SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–44
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Object-Sensitive Cost Analysis for Concurrent Objects

Elvira Albert[1], Puri Arenas[1], Jesús Correas[1], Samir Genaim[1],
Miguel Gómez-Zamalloa[1], Germán Puebla[2], Guillermo Román-Díez[2]

[1] *DSIC, Complutense University of Madrid*
[2] *DLSIIS, Technical University of Madrid*

SUMMARY

Concurrent objects form a well established model for distributed concurrent systems. In this concurrency model, objects are the concurrency *units* which communicate among them via *asynchronous* method calls. This article presents a novel cost analysis framework for concurrent objects. Cost analysis aims at automatically approximating the *resource consumption* of executing a program in terms of its input parameters. While cost analysis for sequential programming languages has received considerable attention, concurrency and distribution have been notably less studied. The main challenges of cost analysis in a concurrent setting are: (1) Inferring precise *size abstractions* for data in the program in the presence of shared memory. This information is essential for bounding the number of iterations of loops. (2) Distribution suggests that analysis must infer the cost of the diverse distributed components separately. We handle this by means of a novel form of *object-sensitive recurrence equations* which use *cost centers* in order to keep the resource usage assigned to the different components separate. We have implemented our analysis and evaluated it on several small applications which are classical examples of concurrent and distributed programming.

Received . . .

KEY WORDS: static analysis; resource usage analysis; cost analysis; concurrency; distributed systems; object-sensitive analysis

1. INTRODUCTION

Distribution and concurrency are currently mainstream. The Internet and the broad availability of multi-processors radically influence software. Many standard desktop programs have to deal with distribution aspects like network transmission delay and failure. Furthermore, many chip manufacturers are turning to multicore processor designs as a way to increase performance in desktop, enterprise, and mobile processors. This brings renewed interest in developing both new concurrency models and associated programming languages techniques that help in understanding, analyzing, and verifying the behaviour of concurrent and distributed programs.

One of the most important features of a program is its resource consumption. By resource, we mean not only traditional cost measures (e.g., number of executed instructions, or memory consumption) but also concurrency-related measures (e.g., number of tasks spawned, number of requests to remote servers). Example 1 will illustrate these types of resources on a fragment of our running example. Cost analysis (a.k.a. *resource usage* analysis) aims at *statically* inferring

*Correspondence to: **elvira@sip.ucm.es** (Elvira Albert) approximations of the resource consumption of executing the program. Automatically inferring the resource usage of concurrent programs is challenging because of the inherent complexity of concurrent behaviours.

In addition to traditional applications, like optimization [42], verification and certification of resource consumption [16], cost analysis opens up interesting applications in the context of concurrent programming. In general, having anticipated knowledge on the resource consumption of the different components which constitute a system, is useful for distributing the load of work. Upper bounds can be used to predict that one component may receive a large amount of remote requests, while other siblings are idle most of the time. Also, our framework allows instantiating the different components with the particular features of the infrastructure on which they are deployed. Then, analysis can be used to detect the components that consume more resources and may introduce bottlenecks. Lower bounds on the resource usage can be used to decide if it is worth executing locally a task or requesting remote execution.

In order to develop our analysis, we consider a concurrency model based on the notion of concurrently running (groups of) objects, similar to the actor-based and active-objects approaches [35, 39, 1, 41, 28, 34]. These models take advantage of the concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. Concurrent objects communicate via *asynchronous* method calls. Intuitively, each concurrent object is a monitor and allows at most one *active* process to execute within the object. Scheduling among the processes of an object is cooperative, i.e., a process has to release the monitor lock explicitly, except when it terminates. Each object has an unbounded set of pending processes. In case the lock of a concurrent object is free any process in the set of pending processes can grab the lock and start to execute (hence process scheduling is non-deterministic).

Example 1 (notion of resource)**.** Let us consider the concurrent method (which is part of our running example that will be showed later) below. The notation $f = o \,!\, m(\bar{e})$ is used to denote that an asynchronous call $m(\bar{e})$ has been posted on object o and f is a future variable which allows us to know if the execution of the asynchronous call has finished. In such case the result can be retrieved by means of a get operation f.**get**. Also, we can synchronize the execution with the asynchronous call by means of an await instruction, namely **await** $f?$ is used to check if the asynchronous call has finished, otherwise the processor can be released such that a task which was pending to execute can take it.

```
Int process(Int pos) {
   Fut⟨Int⟩ f;
   Int i = 0;
   Int res = 0;
   while (i < elems) {
       f = this ! hdRead(pos + i);
       await f?;
       res = this.update(res,f.get);
       i = i + 1;
    }
    return res;
}
```

In the above method, elems is a class field. Our objective is to measure the resource consumption of executing the above method. One crucial aspect will be to find out if field elems can be modified at the **await** f? instruction. Observe that if the processor is released at the await and another process that increases the value of elems takes the processor, the loop above might not terminate. Our method relies on *class invariants* which contain information on the shared memory at processor release points. By assuming that elems is not modified, we can now consider different types of resources of interest. For instance, we can measure traditional cost measures like number of executed instructions or memory consumption. In the former case, we will infer that $3 + \mathsf{elems} * (7 + hdRead_{inst} + update_{inst})$ instructions will be executed. In the expression, elems corresponds to the maximum number of iterations of the loop, at each iteration 7 instructions are executed (the loop condition, the two increments, two method invocations, the **get** and the **await**) and the number of instructions of executing methods hdRead and update, denoted $hdRead_{inst}$ and $update_{inst}$ respectively, are added as well. For simplicity, we ignore here the values of the parameters. Besides, 3 instructions are executed outside the loop. The analysis will need to infer also the cost of the methods hdRead and update and plug them in the expression above. As regards memory consumption, new memory is not created in the loop, hence we would output an expression of the form elems $* \, (hdRead_{heap} + update_{heap})$ which relies on the memory created by the invoked methods (denoted $hdRead_{heap}$ and $update_{heap}$). Interestingly, we can also consider concurrency-related measures like the number of tasks spawned. In the above method, we spawn one task directly (but the method invocations might also spawn new tasks transitively). Therefore, we will compute an expression of the form elems $* \, (1 + hdRead_{tasks} + update_{tasks})$ which relies on the number of tasks spawned by the calls to hdRead and update. Observe that the types of resources we have considered are platform independent (unlike WCET or energy consumption), i.e., they can be inferred by inspecting the program, and the hardware on which the program will be executed can be ignored. Platform dependent resources are beyond the scope of this work. ■

1.1. *Contributions*

We propose a static cost analysis for concurrent objects, which is parametric w.r.t. the notion of resource that can be instantiated to measure both traditional and concurrency-related resources. The main contributions of this work are:

1. We present a flow-sensitive object-sensitive points-to analysis for concurrent programs which adapts Milanova's analysis framework [32] for Java to the concurrent object setting;
2. We introduce a sound size analysis for concurrent execution. The analysis is *field-sensitive*, i.e., it tracks data stored in the heap whenever it is sound to do so; the accuracy of the field-sensitive size analysis can be increased by means of *class invariants* [31] which contain information on the shared memory;
3. We leverage the definition of cost used in sequential programming to the distributed setting by relying on the notion of *cost centers* [33], which represent the (distributed) components and allow separating their costs;
4. We present a novel form of *object-sensitive* recurrence relations which relies on information gathered by the previous object-sensitive points-to analysis in order to generate the cost equations. Interestingly, the resulting recurrence relations can still be solved to closed-form upper/lower bounds using standard solvers for cost analysis of sequential programs;
5. We report on the SACO system, a prototype implementation of a cost analyzer for programs written in ABS [27] (an Abstract Behavioural Specification language based on concurrent objects).

It is recognized that performing the analysis on a high-level concurrency model, like the concurrent objects model, makes verification more feasible. This is because analysis in concurrent systems often needs to consider too many interleavings and thus ends up being limited to very small programs in practice. We argue that our approach is of both practical and theoretical relevance.

This work is an extended and revised version of APLAS'11 [3]. Points 2 and 3 in the contribution list can be considered as original contributions of the conference, while this journal paper has points 1 and 4 as original contributions, and also with an implementation of them (last point). However, the treatment of fields in the size analysis (included in point 2 in the contribution list) has been improved in this article.

1.2. *Organization of the Article*

The remainder of the article is organized as follows. Section 2 presents the syntax and semantics of the language on which we develop our analysis. Section 3 defines the notion of cost for the concurrent distributed programs that we aim at approximating by means of the resource analysis.

The next three sections present the resource-analysis framework in several steps. Our starting point is a powerful cost analysis framework for sequential OO programs [7]. When lifting such framework to the concurrent and distributed setting, there are two main difficulties and novelties. First, it is widely recognized that, due to the possible interleaving between tasks, tracking values of data stored in the heap is challenging [13, 30]. In Section 5, we present the basic, novel, *field-sensitive* size analysis for the concurrent setting.

