedure is defined as a non-empty sequence of rules. Therefore, incrementality is defined at a finer-grained level in [20]. This finer-grained modularity does not fit well in the object oriented or imperative settings since usually the smallest program parts which are easily identifiable at the level of program editors, compilers, etc. are methods.

Other approaches to incremental analysis are developed for other purposes, e.g., [38] proposes an efficient incremental parser for general context-free grammars which allows generating incremental tools. The work in [19] 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 *upper bound summaries*. An incremental analysis based on incremental specifications such as those found in formal models is presented in [15], while we do not rely on specifications. The notion of summary has been previously used in other contexts [29, 13] different from incremental analysis. It is also worth mentioning recent work on incremental analysis [16] which defines an incremental analysis via domain specific solvers, for declarative modeling language based on first-order logic with sets and relations. The latter work is also related to directed incremental symbolic execution (DiSE) [27], a technique which in principle is more related to testing than to static analysis. However, the novelty of DiSE is to combine the efficiencies of static analysis techniques to compute program difference information with the precision of symbolic execution to explore program execution paths and generate path conditions affected by the differences. We believe that a combined approach like this one could also be adopted for the inference of resource consumption information.

Another related technique is that of modular analysis [12, 25]. There, the main aim is to improve the scalability of analysis by reducing the memory consumption, which is a common bottle-neck of global analysis. However, modular analysis may be less time-efficient than global analysis. In modular analysis, rather than analyzing the whole program at once, the program is split into smaller parts and each part is analyzed separately. For this, the analysis results of each part is stored, either automatically or by using user-provided summaries. Our technique has in common with modular analysis that it automatically stores

summaries. However, the main focus in incremental analysis is time-efficiency rather than memory-efficiency and all program parts affected by changes are analyzed at once, not separately. Finally, modularity per se does not handle the efficient recomputation of analysis results after a program change.

9. Conclusions and Future 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. Our experimental results empirically demonstrate that the techniques pays off in practice.

In future work, we plan to extend our generic incremental analysis framework to the concurrent setting. We want to consider first a simple concurrency model and study how our analysis has to be adapted to produce efficiently sound results in the presence of concurrency. A non-incremental cost analysis framework has been recently defined in [1] for a language based on the concurrent objects paradigm [23]. After considering incremental resource analysis for concurrent objects, the final goal will be to devise a framework for thread-based concurrency.

Acknowledgments

We are grateful to Maria Garcia de la Banda for detailed comments on the form and contents of a preliminary version of this article. We are also grateful to the anonymous referees for theirs comments and suggestions that have greatly improved the quality of the paper. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, December 2011.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[3] 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.

[4] 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.

[5] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Incremental Resource Usage Analysis. In *Proc. of PEPM'12*, pages 25–34. ACM Press, 2012.

[6] 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.

[7] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Garbage Collected Languages. *Science of Computer Programming*, 2012. To appear.

[8] E. Albert, S. Genaim, and A. N. Masud. More precise yet widely applicable cost analysis. In *Proc. of VMCAI'11*, volume 6538 of *LNCS*, pages 38–53. Springer, 2011.

[9] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.

[10] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[11] 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.

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

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

[14] D. Kapur E. Rodriguez-Carbonell. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

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

[16] S. R. Ganov, S. Khurshid, and D. E. Perry. Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers. In *ICFEM'12*, LNCS 7635, pages 414–429. Springer, 2012.

[17] S. Genaim and F. Spoto. Constancy Analysis. In *10th Workshop on Formal Techniques for Java-like Programs*, 2008.

[18] 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.

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

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

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

[22] L. Hubert, T. P. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Procs. of FMOODS'08*, volume 5051 of *LNCS*, pages 132–149. Springer, 2008.

[23] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

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

[25] 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.

[26] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[27] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *PLDI'11*, pages 504–515. ACM, 2011.

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

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

[30] 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.

[31] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of SPLC'10*, pages 77–91. ACM, 2010.

[32] 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.

[33] F. Spoto. Precise null-pointer analysis. *Software and System Modeling*, 10(2):219–252, 2011.

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

[35] 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.

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

[37] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *Proc. of CASCON'99*, pages 125–135. IBM, 1999.