The second difficulty is related to the fact that standard recurrence relations (in the sequential setting) assume a single cost center which accumulates the cost of the whole execution. We propose a novel form of *recurrence relations* which use *cost centers* to split the cost of the diverse distributed components. This requires first the inference of object-sensitive points-to information which approximates the set of objects which each reference variable may point. Section 4 adapts the object-sensitive points-to analysis of Milanova [32, 37] to our setting. Then, the points-to information gathered by the analysis allows us to define in Section 6 object-sensitive recurrence relations which, together with the size abstractions, constitute the core of our analysis.

Section 7 presents SACO, a prototype implementation of our analysis, and evaluates it on a series of typical applications of concurrent and distributed programming. Finally, Section 8 reviews the related work and Section 9 recaps the main conclusions.

2. A LANGUAGE WITH CONCURRENT OBJECTS

The concurrency model of Java and C# is based on threads that share memory and are scheduled preemptively, i.e., they can be suspended or activated at any time. To avoid undesired interleavings, low-level synchronization mechanisms such as locks have to be used. Thread-based programs are error-prone, difficult to debug, verify and maintain. In order to overcome these problems, several higher-level concurrency models that take advantage of the inherent concurrency implicit in the notion of object have been developed [35, 39, 26, 17, 31]. They provide simple language extensions that allow programming concurrent applications with relatively little effort. Concurrent objects [26, 17] form today a well established high-level model for distributed concurrent systems.

2.1. *The Concurrency Model*

For the sake of concreteness, we develop our analysis on a simple imperative language with concurrent objects, which is the imperative subset of the ABS language [27]. However, our techniques work for other languages that use actors (e.g., there are implementations of actor libraries for Scala, Java, Erlang, among others). The central concept of this concurrency model is that of *concurrent object*. Conceptually, each object has a dedicated processor and encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the *this* object, and any other object can only access such fields through method calls. Concurrent objects live in a distributed environment with asynchronous and unordered communication by means of asynchronous method calls. Thus, an object has a set of tasks (i.e., calls) to execute and, among them, at most one task is *active* and the others are *suspended* on a task queue.

Asynchronous method calls may be seen as triggers of concurrent activity, spawning new tasks (so-called processes) in the called object. After asynchronously calling method m of object o with arguments $\overline{e}$, denoted by $f := o\ !\ m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here f is a *future variable* which refers to a return value which has yet to be computed. There are two operations on future variables, which control external synchronization. First, **await** $f?$ suspends the active task (allowing other tasks in the object to be scheduled) until the future variable f has been assigned a value. Second, the value stored in f can be retrieved using f.**get**, which blocks all execution in the object until f gets a value (in case it has not been assigned a value yet).

Example 2 (syntax of ABS)**.** Figure 1 shows the *s*ource code of our running example which implements a simple *file input stream* (defined in class FileIS) that provides two different ways of processing a file. The class contains three fields (defined as class parameters) which represent, respectively, the name of the file fp, the length of the file lth, and the size of the block to be read from the field blockS. Method readBlock reads file fp block by block (of sizes blockS) and sums the values

```
class FileIS(String fp, Int lth, Int blockS) {
  Int readBlock () {
    Int res = 0; Int i = this.lth;
    Int incr = 0; Int pos = 0;
    while (i > 0) {
      if (this.blockS > i) incr = i;
      else incr = this.blockS;
      Fut⟨Int⟩ f;
      f = this ! readContent(pos,incr);
      await f?;
      res = res + f.get;
      i = i - incr;
      pos = pos + incr;
    }
    return res;
  }
  Int readOnce() {
    Fut⟨Int⟩ f = this ! readContent(0,this.lth);
    await f?;
    return f.get;
  }
  Int readContent(Int pos, Int elems) {
    Reader rd = new Reader (this.fp,elems);
    Fut⟨Int⟩ f = rd ! process(pos);
    await f?;
    return f.get;
  }
}// end class FileIS
```

```
class Reader(String fp, Int elems) {
  Int hdRead(Int i){ ··· }
  Int update(Int a, Int b){ ··· }
  Int process(Int pos) {
    Fut⟨Int⟩ f;
    Int i = 0;
    Int res = 0;
    while (i < this.elems) {
      f = this ! hdRead(pos + i);
      await f?;
      res = this.update(res,f.get);
      i = i + 1;
    }
    return res;
  }
}// end class Reader

main {
  FileIS o1 = new FileIS("A.txt",20,2);
  FileIS o2 = new FileIS("A.txt",20,3);
  Fut⟨Int⟩ f1; Fut⟨Int⟩ f2;
⊛f1 = o1 ! readOnce();
  f2 = o2 ! readBlock();
  await f1?;
  Int r1 = f1.get;
  await f2?;
  Int r2 = f2.get;
}
```

Figure 1. Running Example

retrieved using **get**. Method readOnce reads the whole file in just one invocation to readContent. The latter method invokes method process of class Reader which reads and processes elems elements of the file starting at position pos. Method hdRead represents the low-level access to the hard-disk and method update performs some arithmetic operation on its arguments and returns an integer value. We do not show the code of these methods as they are not relevant for the purpose of this article. ■

2.2. *A Rule-based Intermediate Language*

To contextualize the formalization of the analysis in a simpler model, we develop our analysis on an intermediate representation (IR) similar to those for Java bytecode and .NET [40, 7, 38, 18]. In the IR, *recursion* is the only iterative mechanism and *guards* are the only form of conditional. In the following, given any entity t, we use $\bar{t}$ to denote the tuple $\langle t_1, \ldots, t_n \rangle$. The compilation of a program into the IR is done by building the CFG for the original program and representing each block in the CFG by means of a rule. The following definition establishes the formal syntax of the IR.

Definition 1 (syntax of IR programs)**.** *A program in the IR consists of a set of* classes $\bar{C}$. *Each class* C *contains a set of* fields $\bar{f}_C$ *and a set of procedures. A* procedure m *of class* C *is defined by a set of* guarded rules. *A guarded rule for* m *has the form "*$r \equiv C.m(this, \bar{x}, \bar{y}) \leftarrow g, \bar{b}$*.", where* $C.m(this, \bar{x}, \bar{y})$ *is the* head *of the rule, this is the identifier of the object on which the method is executing,* g *specifies the conditions for the rule to be applicable, and,* $\bar{b}$ *is the rule's* body. *Guards* g *and instructions* $b \in \bar{b}$ *are defined according to the following grammar:*

$$
\begin{array}{lll}
b & ::= x := rhs \mid this.f{:=}y \mid \mathbf{await}\ x? \mid \mathbf{call}(ct, m(rec, \bar{x}, \bar{y})) & \\
g & ::= true \mid x\ op_R\ y & \\
rhs & ::= e \mid \mathbf{new}\ C \mid x.\mathbf{get} & \\
e & ::= \mathbf{null} \mid this.f \mid x \mid n \mid x\ op_A\ y & \\
op_R & ::= < \mid > \mid = \mid \neq \mid \geq \mid \leq \qquad op_A ::= + \mid - \mid / \mid * & ct ::= \mathbf{m} \mid \mathbf{b}
\end{array}
$$

where x and y denote variable names*, f a* field name*, $\texttt{call}(ct, m(rec, \bar{x}, \bar{y}))$ a call to a method or a block and $n \in \mathbb{Z}$.*

The first argument ct of a call $\texttt{call}(ct, m(rec, \bar{x}, \bar{y}))$ can be either **m** or **b**. The identifier **m** is used for asynchronous method calls whereas **b** is used for synchronous method calls or calls to intermediate blocks. For instance, intermediate blocks can correspond to if-then-else statements or loops; rec is a variable that refers to the receiver object. For synchronous method calls or calls to blocks, rec is always $this$; the variables $\bar{x}$ (respectively $\bar{y}$) are the formal parameters (respectively return values). For methods, $\bar{y}$ is either empty $\langle\rangle$ or contains a single output variable $\langle y\rangle$.

An instruction $\mathsf{x} = \textbf{new}\ \mathsf{C}(\bar{\mathsf{t}})$ in the target language, is represented in the IR by $x := \textbf{new}\ C$ followed by a call to the class constructor with the corresponding parameters $\bar{t}$. For example, assuming that the class C contains a field f of type integer, the instruction $\mathsf{x} = \textbf{new}\ \mathsf{C}(2)$ will be translated into $x := \textbf{new}\ C, y := 2; C_{init}(this, \langle y\rangle, \langle\rangle)$, where $C_{init}(this, \langle y\rangle, \langle\rangle) \leftarrow this.f := y$.

The translation from the high-level programs to the IR is (almost) identical to the translation of Java (bytecode) to the IR in [7], where classes and fields in the IR are the same as in the original program and each method m of a class C is represented in the IR by a single procedure named $C.m$ (the method entry). The other rules in the IR are intermediate procedures used only within the method, with $ct = \textbf{b}$. The **main** method does not belong to any class. Without lack of generality, we assume that method names are unique, and we omit C when referring to m. Furthermore, we will omit those guards which are $true$. This happens in the rules corresponding to method entries.

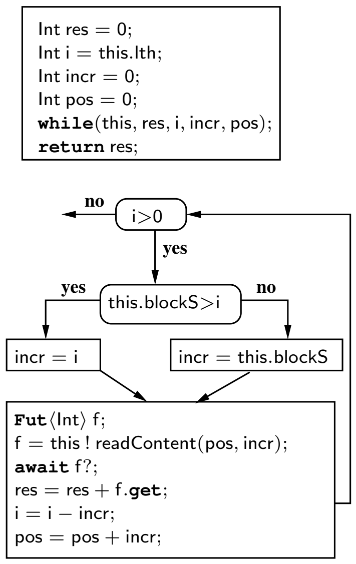
Example 3 (CFG of an IR for the running example)**.** Figure 2 depicts the IR (left) and the CFG (right) of method readBlock. Loops are extracted in separate CFGs to enable compositional cost analysis (e.g., the CFG at the bottom is the one for the *while* loop). The method is represented by four procedures, *readBlock*, *while*, *if* and *if*^{c}, which have a correspondence with blocks in the CFG and the entry to the loop. Each procedure is defined by means of guarded rules. As notation $\overline{inp}$ stands for $\langle res, i, incr, pos\rangle$ and $\overline{out}$ for $\langle res, i, incr, pos\rangle$. Guards in rules state the conditions under which the corresponding blocks in the CFG can be executed. When there is more than one successor in the CFG, we create a *continuation procedure* and the corresponding call in the rule. Blocks in the continuation will in turn be defined by means of (mutually exclusive) guarded rules. As a result of the translation, observe that all forms of iteration in the program are represented by means of *recursive* calls. The unique parameter of the procedure readBlock is the reference to the *this* object. When calling a block, we pass as arguments all local variables that are needed in the block. The heap remains implicit. ■

2.3. Operational Semantics

An *object* is of the form $\mathtt{ob}(o, C, h, \langle tv, \bar{b}\rangle, \mathcal{Q})$, where o is the *object identifier*, C is its class name, h is its local *heap*, $\langle tv, \bar{b}\rangle$ is the *execution context* of the current task, being tv the *table of local variables* and $\bar{b}$ the sequence of instructions to be executed by the current task, and $\mathcal{Q}$ is the set of *pending tasks*, being each of them an execution context. In the following we use ϵ to denote either an empty sequence of instructions or an empty execution context. A heap h maps *field names* $\bar{f}_C$ declared in C to $\mathbb{V} = \mathbb{Z} \cup \{\mathsf{null}\} \cup \mathit{Objects}$, where $\mathit{Objects}$ denotes the set of object identifiers. A table of variables tv maps local variables to $\mathbb{V}$. It contains the special entry $\mathtt{ret}$ to associate the return variable of a method to the corresponding future variable. *Future events* have the form $\mathtt{fut}(\mathtt{fn}, v)$ where $v \in \mathbb{V} \cup \{\perp\}$ and $\mathtt{fn}$ stands for a future variable identifier. The symbol $\perp$ indicates that $\mathtt{fn}$ does not have a value yet. For simplicity, we assume that all methods return a single value, while intermediate blocks will often have several return values. An *execution state* (or *configuration*) $\mathcal{S}$ has the form $\{a_1, \ldots, a_n\}$, where a_i can be either an object or a future event. Execution states are in fact represented as sets of objects and future events. In the following, we use the notation $\{a|\mathcal{S}\}$ to denote the set $\{a\} \cup \mathcal{S}$.

The *operational semantics* is given in a rewriting-based style, where, at each step, a subset of the state is rewritten according to the rules in Figure 3. Let us intuitively explain the semantics. Function

```
$readBlock(this, \langle\rangle, \langle r\rangle) \leftarrow$
    $res := 0, i := this.lth,$
    $incr := 0, pos := 0,$
    $\mathtt{call}(\mathbf{b}, while(this, \overline{inp}, \overline{out})),$
    $r := res.$
$while(this, \overline{inp}, \overline{out}) \leftarrow i \leq 0.$
$while(this, \overline{inp}, \overline{out}) \leftarrow i > 0,$
    $\mathtt{call}(\mathbf{b}, if(this, \overline{inp}, \overline{out})).$
$if(this, \overline{inp}, \overline{out}) \leftarrow this.blockS > i,$
    $incr := i,$
    $\mathtt{call}(\mathbf{b}, if^c(this, \overline{inp}, \overline{out})).$
$if(this, \overline{inp}, \overline{out}) \leftarrow this.blockS \leq i,$
    $incr := this.blockS,$
    $\mathtt{call}(\mathbf{b}, if^c(this, \overline{inp}, \overline{out})).$
$if^c(this, \overline{inp}, \overline{out}) \leftarrow$
    $\mathtt{call}(\mathtt{m}, readContent(this, \langle pos, incr\rangle, \langle f\rangle)),$
    $\mathtt{await}\ f?, v := f.\mathtt{get},$
    $res := res + v, i := i - incr,$
    $pos := pos + incr,$
    $\mathtt{call}(\mathbf{b}, while(this, \overline{inp}, \overline{out})).$
```



```
Int res = 0;
Int i = this.lth;
Int incr = 0;
Int pos = 0;
while(this, res, i, incr, pos);
return res;
```

```
Fut⟨Int⟩ f;
f = this ! readContent(pos, incr);
await f?;
res = res + f.get;
i = i − incr;
pos = pos + incr;
```

Figure 2. The IR and CFG for method readBlock

$eval_e$ evaluates an expression e with respect to a heap h and a table of variables in the standard way. Note that the heap is required to evaluate expressions of the form $this.f$, that returns $h(f)$ as result. Function $eval_{gd}(g, tv)$ in rule 4 returns $true$ iff $g \equiv true$ or $g \equiv x_1\ op_R\ x_2$ and $tv(x_1)\ op_R\ tv(x_2)$ holds. Finally, the evaluation of the conditions in **await** instructions is done by function $eval_{aw}$. In particular, in rules 9 and 10, this function behaves as follows: $eval_{aw}(x?, tv, \mathcal{S}) = true$ iff $tv(x) = \mathtt{fn}$ and $\mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}$ and $v \neq \bot$. The notation $tv[x \mapsto v]$ (respectively $h[f \mapsto v]$) is used for storing v in the local variable x (respectively field f).

Rules 1 and 2 operate in the expected way. In rule 3, it can be observed that the table of variables tv maps x to o_1. Function $newRef()$ is in charge of generating fresh object identifiers and procedure $newHeap(D, h_1)$ creates a new mapping for fields in D, where each field is initialized to either 0 or null. In rule 4, a call to a block is resolved by finding a matching rule and adding its body to the sequence of instructions to be executed. The notation $r \equiv p(this', \bar{x}', \bar{y}) \leftarrow g', b_1', \ldots, b_n' \ll P$ stands for a fresh renaming of a rule in P except for output variables $\bar{y}$ and function $newEnv(vars(r) - \{\bar{y}\})$ creates a new mapping for variables in r except for $\bar{y}$ which remain the same, where each variable is initialized to either 0 or **null**. In general, given any entity t, we use $vars(t)$ to denote the set of variables occurring in t. Furthermore, $tv_2 = tv_1[this' \mapsto o, \bar{x}' \mapsto tv(\bar{x})]$ defines a new mapping tv_2 as $tv_2(this') = o$, $tv_2(\bar{x}') = tv(\bar{x})$ and $tv_2(z) = tv_1(z)$ otherwise. Similarly, $tv \cup tv_2$ defines the following mapping: if $x \in dom(tv)$ then $(tv \cup tv_2)(x) = tv(x)$, and $(tv \cup tv_2)(x) = tv_2(x)$ otherwise. It is used to extend a local variable table with the new variables introduced by the block. As notation $dom(tv)$ stands for the set of variables on which tv is defined. When the execution of a block finishes (rule 6) the state is prepared to later apply rule 11 to select a new task from the queue. The condition $\mathtt{ret} \notin dom(tv)$ in rule 6 ensures that execution does not correspond to an asynchronous call (because they always have a return `ret`) but rather to a block or a synchronous one.