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

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

Appendix A. Proofs of Section 4

Appendix A.1. Proof for Proposition 4.14

Proposition 4.14: *The loop in L4-18 of Algorithm 2 obtains correct analysis information for affected methods.*

Proof. First, let us prove that one iteration of the loop in L4-18 of the incremental analysis (Algorithm 2) correctly propagates its new information to other nodes. We consider a global answer table, $\mathcal{G}_0$, the results from the analysis of method m, extracted from $\mathcal{P}$, and with entry $m : CP_m \mapsto AP_m \in \mathcal{G}_0$. The analysis of m starts by calling *noincr:analysis*, which consecutively calls *process_analysis*. During the analysis of m, we distinguish the following two cases:

- **Descendants**: the reanalysis of a method may find calls to another method m' with calling pattern CP' (L32 of Algorithm 1). Let CP'_0 represent the call pattern of the corresponding entry for m' in $\mathcal{G}_0$, if it exists. One of the following possibilities may occur:
  (a) m calls m', which has no entry in $\mathcal{G}_0$. As before, we will consider w.l.o.g. that $m' : \bot \mapsto \bot$ exists in $\mathcal{G}_0$. Method m' must be reanalyzed for the calling pattern CP'. Algorithm 2 covers this case by evaluating in L21 if the method exists in $\mathcal{G}_0$ and forcing its reanalysis by calling *noincr:get_proc_answer* with call pattern CP' in L30.
  (b) m calls m' with a call pattern CP' that is not contained in the call pattern CP'_0 stored in $\mathcal{G}_0$, that is, $(m', D) : CP'_0 \mapsto AP'_0 \in \mathcal{G}_0$ and $CP' \not\sqsubseteq CP'_0$. In this case, m' must be reanalyzed taking into account the new information. This condition is evaluated in *incr:get_proc_answer* (L24) which compares the new call pattern with the one stored in $\mathcal{G}_0$. Since the condition in L24 does not hold, m' is scheduled for reanalysis by calling *noincr:get_proc_answer* (L31) with $CP'_0 \sqcup CP'$ as calling pattern. m' will be reanalized because it is added to $\mathcal{Q}$ in L17 of Algorithm 1 and this reanalysis will transitively handle all methods that have a call pattern not contained in $\mathcal{G}_0$.
  (c) m calls m' with a call pattern CP' that is contained in the one CP'_0 stored in $\mathcal{G}_0$, that is, $CP' \sqsubseteq CP'_0$. The information stored in $\mathcal{G}_0$ is valid for m' in this iteration, so we can reuse the information stored in $\mathcal{G}_0$. This is evaluated by Algorithm 2 in L24 and the information stored in $\mathcal{G}_0$ is returned in L25.
- **Ancestors**: the reanalysis of a method produces a resulting AP' for the reanalyzed method m. AP_0 represents the answer pattern of the corresponding entry for m in $\mathcal{G}_0$. Let m' represent a caller of method m. One of the following possibilities may occur:

(a) Method m' has not been reanalyzed during the execution of *process_analysis* for method m. In this case we can have two possibilities.

   * If $AP' \sqsubseteq AP$, no new information is generated and, thus, no new analysis is required. This case is handled by Algorithm 2 in L14, checking if the new AP, stored in $\mathcal{L}$, is greater than the one stored in $\mathcal{G}_0$, that is, if $AP_n^{\mathcal{L}} \sqsubseteq AP_n^{\mathcal{G}}$. In that case, the algorithm does not generate any new event.
   * If $AP' \not\sqsubseteq AP$, new information must be propagated to m'. This case is handled by Algorithm 2 in L14, checking if the new AP, stored in $\mathcal{L}$, is greater than the one stored in $\mathcal{G}_0$, that is, if $AP_n^{\mathcal{L}} \not\sqsubseteq AP_n^{\mathcal{G}}$. In that case, the algorithm adds m' to $\mathcal{P}$, forcing its reanalysis in subsequent iterations.

(b) Method m' has been reanalyzed during the execution of *process_analysis* for method m. This is because m' has a new calling pattern found during the non-incremental analysis. In this case we can have two possibilities.