Rule 5 deals with asynchronous method invocations. When an object o_1 calls a method $p(\bar{x})$, the information required to execute the call is stored in the queue of the object identified by o_1. Note that parameter passing is done in the construction of tv_3, where the entries for $this'$ and $\bar{x}'$ are assigned a local copy of the value of the actual parameters rec and $\bar{x}$, respectively. Objects are not directly passed as parameters. Instead we pass the corresponding object identifier, which is unique. Function $newFut()$ generates a fresh future variable identifier. Observe that tv_3 has the special entry `ret` to store the relation between the future variable `fn` where the result is stored and the output parameter y'. This future variable is initially undefined, thus $\mathtt{fut}(\mathtt{fn}, \bot)$ is added to the state. When the method

$$(1)\ \dfrac{v = eval_e(e, h, tv)}{\{\texttt{ob}(o, C, h, \langle tv, x := e \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \langle tv[x \mapsto v], \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(2)\ \dfrac{v = tv(y)}{\{\texttt{ob}(o, C, h, \langle tv, \mathit{this}.f := y \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h[f \mapsto v], \langle tv, \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(3)\ \dfrac{o_1 = \mathit{newRef}(),\ \mathit{newHeap}(D, h_1)}{\begin{array}{c}\{\texttt{ob}(o, C, h, \langle tv, x := \mathbf{new}\ D \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \\ \{\texttt{ob}(o, C, h, \langle tv[x \mapsto o_1], \bar{b}\rangle, \mathcal{Q}), \texttt{ob}(o_1, D, h_1, \epsilon, \emptyset)|\mathcal{S}\}\end{array}}$$

$$(4)\ \dfrac{\begin{array}{c}r \equiv p(\mathit{this}', \bar{x}', \bar{y}) \leftarrow g', b'_1, \ldots, b'_n \ll P, o \equiv tv(\mathit{this}), tv_1 = \mathit{newEnv}(\mathit{vars}(r) - \{\bar{y}\}), \\ tv_2 = tv_1[\mathit{this}' \mapsto o, \bar{x}' \mapsto tv(\bar{x})], \mathit{eval}_{gd}(g', tv_2) = \mathit{true}\end{array}}{\{\texttt{ob}(o, C, h, \langle tv, \mathbf{call}(\mathbf{b}, p(\mathit{this}, \bar{x}, \bar{y})) \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \langle tv \cup tv_2, b'_1 \cdots b'_n \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(5)\ \dfrac{\begin{array}{c}r \equiv p(\mathit{this}', \bar{x}', y') \leftarrow b'_1, \ldots, b'_n \ll P, o_1 \equiv tv(\mathit{rec}), \texttt{fn} = \mathit{newFut}(), \\ tv_2 = \mathit{newEnv}(\mathit{vars}(r)), tv_3 = tv_2[\mathit{this}' \mapsto o_1, \bar{x}' \mapsto tv(\bar{x}), \texttt{ret} \mapsto (y', \texttt{fn})]\end{array}}{\begin{array}{c}\{\texttt{ob}(o, C, h, \langle tv, \mathbf{call}(\mathbf{m}, p(\mathit{rec}, \bar{x}, y)) \cdot \bar{b}\rangle, \mathcal{Q}), \texttt{ob}(o_1, D, h_1, \langle tv_1, \bar{b}_1\rangle, \mathcal{Q}')|\mathcal{S}\} \rightsquigarrow \\ \{\texttt{ob}(o, C, h, \langle tv[y \mapsto \texttt{fn}], \bar{b}\rangle, \mathcal{Q}), \texttt{ob}(o_1, D, h_1, \langle tv_1, \bar{b}_1\rangle, \{\langle tv_3, b'_1 \cdots b'_n\rangle\} \cup \mathcal{Q}'), \texttt{fut}(\texttt{fn}, \bot)|\mathcal{S}\}\end{array}}$$

$$(6)\ \dfrac{\texttt{ret} \notin \mathit{dom}(tv)}{\{\texttt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \epsilon, \mathcal{Q})|\mathcal{S}\}}$$

$$(7)\ \dfrac{\texttt{ret} \in \mathit{dom}(tv), (y, \texttt{fn}) = tv(\texttt{ret}), v = tv(y)}{\{\texttt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q}), \texttt{fut}(\texttt{fn}, \bot)|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \epsilon, \mathcal{Q}), \texttt{fut}(\texttt{fn}, v)|\mathcal{S}\}}$$

$$(8)\ \dfrac{\texttt{fn} = tv(y),\ v \neq \bot}{\{\texttt{ob}(o, C, h, \langle tv, x := y.\mathbf{get} \cdot \bar{b}\rangle, \mathcal{Q}), \texttt{fut}(\texttt{fn}, v)|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \langle tv[x \mapsto v], \bar{b}\rangle, \mathcal{Q}), \texttt{fut}(\texttt{fn}, v)|\mathcal{S}\}}$$

$$(9)\ \dfrac{\mathit{eval}_{aw}(x?, tv, \mathcal{S}) = \mathit{true}}{\{\texttt{ob}(o, C, h, \langle tv, \mathbf{await}\ x? \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \langle tv, \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\}}$$

$$(10)\ \dfrac{\mathit{eval}_{aw}(x?, tv, \mathcal{S}) = \mathit{false}}{\{\texttt{ob}(o, C, h, \langle tv, \mathbf{await}\ x? \cdot \bar{b}\rangle, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, \epsilon, \{\langle tv, \mathbf{await}\ x? \cdot \bar{b}\rangle\} \cup \mathcal{Q})|\mathcal{S}\}}$$

$$(11)\ \dfrac{b \in \mathcal{Q}}{\{\texttt{ob}(o, C, h, \epsilon, \mathcal{Q})|\mathcal{S}\} \rightsquigarrow \{\texttt{ob}(o, C, h, b, \mathcal{Q} - \{b\})|\mathcal{S}\}}$$

Figure 3. Operational Semantics

returns a value (rule 7), the entry `ret` is used to look for the corresponding future variable and $\bot$ is updated with the returned value.

Rule 9 checks if a future variable is ready. In such case the computation proceeds. Otherwise, in rule 10, the **await** task is introduced in the corresponding queue, and the processor is released. The instruction **get** blocks the execution until the future variable has a value in rule 8. In rule 11 another task is dequeued (because the current one has terminated or released the processor). Note that this rule is applicable after applying rules 6 and 7 which correspond, respectively, to the complete execution of a block and a method, and rule 10 in which the processor is released.

We assume that executions start from a **main** method. Thus, the *initial configuration* is of the form $\{\texttt{ob}(\texttt{main}, \bot, \bot, \langle tv_0, \mathbf{call}(\mathbf{b},\ \mathsf{main}(\mathit{this}, \langle\rangle, \langle\rangle))\rangle, \emptyset)\}$ where the local variables in tv are initialized to their default values. Abusing notation, we use $\bot$ to denote an empty heap and an undefined class. The execution then proceeds by applying non-deterministically the execution steps in Figure 3. It is non-deterministic both in task and object selection. The execution finishes in a *final configuration* in which all events are either future events or objects of the form $\texttt{ob}(o, C, h, \epsilon, \emptyset)$. Executions can be regarded as *traces* $\mathcal{T}$ of the form $\mathcal{S}_0 \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}_n$.

Example 4 (a trace in the running example)**.** Consider the **main** method of the running example (Figure 1). After executing the constructors we reach a configuration with three objects:
$\{\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}, \bar{b}\rangle, \emptyset), \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon, \emptyset), \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon, \emptyset)\}$
where $\bar{b}$ corresponds to the sequence of instructions from the mark $\circledast$ on. After processing both asynchronous calls (rule 5) consecutively, the new state takes the form:

$$\begin{array}{ll} \{ & \mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}[f_1 \mapsto \mathtt{fn}_1, f_2 \mapsto \mathtt{fn}_2], \bar{b}'\rangle, \emptyset), \\ & \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon, \{\langle tv_{o_1}, \mathit{body}_{o_1}\rangle\}), \mathtt{fut}(\mathtt{fn}_1, \bot), \\ & \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon, \{\langle tv_{o_2}, \mathit{body}_{o_2}\rangle\}), \mathtt{fut}(\mathtt{fn}_2, \bot) \quad \} \end{array}$$

where body_{o_1} (respectively body_{o_2}) is the renamed body of method readOnce (respectively readBlock). Furthermore, tv_{o_1} (respectively tv_{o_2}) stores the assignment $tv_{o_1}(\mathtt{ret}) = (f_1, \mathtt{fn}_1)$ (respectively $tv_{o_2}(\mathtt{ret}) = (f_2, \mathtt{fn}_2)$). When the event $\langle tv_{o_1}, \mathit{body}_{o_1}\rangle$ is extracted from the queue of o_1 (rule 11), its complete processing will replace $\mathtt{fut}(\mathtt{fn}_1, \bot)$ by $\mathtt{fut}(\mathtt{fn}_1, v)$ (rule 7), where v is the value returned by the method readOnce. Then, rule 9 can be used to process the instruction **await** f_1? of the object main. At this point the new state will take this form:

$$\begin{array}{ll} \{ & \mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}[f_1 \mapsto \mathtt{fn}_1, f_2 \mapsto \mathtt{fn}_2], \bar{b}''\rangle, \emptyset), \\ & \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon, \emptyset), \mathtt{fut}(\mathtt{fn}_1, v), \\ & \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon, \{\langle tv_{o_2}, \mathit{body}_{o_2}\rangle\}), \mathtt{fut}(\mathtt{fn}_2, \bot) \quad \} \end{array}$$

■

3. COST AND COST MODELS FOR CONCURRENT PROGRAMS

In this section we define the notion of cost we want to approximate by using static analysis. An execution step is annotated as $\mathcal{S} \leadsto_o^b \mathcal{S}'$, which denotes that we move from a state $\mathcal{S}$ to a state $\mathcal{S}'$ by executing instruction b in object o. Note that from a given state there may be several possible execution steps that can be taken since we make no assumptions on task scheduling and object selection. In order to quantify the cost of an execution step, we use a generic cost model. The following definition formalizes the notion of cost model, where *Ins* stands for the set of instructions b built using the grammar in Def. 1.

Definition 2 (cost model and cost of execution steps)**.** *A* cost model $\mathcal{M}$ *is a function defined as* $\mathcal{M} : \mathit{Ins} \mapsto \mathbb{R}$. *The cost of an execution step is defined as* $\mathcal{M}(\mathcal{S} \leadsto_o^b \mathcal{S}') = \mathcal{M}(b)$.

In the execution of sequential programs, the *cumulative cost of a trace* $\mathcal{T}$ is obtained by applying the cost model to each step of the trace. In our setting, this has to be extended because, rather than considering a single machine in which all steps are performed, we have a potentially distributed setting, with multiple objects possibly running concurrently on different CPUs. Thus, rather than aggregating the cost of all executing steps, it is more useful to treat execution steps which occur on different computing infrastructures separately. With this aim, we adopt the notion of *cost centers* [33], proposed for profiling functional programs. Since the concurrency unit of our language is the object, cost centers are used to charge the cost of each step to the cost center associated to the object where the step is performed. For a given set of object identifiers O and a trace $\mathcal{T}$, we define a restricted trace, $\mathcal{T}|_O = \{\mathcal{S}_i \leadsto_o^b \mathcal{S}_{i+1} \mid \mathcal{S}_i \leadsto_o^b \mathcal{S}_{i+1} \in \mathcal{T},\ o \in O\}$ to denote the set of execution steps that are attributed to the objects in O. Then, we can define the cost executed by a particular object for a given trace:

Definition 3 (cost attributed to an object)**.** *Given a trace* $\mathcal{T}$*, a cost model* $\mathcal{M}$ *and an object identifier* o*, we define the* cost of $\mathcal{T}$ w.r.t. $\mathcal{M}$ attributed to o *as:*

$$\mathcal{C}(\mathcal{T}, o, \mathcal{M}) = \sum_{t \in \mathcal{T}|_{\{o\}}} \mathcal{M}(t)$$

3.1. *Examples of Cost Models*

We consider platform independent cost models (e.g., worst-case execution time or energy consumption are excluded). A cost model for approximating the *number of executed instructions* can be defined as $\mathcal{M}_i(b) = 1$, for all $b \in \mathit{Ins}$. Note that also calls of the form **call**(**b**, _) count 1. This is because the call either corresponds to a synchronous call or to a call block requiring the execution of a guard. By '_', we mean any (valid) expression.

Other interesting cost models can be defined. For instance, a cost model that counts the *total number of objects created* along the execution can be defined as $\mathcal{M}_o(b) = 1$ if $b \equiv$ **new** C and $\mathcal{M}_o(b) = 0$ otherwise. Since objects are the concurrency units, this cost model provides an indication on the amount of parallelism that might be achieved. A cost model that counts **call**(**m**, _), can be used to infer the number of tasks that are spawned along an execution. We can also count the number of calls to specific methods or objects, e.g., by counting **call**(**m**, _(o, _, _)) we obtain bounds on the number of requests to a component o. This is useful for approximating the components' load and finding optimal deployment configurations (e.g., group objects according to the amount of tasks they receive to execute, by also taking into account the infrastructure on which they are deployed). The above cost models can also be used to prove termination of the program by setting the underlying solver [4] to only bound the number of iterations in loops. It is customary to have a cost model for memory consumption. The ABS language has a functional sub-language used to create data types. Hence in our actual implementation, the memory consumption cost model counts the sizes of constructed terms, and the sizes of the objects which are not intended to be concurrency units.

4. POINTS-TO ANALYSIS FOR CONCURRENT PROGRAMS

The aim of points-to analysis is to approximate the set of objects which each reference variable may point to during program execution. An analysis is *object-sensitive* [32, 37] if methods may be analyzed separately for different (sets of) objects on which they are invoked. More precisely, the analysis uses a finite set of *object names* to partition the (possibly infinite) set of objects allocated at runtime into *contexts* which are analyzed separately.

This section presents a flow-sensitive object-sensitive points-to analysis for concurrent programs. It is based on Milanova's analysis framework [32] for Java. As Milanova's analysis is flow-insensitive, it is sound for concurrent programs because it implicitly considers all possible interactions and interleavings between tasks that may happen in a concurrent program. However, our proposed analysis is flow-sensitive since for the inference of the object-sensitive recurrence relations in Section 6, it is fundamental to track flow-sensitive relations among objects.

It is known that flow-sensitive analysis of concurrent programs is challenging due to the complexity of their flow. All possible task interleavings must be considered in order to develop a sound analysis. As our contribution in this regard, we extend the analysis of [32] to make it flow-sensitive in the presence of concurrent behaviours. The concurrency model guarantees that both fields and local variables can only be modified locally in the active task (i.e., in a flow-sensitive way) until the processor is released. At such release points, the values of local variables cannot be changed, whereas the state of the fields might be modified by other tasks. The main idea of our approach is to keep the flow-sensitive and the flow-insensitive abstractions separate. Besides, as we will see in the analysis, not all information on fields has to be lost. By keeping track of the values of the reference *this*, we can notably reduce information loss.

4.1. The Abstract Domain

The term *allocation site* refers to program points in which objects are created by executing a **new** instruction. Let S=$\{1, \ldots, n\}$ be the set of all allocation sites in a program. Given $q \in \mathbb{N}$, where $1 \le q \le n$, we define the set S^q as:

$$S^q = \{s \equiv \langle i_1, \ldots, i_p\rangle \mid i_j \in \{1, \ldots, n\}, 1 \le j \le p, |s| = q\}$$

where $|s|$ stands for the number of elements in the tuple $\langle i_1, \ldots, i_p\rangle$. Now, given a constant $k \in \mathbb{N}$, the analysis considers a finite set of *object names*, denoted $\mathcal{N}^k$, which is defined as $\mathcal{N}^k = \{o_s \mid o_s \equiv \epsilon \vee s \in S^q, 1 \le q \le k\}$.

For example, if $S=\{1,2\}$, then $S^1=\{\langle 1\rangle,\langle 2\rangle\}$, $S^2=\{\langle 1,1\rangle,\langle 2,2\rangle,\langle 1,2\rangle,\langle 2,1\rangle\}$ and $\mathcal{N}^2=\{\epsilon,o_{\langle 1\rangle},o_{\langle 2\rangle},o_{\langle 1,1\rangle},o_{\langle 2,2\rangle},o_{\langle 1,2\rangle},o_{\langle 2,1\rangle}\}$. Note that k defines the maximum size of sequences of allocation sites, and it allows controlling the precision of the analysis. Allocation sequences have in principle unbounded length and, thus, to ensure termination it is sometimes necessary to lose precision during analysis. This is done by just keeping the k rightmost positions in sequences whose length is greater than k. We define the operation $\oplus_k$ as:

$$\langle i_1,\ldots,i_p\rangle \oplus_k i \;=\; \begin{cases} \langle i\rangle & k=1 \\ \langle i_1,\ldots,i_p,i\rangle & k>1 \wedge |\langle i_1,\ldots,i_p,i\rangle| \le k \\ \langle i_{(p+2)-k},\ldots,i_p,i\rangle & k>1 \wedge |\langle i_1,\ldots,i_p,i\rangle| > k \end{cases}$$

Note that a variable in a program can be assigned objects with different object names. In order to represent all possible objects pointed to by a variable, sets of object names are used. Given a program, the set S of all allocation sites for it and $k \geq 0$, the *abstraction* of an object created in the program is an element of $\mathcal{N}^k$. Furthermore, $o_{\langle i_1,\ldots,i_p\rangle}$ represents all run-time objects that were created at program point i_p when the enclosing instance method was invoked on an object which was in turn created at program point i_{p-1}.

As notation, we use $\mathcal{V}$ to represent the set of all possible *reference* local variables that may occur in a program and $\mathcal{F}^*$ to represent all possible pairs (o_s, f) which denote all possible accesses to the *reference* field f through the objects $o_s \in \mathcal{N}^k$. In what follows, such pairs are represented as $o_s.f$.

Following Milanova's approach, context sensitivity is achieved by maintaining multiple replicas of each reference variable x for each possible context in which x may be used for calling a method. Let x be a local variable and o an object name to which *this* may point to, we use the fresh variable name x^o to store the analysis information for x and context o. We drop the superscript o when it is not relevant. The set of *replicas* is defined by $\mathcal{R}:\mathcal{V}\times\mathcal{N}^k\mapsto\mathcal{V}^*$, where $\mathcal{V}^*=\{x^o \mid o\in\mathcal{N}^k, x\in\mathcal{V}\}$.