   * If $AP' \sqsubseteq AP$, no new information is generated and, thus, no new analysis is required. This case is handled by the incremental algorithm (Algorithm 2) in L14 and by the non-incremental algorithm (Algorithm 1) in L39, not generating any new event if $AP' \sqsubseteq AP$.
   * If $AP' \not\sqsubseteq AP$, we have to propagate new information to m'. This case is handled by Algorithm 1 in L39-40 by calling *invalidate_callers*, if the new answer pattern is greater than the one stored in $\mathcal{L}$ (L39: $AP \not\sqsubseteq AP^{\mathcal{L}}$). Procedure *invalidate_callers* of Algorithm 1 checks if $\mathcal{L}$ has an entry for m', if true, L24 schedules a new event to be processed for m', forcing its reanalysis.

Once the execution of the reanalysis is completed, $\mathcal{G}_0$ must be updated with the new information to continue the analysis of the rest of remaining methods that are pending to be analyzed with the up-to-date information. This is done by the incremental algorithm in the loop in L11-16, that takes the information from $\mathcal{L}$ and copies it to $\mathcal{G}$.

Now let us prove that when the loop L4-18 terminates $\mathcal{G}$ contains correct information for all affected methods. The incremental algorithm starts by executing *incremental_fixpoint* from the modified method m using the calling pattern information taken from $\mathcal{G}$ (L9) and propagates its new information to its neighbours, generating a sequence of global answer tables, $\mathcal{G}_1 \rightsquigarrow \mathcal{G}_2 \rightsquigarrow \mathcal{G}_3 \ldots \mathcal{G}_n$, one $\mathcal{G}$ for each iteration of the loop in L4-18. As it is proved in Proposition 4.13, all affected methods are reachable starting from m, and the incremental algorithm starts the propagation of new information from m. Note that m is the only method that has been modified, the rest of the methods only propagate the updated information received from affected edges. Thus, when a fixpoint is

reached, that is, when no new information is propagated, the global answer table for the final iteration, $\mathcal{G}_n$, contains all the information propagated from the modified method through the whole program and consequently, $\mathcal{G}_n$ will contain correct information for all methods.

□

Appendix A.2. Proof for Theorem 4.17

Theorem 4.17:. Incremental_fixpoint(m) *terminates and returns a correct* $\mathcal{G}$ *for all methods.*

Proof. Correctness is an inmediate consequence of Propositions 4.13 and 4.16. Termination is guaranteed by the ascending chain condition (ACC) of the abstract domains and the $\sqcup$ domain operator. For proving termination of the incremental algorithm (Algorithm 2), we define the pair (Λ, L) where:

- Λ is a measure of the distance to $\top$ of the elements in $\mathcal{L}$, defined as

$$\Lambda = \sum_{CP_i \mapsto AP_i \in \mathcal{G}} (\delta(CP_i) + \delta(AP_i)) \tag{A.1}$$

  where $\delta(ST)$ is the number of abstract states from the abstract substitution ST to $\top$. The ascending chain condition (ACC) of the abstract domains guarantees that this distance is finite from any state (using widening when needed).

- L is the length of $\mathcal{P}$.

Let $>^{\Lambda,L}$ be the lexicographical ordering induced by $>_{\mathbb{N}}$ over $\mathbb{N}^2$:

$$(\Lambda_1, L_1) >^{\Lambda,L} (\Lambda_2, L_2) \iff (\Lambda_1 > \Lambda_2) \vee (\Lambda_1 = \Lambda_2 \wedge L_1 > L_2)$$

Since $>^{\Lambda,L}$ is a well-founded ordering, termination of Algorithm 2 can be concluded if we show that for each iteration i of Algorithm 2, $(\Lambda_{i-1}, L_{i-1}) >^{\Lambda,L} (\Lambda_i, L_i)$.

We will assume that if in $\mathcal{G}$ there is no entry for method p, it contains $p : \bot \mapsto \bot$ for every rule p that is required by the analysis. At each iteration i of the *while* loop in L4-17 of Algorithm 2, Λ and L may change in the following cases:

- $\mathcal{G}$ is updated by *incremental_fixpoint* in L16. Regarding the calling pattern information, $CP_n^{\mathcal{L}}$ stored in $\mathcal{G}$ in L16 comes from the information contained in $\mathcal{L}$ (L13). We will show that $\delta(CP_n^{\mathcal{L}}) \leq \delta(CP_n^{\mathcal{G}})$ for any $(n, D) \in \mathcal{R}$.