Definition 4 (points-to abstract state)**.** *An abstract state is a tuple $\langle\phi,\theta\rangle$ where ϕ is a mapping $\phi:\mathcal{V}^*\cup\mathcal{F}^*\mapsto\wp(\mathcal{N}^k)$, and θ is a mapping $\theta:\mathcal{F}^*\mapsto\wp(\mathcal{N}^k)$.*

In an abstract state $\langle\phi,\theta\rangle$, $\phi(x^o)$ is the set of object names that represents the flow-sensitive information regarding all possible objects that may be assigned to the local variable x when *this* points to the object name o, and analogously for $\phi(o.f)$; and $\theta(o.f)$ is the set of object names that represents the flow-insensitive information regarding all possible objects that may be assigned to the field f for the object name o.

The abstract domain is the lattice $\langle AS,\top,\bot,\sqcup,\sqsubseteq\rangle$, where AS is the set of abstract states and $\top$ is the top of the lattice defined as $\langle\phi_\top,\theta_\top\rangle$ s.t. $\forall x^o\in\mathcal{V}^*,\phi_\top(x^o)=\mathcal{N}^k$, and $\forall o.f\in\mathcal{F}^*,\phi_\top(o.f)=\theta_\top(o.f)=\mathcal{N}^k$. The bottom element of the lattice is $\bot$, i.e., $\forall x^o\in\mathcal{V}^*,\phi_\bot(x^o)=\emptyset$, $\forall o.f\in\mathcal{F}^*,\phi_\bot(o.f)=\theta_\bot(o.f)=\emptyset$. Given two abstract states $\langle\phi_1,\theta_1\rangle$ and $\langle\phi_2,\theta_2\rangle$, we use $\langle\phi,\theta\rangle=\langle\phi_1,\theta_1\rangle\sqcup\langle\phi_2,\theta_2\rangle$ to denote that $\langle\phi,\theta\rangle$ is the least upper bound, defined as $\forall x^o\in\mathcal{V}^*,\phi(x^o)=\phi_1(x^o)\cup\phi_2(x^o)$ and $\forall o.f\in\mathcal{F},\phi(o.f)=\phi_1(o.f)\cup\phi_2(o.f)$ and $\theta(o.f)=\theta_1(o.f)\cup\theta_2(o.f)$. Similarly, $\langle\phi_1,\theta_1\rangle\sqsubseteq\langle\phi_2,\theta_2\rangle$ holds iff $\forall x^o,\phi_1(x^o)\subseteq\phi_2(x^o)$ and $\forall o.f\in\mathcal{F}^*,\phi_1(o.f)\subseteq\phi_2(o.f)$ and $\theta_1(o.f)\subseteq\theta_2(o.f)$.

4.2. *The Transfer Function*

Our proposed analysis is a standard forward analysis that assigns an abstract state to each program point by relying on a transfer function defined as follows.

Definition 5 (points-to transfer function)**.** *Given a set of object names $\mathcal{N}^k$, the set of abstract states AS and the set of instructions in the program Ins, the points-to transfer function τ is defined as a mapping $\tau:\wp(\mathcal{N}^k)\times Ins\times AS\mapsto AS$ computed according to the table in Figure 4.*

We use *This* to represent the set of object names which currently approximate the value of *this*, namely $\phi(\mathit{this})$. We assume that the considered instruction b is located at program point i (noted as $i{:}b$ in Figure 4), that x,y are reference variables and that f, g are reference fields. It is important to note that modifications to local variables (rows 1-4) affect ϕ in a flow-sensitive way (i.e., updates on variables overwrite the previous abstract value). Fields behave differently (row 5), since they

	$i : b$	$\tau(\mathit{This}, b, \langle\phi,\theta\rangle)$	
(1)	$x := \mathbf{new}\ C$	$\langle\phi[x^l \mapsto \{l \oplus_k i\}], \theta\rangle$	$\forall l \in \mathit{This}$
(2)	$x := y$	$\langle\phi[x^l \mapsto \phi(y^l)], \theta\rangle$	$\forall l \in \mathit{This}$
(3)	$x := \mathit{this}.f$	$\langle\phi[x^l \mapsto \phi(l.f)], \theta\rangle$	$\forall l \in \mathit{This}$
(4)	$x := \mathbf{null}$	$\langle\phi[x^l \mapsto \emptyset], \theta\rangle$	$\forall l \in \mathit{This}$
(5)	$\mathit{this}.f := y$	$\langle\phi[l.f \mapsto \phi(y^l)], \theta[l.f \mapsto \phi(y^l) \cup \theta(l.f)]\rangle$	$\forall l \in \mathit{This}$
(6)	$\mathbf{call}(\mathbf{b}, m(\mathit{this}, \bar{z}, \bar{y})$	$\mathit{interp}(\langle\phi,\theta\rangle, \mathit{This}, m(\mathit{this}, \bar{z}, \bar{y}))$	
(7)	$\mathbf{call}(\mathbf{m}, m(x, \bar{z}, y)$	$\sqcup\{\mathit{interp}(\langle\phi,\theta\rangle, \phi(x^l), m(x, \bar{z}, y)) \mid l \in \mathit{This}\}$	
(8)	$\mathbf{await}\ x?$	$\langle\phi[l.f \mapsto \phi(l.f) \cup \theta(l.f)], \theta\rangle$	$\forall l \in \mathit{This}, \forall f \in \bar{f}_C$
(9)	$x := y.\mathbf{get}$	$\langle\phi[x^l \mapsto \phi(y^l)], \theta\rangle$	$\forall l \in \mathit{This}$
(10)	$\mathit{otherwise}$	$\langle\phi, \theta\rangle$	

Figure 4. Transfer Function (where $l \equiv o_{\langle i_1,\ldots,i_p\rangle}$, $l \oplus_k i \equiv o_{\langle i_1,\ldots,i_p\rangle \oplus_k i}$, and $\bar{f}_C$ is the set of fields of the objects pointed to by This).

are replicated in both ϕ and θ. The mapping θ maintains global, flow-insensitive information for fields, whereas ϕ keeps flow-sensitive local updates. At the beginning of a method (row 7) and at release points (row 8) flow-insensitive information for fields is added to ϕ, as fields might have been modified by other tasks. Calls to blocks (row 6) are handled by $\mathit{interp}(\langle\phi,\theta\rangle, \mathit{This}, m(\mathit{this}, \bar{z}, \bar{y}))$, which (a) looks up the definition for block m, let us say $m(\mathit{this}, \bar{z}', \bar{y}') \leftarrow \ldots$, (b) projects $\bar{z}$ using ϕ to fit the calling context of m, resulting in a new mapping ϕ', (c) uses ϕ' and θ to analyze m, and (d) after the analysis, which can modify θ, gets the analysis output ϕ'' and modifies ϕ to set the new values for $\bar{y}$, namely $\phi[\bar{y}^l \mapsto \phi''((\bar{y}')^l)]$. Calls to methods (row 7) differ from calls to blocks in two aspects: the abstract value of x, i.e., $\phi(x^l)$, is used instead of This, and the values for all fields of x^l must be updated with θ for each $r \in \phi(x^l)$ to produce the mapping ϕ' to analyze m, i.e., $\phi'[r.g \mapsto \theta(r.g)], \forall r \in \phi(x^l), \forall g \in \bar{f}_D$, where D is the class of the objects referenced by x^l. In addition, the abstract value returned by the execution of a method is stored in the local future variable y, in order to be retrieved by a **get** instruction (row 9). The analysis of loops requires iterating the corresponding code several times until a fix-point is reached. The analysis merges abstract states at join points (i.e., after if and at loop entries) using the join operation $\sqcup$, and thus the corresponding abstract states can only grow, which in turn guarantees convergence of the analysis because the abstract domain is finite. Note that the points-to analysis computes, for each program point i, a pair $\langle\phi_i, \theta_i\rangle$. When it is clear from the context, we will omit the sub-index i from $\langle\phi,\theta\rangle$.

Example 5 (points-to analysis on the running example)**.** Figure 5 shows (part of) the result of applying the points-to analysis to the running example with $k = 2$. This is the smallest k for which no information is lost when handling object names. For simplicity, we omit tuples when it is clear from the context, i.e., we use $o_{i_1 \ldots i_p}$ to denote $o_{\langle i_1,\ldots,i_p\rangle}$. Brackets are also omitted in singleton sets. Keeping track of the value of the *this* reference is crucial for the precision of the points-to analysis. All object creations use the object name(s) pointed to by *this* to generate new object names by adding the current allocation site. E.g., at program point ③, *this* may be either $\mathit{this} \mapsto o_1$ or $\mathit{this} \mapsto o_2$; the new object names created are o_{13} and o_{23}, respectively. Observe that we keep the calling context as a superscript to the variable such that rd^{o_1} denotes the abstract value for rd when *this* is o_1. The value of *this* within a method comes from the object name(s) for the variable used to call the method. The example only shows ϕ from the transfer function, since only local variables are changed. Assignments to field variables would affect θ accordingly, as mentioned above. ■

The following example illustrates how points-to analysis deals with fields.

Example 6 (points-to analysis with reference fields)**.** Figure 6 shows an example of the points-to analysis used in a program with fields. The points-to analysis generates ϕ for each program point as shown to the right, and $\theta = \{o_a.f \mapsto \{o_{ab}, o_{ac}\}\}$. The program shown to the left first creates an object of class C_1 and then calls m_1 and m_2 on the newly created object. The variable f is a reference field of class C_1 and is modified at program points ⓑ and ⓒ. Therefore, at the end of the points-to analysis, $o_a.f$ may refer to either o_{ab} or o_{ac}, as it is stated in θ. At the beginning of m_1,

IR Program	ϕ
$readBlock(this, \langle\rangle, \langle r\rangle) \leftarrow$	$\{this \mapsto o_2\}$
$\ldots,$	$\{this \mapsto o_2\}$
$\mathbf{call}(\mathbf{b}, while(this, \overline{inp}, \overline{out})),$	$\{this \mapsto o_2\}$
$r := res.$	$\{this \mapsto o_2\}$
$while(this, \overline{inp}, \overline{out}) \leftarrow i \leq 0.$	$\{this \mapsto o_2\}$
$while(this, \overline{inp}, \overline{out}) \leftarrow i > 0,$	$\{this \mapsto o_2\}$
$\mathbf{call}(\mathbf{b}, if(\overline{inp}, \overline{out})).$	$\{this \mapsto o_2\}$
$if(this, \overline{inp}, \overline{out}) \leftarrow \ldots$	$\{this \mapsto o_2\}$
$if(this, \overline{inp}, \overline{out}) \leftarrow \ldots,$	$\{this \mapsto o_2\}$
$\mathbf{call}(\mathbf{b}, if^c(this, \overline{inp}, \overline{out})).$	$\{this \mapsto o_2\}$
$if^c(this, \overline{inp}, \overline{out}) \leftarrow$	$\{this \mapsto o_2\}$
$\mathbf{call}(\mathbf{m}, readContent(this, \langle pos, incr\rangle, \langle f\rangle)),$	$\{this \mapsto o_2\}$
$\ldots.$	$\{this \mapsto o_2\}$
$readOnce(this, \langle\rangle, \langle r\rangle) \rightarrow$	$\{this \mapsto o_1\}$
$\mathbf{call}(\mathbf{m}, readContent(this, \langle 0, lth\rangle, \langle f\rangle)),$	$\{this \mapsto o_1\}$
$\mathbf{await}\ f?,$	$\{this \mapsto o_1\}$
$r := f.\mathbf{get}.$	$\{this \mapsto o_1\}$
$readContent(this, \langle pos, elems\rangle, \langle r\rangle) \rightarrow$	$\{this \mapsto \{o_1, o_2\}\}$ ⓡ
③Reader_$init(\langle fp, elems\rangle, \langle rd\rangle),$	$\{this \mapsto \{o_1, o_2\}, rd^{o_1} \mapsto o_{13}, rd^{o_2} \mapsto o_{23}\}$
$\mathbf{call}(\mathbf{m}, process(rd, \langle pos\rangle, \langle f\rangle)),$	$\{this \mapsto \{o_1, o_2\}, rd^{o_1} \mapsto o_{13}, rd^{o_2} \mapsto o_{23}\}$ ⓟ
$\mathbf{await}\ f?,$	$\{this \mapsto \{o_1, o_2\}, rd^{o_1} \mapsto o_{13}, rd^{o_2} \mapsto o_{23}\}$
$r := f.\mathbf{get}.$	$\{this \mapsto \{o_1, o_2\}, rd^{o_1} \mapsto o_{13}, rd^{o_2} \mapsto o_{23}\}$
$main(this, \langle\rangle, \langle\rangle) \rightarrow$	$\{this \mapsto \epsilon\}$
①FileIS_$init(ob_1, \langle$"A.txt"$, 20, 2\rangle, \langle\rangle),$	$\{this \mapsto \epsilon, ob_1 \mapsto o_1\}$
②FileIS_$init(ob_2, \langle$"A.txt"$, 20, 3\rangle, \langle\rangle),$	$\{this \mapsto \epsilon, ob_1 \mapsto o_1, ob_2 \mapsto o_2\}$
$\mathbf{call}(\mathbf{m}, readOnce(ob_1, \langle\rangle, \langle f_1\rangle)),$	$\{this \mapsto \epsilon, ob_1 \mapsto o_1, ob_2 \mapsto o_2\}$
$\mathbf{call}(\mathbf{m}, readBlock(ob_2, \langle\rangle, \langle f_2\rangle)).$	$\{this \mapsto \epsilon, ob_1 \mapsto o_1, ob_2 \mapsto o_2\}$

Figure 5. Points-to analysis results for the running example.

IR Program	ϕ
$main(this, \langle\rangle, \langle\rangle) \rightarrow$	
ⓐ$w := \mathbf{new}\ C_1,$	$\{this \mapsto o_\epsilon, w^{o_\epsilon} \mapsto o_a\}$
$C_{init}(w, \langle\mathbf{null}\rangle, \langle\rangle),$	$\{this \mapsto o_\epsilon, w^{o_\epsilon} \mapsto o_a\}$
$\mathbf{call}(\mathbf{m}, m_1(w, \langle\rangle, \langle\rangle)),$	$\{this \mapsto o_\epsilon, w^{o_\epsilon} \mapsto o_a\}$
$\mathbf{call}(\mathbf{m}, m_2(w, \langle\rangle, \langle\rangle)).$	$\{this \mapsto o_\epsilon, w^{o_\epsilon} \mapsto o_a\}$
$m_1(this, \langle\rangle, \langle r\rangle) \leftarrow$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}\}$
ⓑ$y := \mathbf{new}\ C_2(),$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}, y^{o_a} \mapsto o_{ab}\}$
⊕$this.f := y,$	$\{this \mapsto o_a, o_a.f \mapsto o_{ab}, y^{o_a} \mapsto o_{ab}\}$
$\mathbf{call}(\mathbf{m}, m_3(this, \langle\rangle, \langle ff\rangle)),$	$\{this \mapsto o_a, o_a.f \mapsto o_{ab}, y^{o_a} \mapsto o_{ab}\}$
$x := this.f,$	$\{this \mapsto o_a, o_a.f \mapsto o_{ab}, y^{o_a} \mapsto o_{ab}, x^{o_a} \mapsto o_{ab}\}$
⊛$\mathbf{await}\ ff?,$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}, y^{o_a} \mapsto o_{ab}, x^{o_a} \mapsto o_{ab}\}$
$r := 0.$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}, y^{o_a} \mapsto o_{ab}, x^{o_a} \mapsto o_{ab}\}$
$m_2(this, \langle\rangle, \langle\rangle) \leftarrow$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}\}$
ⓒ$\ldots, u := \mathbf{new}\ C_2,$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}, u^{o_a} \mapsto o_{ac}\}$
$this.f := u.$	$\{this \mapsto o_a, o_a.f \mapsto o_{ac}, u^{o_a} \mapsto o_{ac}\}$
$m_3(this, \langle\rangle, \langle\rangle) \leftarrow$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}\}$
$\ldots, z := this.f, \ldots$	$\{this \mapsto o_a, o_a.f \mapsto \{o_{ab}, o_{ac}\}, z^{o_a} \mapsto \{o_{ab}, o_{ac}\}\}$

Figure 6. Points-to analysis for fields.

$\phi(o_a.f)$ is set to the flow-insensitive information stored in θ, but local information stored in ϕ can be constrained to $\{o_{ab}\}$ at program point ⊕, as $this.f$ is assigned a fresh object. Local information for the field f is valid until a release point is reached at ⊛. Therefore, $\phi(x)$ is set to the local value for f, $\{o_{ab}\}$. From program point $\circledast$ downwards, local information regarding f is updated with the information in θ, as other methods might modify f in the **`await`** instruction, as it happens in m_2. Regarding m_2 and m_3, points-to information for f is initially set to the global information stored in θ at the beginning of those methods, although it is constrained locally in m_2 where f is assigned, similarly as it is done in m_1. ■

The next theorem states the soundness of the analysis. To that end, let us assume that object identifiers in the semantics are of the form $\langle Oid, s\rangle$, where Oid is a unique identifier and s is the (unbounded) allocation sequence for the object. The semantics can be easily adapted to these identifiers by setting the function $newRef()$ for generating fresh object identifiers to $newRef(i, o_{this})$, where i is the program point where the object was created and o_{this} is the object identifier for $this$, i.e., $o_{this} = \langle Oid, l\rangle$. This function returns a unique reference as a pair $\langle Oid', l \oplus_\infty i\rangle$ ($\oplus_\infty$ stands for unbounded allocation site concatenation). In order to relate the concrete semantics to the results inferred by the points-to analysis, we use *name*(o) to refer to the object name in $\mathcal{N}^k$ that represents the concrete object identifier o. Concretely, *name*(o) is the longest suffix of length at most k of the unbounded allocation sequence of o encoded in the object identifier.