  The analyses of pending methods in $\mathcal{P}$ are launched in the loop at L7-10. These analyses use as calling patterns the information already existing in $\mathcal{G}$ (L9). Therefore, *noincr:process_analysis* will analyze the pending methods using the calling pattern in $\mathcal{G}$.

During the execution of *noincr:process_analysis* there may be calls to other methods which are handled by *incr:get_proc_answer* (L32 of Algorithm 1). In *incr:get_proc_answer* there are three possibilities:

- If there is a valid entry in $\mathcal{G}$ for an invoked method m which is applicable (L24), the answer pattern in $\mathcal{G}$ is used (L25), without updating $\mathcal{L}$.
- If it is not applicable, the calling pattern for m is lubbed with the existing calling pattern in $\mathcal{G}$ (L29), and then *noincr:get_proc_answer* updates $\mathcal{L}$ with this lubbed calling pattern (L18, Algorithm 1).
- If there is no entry in $\mathcal{G}$, the calling pattern for m is stored in $\mathcal{L}$, (L18, Algorithm 1).

In all cases, $\delta(CP_n^{\mathcal{L}}) \leq \delta(CP_n^{\mathcal{G}})$, since $CP_n^{\mathcal{L}} \sqsupseteq CP_n^{\mathcal{G}}$ in any case.

- Regarding answer patterns, $AP_n^{\mathcal{L}} \sqcup AP_n^{\mathcal{G}}$ stored in $\mathcal{G}$ in L16 trivially verifies $AP_n^{\mathcal{L}} \sqcup AP_n^{\mathcal{G}} \sqsupseteq AP_n^{\mathcal{G}}$, and thus $\delta(AP_n^{\mathcal{L}} \sqcup AP_n^{\mathcal{G}}) \leq \delta(AP_n^{\mathcal{G}})$.

In all cases, $\Lambda_{i-1} \geq \Lambda_i$. Let $\mathcal{G}_i$ be the state of $\mathcal{G}$ after iteration i. There are two cases:

- If $\Lambda_{i-1} > \Lambda_i$, then $(\Lambda_{i-1}, L_{i-1}) >^{\Lambda,L} (\Lambda_i, L_i)$ holds.
- If $\Lambda_{i-1} = \Lambda_i$, then all entries $CP \mapsto AP \in \mathcal{G}_{i-1}$ remain unchanged in $\mathcal{G}_i$. We now prove that no new element is added to $\mathcal{P}$. Elements are added to $\mathcal{P}$ in L28 and L15.
  - L28 is not executed because $CP \sqsubseteq CP^{\mathcal{G}}$ is true in L23.
  - L15 is not executed because $AP_n^{\mathcal{L}} \not\sqsubseteq AP_n^{\mathcal{G}}$ is false in L14.

  Since no elements are added to $\mathcal{P}$, its size decreases in L6. Thus, if $\Lambda_{i-1} = \Lambda_i$ and the size of $\mathcal{P}$ decreases, then $(\Lambda_{i-1}, L_{i-1}) >^{\Lambda,L} (\Lambda_i, L_i)$ holds.

In every case the lexicographical order strictly decreases, thus the termination of Algorithm 2 is proved. □

Appendix B. Proofs of Section 6

Theorem 6.9:. *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 $\mathcal{U}$.*

Proof. For proving the theorem, let us introduce some notation. Given a table of summaries $\mathcal{U}$, where C and Y are two relations defined in $\mathcal{U}$, we say that C depends on Y, denoted $C \mapsto Y$, iff there is an upper bound summary such that $UB_C^S = \langle \#iter \cdot \mathtt{aexp}, \psi, \varphi \rangle$, where $\mathtt{aexp} = \mathtt{exp}_0 + \sum_{i=1}^{k} [\mathtt{exp_i}]_{Y_i}$ and $\mathtt{exp}_i$ denote unannotated expressions. A *directed graph* G is a pair $\langle N, E \rangle$ where N

is the set of nodes and $E \subseteq N \times N$ is the set of edges. We associate to each table of summaries $\mathcal{U}$ a graph $G_{\mathcal{U}}$, which is the directed graph obtained from $\mathcal{U}$ by taking the set of entries in $\mathcal{U}$ as N and where $(C, Y) \in E$ iff $C \mapsto Y$. A relation Y is *reachable* from a relation C in $\mathcal{U}$ iff there is a path from C to Y in $G_{\mathcal{U}}$.

The resulting graph is a directed acyclic graph, since the entries in $\mathcal{U}$ are in closed-form. Given a node n, $ancestors(n)$ is the set of nodes in the graph from which n is reachable (i.e., the set of entries whose closed-form upper bound depends on the expression of n). $descendants(n)$ is the set of nodes in the graph reachable from n (the set of entries in $\mathcal{U}$ which must be computed to obtain the upper bound of n). *ancestors* and *descendants* can also be applied to sets of nodes. The *level* of a node n in a directed acyclic graph is the length of the longest path reachable from n. The *depth* of a directed acyclic graph is the longest path in the graph.

It is trivial to prove that Algorithm 3 adds all reanalyzed methods and their ancestors to $\mathcal{P}$. Observe that in L7, not only is m added to $\mathcal{P}$, but also the ancestors of all reanalyzed methods ($ancestors(\mathcal{C})$). Termination of function *do_summary* is ensured by the fact that $G_{\mathcal{U}}$ is a directed acyclic graph. This is because the entries in $\mathcal{U}$ are in closed form. The recursive *do_summary* function performs a depth traversal of this graph.

As regards correctness, the proof is by induction on the depth of $G_{\mathcal{U}}$.

- **Base case**: Let $G_{\mathcal{U}}$ be a graph of depth 0 and m be a node in $G_{\mathcal{U}}$ of level 0. There are no paths starting from m in the graph. In L13 of function *do_summary*, we distinguish two cases:
  - If $m \notin \mathcal{P}$, then the summary stored in $\mathcal{U}$ is correct.
  - Otherwise, since there is no edge from m to any other node, the loop in L15 does not iterate, and m is removed from $\mathcal{P}$. The skeleton generated and stored in $\mathcal{U}$ (L4-5) is correct.

- **Inductive case**: Let us suppose that the summaries stored in $\mathcal{U}$ are correct for a graph $G^i_{\mathcal{U}}$ of depth smaller or equal than i.

  Let $G^{i+1}_{\mathcal{U}}$ a graph of depth $i+1$ and m be a node of level $i+1$ in $G^{i+1}_{\mathcal{U}}$. According to L13, there are two cases:
  - If m is not in $\mathcal{P}$, we have two situations:
    - m was not added to $\mathcal{P}$ in L7. This is because $m \notin \mathcal{C}$ and $m \notin ancestors(\mathcal{C})$. The summary for m stored in $\mathcal{U}$ is not affected by the change, and therefore correct.
    - m was added to $\mathcal{P}$ in L7, its entry in $\mathcal{U}$ recomputed (L15-19), updated in $\mathcal{U}$ (L21) and removed from $\mathcal{P}$ (L20). Therefore, in $\mathcal{U}$ the entry has already been correctly recomputed.
  - Otherwise, all relations from which m directly depends on are traversed in L15. They correspond in $G^{i+1}_{\mathcal{U}}$ to the nodes for which there is an edge from m to them. Let n_j, $1 \leq j \leq k$ be those nodes.

All relations in $\mathcal{P}$ are marked as *invalid* in $\mathcal{U}$ (L6), and are not validated again until the algorithm finishes (L10).

L16 leads to several cases:

* If $m \in \mathcal{C}$, then its skeleton was generated in L4 and its summary is reconstructed (L17-19), updated in $\mathcal{U}$ (L21) and removed from $\mathcal{P}$ (L20) avoiding its recomputation. The recursive call to *do_summary* in L17 is applied to n_j, which in $G_{\mathcal{U}}^{i+1}$ have levels $l_j < i+1$. The induction hypothesis guarantees that *do_summary* produces correct results for a graph of depth less than or equal to i. The subgraph formed by n_j and all nodes reachable from n_j is a graph of depth less or equal than i, and thus the hypothesis holds. Consequently, the summary for m is reconstructed using correct information from n_j.
* If $m \notin \mathcal{C}$, those nodes n_j which are invalid in $\mathcal{U}$ are correctly reconstructed in L17-19 as in the previous case, and the summary for m is updated in $\mathcal{U}$ with the correct maximized subexpressions for n_j (L21).

□