Theorem 1. *Let P be a program, $\mathcal{T} \equiv \mathcal{S}_0 \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}_n$ a trace, and $\langle \phi_i, \theta_i\rangle$ the result of the points-to analysis for every program point i in P. For every trace step $\mathcal{S}_j \rightsquigarrow_o^{i:b} \mathcal{S}_{j+1}$, $0 \leq j < n$, and for every object* $\mathtt{ob}(o, C, h, \langle tv, \_\rangle, \_) \in \mathcal{S}_{j+1}$*, the following holds:*

a) $name(o) \in \phi_i(this)$;
b) *If* $x \in dom(tv)$ *is a local reference variable,* $tv(x) \neq \mathsf{null}$ *and* $s = name(tv(x))$*, then* $s \in \phi_i(x^{name(o)})$;
c) *If* $f \in dom(h)$ *is a reference field of class* C*,* $h(f) \neq \mathsf{null}$ *and* $q = name(h(f))$*, then* $q \in \phi_i(name(o).f)$.

5. FIELD-SENSITIVE SIZE ANALYSIS FOR CONCURRENT OO PROGRAMS

The objective of size analysis is to infer *size abstractions* which allow reasoning on how the sizes of data change along a program's execution, which is fundamental for bounding the number of iterations that loops perform. Intuitively, the cost of executing a loop can be then obtained by multiplying the cost of each iteration by the number of iterations that it performs. This processed is formalized by means of the recurrence equations presented in Section 6 which integrate the size relations computed in the section.

5.1. *The Basic Size Analysis*

We present the size analysis in two steps: we first recall the notion of size measure that maps variables and values to their sizes; and we then present an abstraction which compiles instructions into size constraints, keeping as much information on global data (i.e., fields) as possible, while still being sound in concurrent executions.

Recall that the language on which we develop our analysis is deliberately simplified so that it only considers numerical and reference types, and thus the size analysis that we present in this section will also be restricted to such types. However, our implementation supports other data-types, in particular *String* and *user-defined* algebraic data-types, that we omit for the sake of simplifying the formal presentation. In Section 5.2, we comment on the additional bits required to handle these types in the size analysis.

Size Measures. For numerical data, the size is the actual numerical value. On the contrary, references require a more sophisticated treatment. A commonly used size measure is *path-length* [38], which counts the number of elements of the longest chain of references that can be traversed through the initial object (e.g., length of a list, depth of a tree, etc.). However, in our

	B	$\alpha_\rho(B)$	ρ'
(1)	$x\ op\ y$	$\rho(x)\ op\ \rho(y)$	$\rho'{=}\rho$
(2)	$x\ op'\ y$	$\_$	$\rho'{=}\rho$
(3)	$\mathbf{null} \mid x \mid this.f \mid n$	$0 \mid \rho(x) \mid \rho(f) \mid n$	$\rho'{=}\rho$
(4)	$\mathbf{await}\ x?$	$\perp$	$\rho'{=}\rho[\bar{f}_C \mapsto fresh(\bar{f}_C)]$
(5)	$x := y.\mathbf{get} \mid x := e$	$\rho'(x){=}\rho(y) \mid \rho'(x){=}\alpha_\rho(e)$	$\rho'{=}\rho[x \mapsto fresh(x)]$
(6)	$this.f := y$	$\rho'(f){=}\rho(y)$	$\rho'{=}\rho[f \mapsto fresh(f)]$
(7)	$x := \mathbf{new}\ C$	$\rho'(x){=}1$	$\rho'{=}\rho[x \mapsto fresh(x)]$
(8)	$\mathbf{call}(\mathbf{b}, q(rec, \bar{x}, \bar{y}))$	$q(\rho(rec), \rho(\bar{x}\cdot\bar{f}_C), \rho'(\bar{y}\cdot\bar{f}_C))$	$\rho'{=}\rho[\bar{y}{\cdot}\bar{f}_C{\mapsto}fresh(\bar{y}{\cdot}\bar{f}_C)]$
(9)	$\mathbf{call}(\mathbf{m}, q(rec, \bar{x}, y))$	$q(\rho(rec), \rho(\bar{x}), \rho'(y))$	$\rho'{=}\rho[y \mapsto fresh(y)]$
(10)	$otherwise$	$true$	$\rho'{=}\rho$

where in case (1) $op \in op_R \cup \{+,-\}$ and in (2) $op' \in \{*,/\}$

Figure 7. Abstract compilation. $\mathtt{ABST}(B_{k:i},\rho){=}\langle\alpha_\rho(B_{k:i}),\rho'\rangle$

context, objects are intended to simulate concurrent computing entities and not data structures. Thus, it is not common that they affect the number of iterations that loops perform. Therefore, ignoring their sizes is sound and precise enough in most cases. A slightly more precise abstraction distinguishes between the case in which a reference variable points to an object (size 1) or to null (size 0). The size of a future variable is the same as the size of the value it holds. This is sound since such variables can be used only through **get**, and the instruction **get** blocks the execution until the variable has a value.

Abstract Compilation. Modeling shared memory is a main challenge in static analysis of OO programs. Our starting point is [5], which models fields as local variables when the field to be tracked satisfies: (1) its memory location does not change; and (2) it is always accessed through the same reference (i.e., not through aliases). Both conditions can often be proven statically and the transformation of fields into local variables can then be applied for many fragments of the program. If we ignore concurrency, this approach could be directly adopted for our language. However, concurrency introduces new challenges.

Example 7 (treatment of fields in release points)**.** Consider the loop in the readBlock method in Figure 1. Ignoring the **await** instruction, the above soundness conditions (1) and (2) hold for the field blockS, and hence, we can track it as if it was a local variable. In a concurrent setting, however, while readBlock is executing, another task in the same object might modify blockS. Therefore, when analyzing readBlock, we cannot assume that the value of blockS is locally trackable. For instance, readBlock might introduce non-termination if we add a method void p() {blockS = blockS − 2; } to class FileIS. When the **await** is executed inside the loop, method p might change the value of blockS to a non-positive value, and thus the loop counter i would not decrement. ■

Handling fields requires identifying program points at which the shared memory might be modified by other tasks. This can happen when: (1) an **await** is explicitly executed, and thus allows other tasks (of the same object) to run; and (2) an asynchronous invocation is issued, and until the called method starts to execute, the fields of the called object may be modified by other tasks. We refer to such program points as *release points*. The above observation suggests that in a sequence of instructions not including **await**, the shared memory can be tracked locally. However, the values in the shared memory when a method starts to execute may not be identical to those when it was called. We first present a safe abstraction which loses all information at release points and at method entries. In a second step we discuss accuracy improvements at the release points in Section 5.3.

An abstract state is a set of linear constraints whose solutions define possible concrete states. This representation allows describing relations that are essential for inferring cost and proving termination, e.g., the size of x decreases by 1 in two consecutive states. The building blocks for this representation are constraints that describe the effect of each instruction b on a given state. In order to abstract instructions and guards, we use a mapping ρ from variables and field names to constraint variables that represent their sizes in the state before executing it.

Definition 6 (abstract compilation)**.** *Let B be an instruction or a guard. We define its* abstract compilation $\mathtt{ABST}(B, \rho)$ *w.r.t. a mapping ρ as $\langle \alpha_\rho(B), \rho' \rangle$, where $\langle \alpha_\rho(B), \rho' \rangle$ is computed according to the abstraction in Figure 7.*

Let us describe the abstraction of some selected instructions. First, note that, except for method/block calls and **`await`** instructions, $\alpha_\rho(B)$ returns a constraint and ρ' is a new mapping that refers to the sizes in the state after executing B. In Figure 7, given the variables $x_1, \ldots, x_n$ (respectively the fields $f_1, \ldots, f_n$), function $\mathit{fresh}(x_1, \ldots, x_n)$ (respectively $\mathit{fresh}(f_1, \ldots, f_n)$) returns n fresh variable names (respectively field names). In Line 5, the instruction $x := e$ is abstracted into the equality $\rho'(x) = \alpha_\rho(e)$, where $\alpha_\rho(e)$ is the size of e w.r.t. ρ. As mentioned before, $\rho'(x)$ (respectively $\rho(x)$) refers to the size of x *after* (respectively *before*) executing the instruction. The abstraction of **`await`** at Line 4 "forgets" sizes of those fields $\bar{f}_C$ of class C. This is because they might be updated by other methods that take the control when the current task suspends. When abstracting a call to a block in Line 8, the class fields $\bar{f}_C$ are added as arguments in order to track their values. However, when abstracting calls to methods (Line 9) the fields are not added. For methods, they are not added because their values at call time might not be the same as when the method actually starts to execute. Since we use linear constraints only, non-linear arithmetic expressions (Line 2) are abstracted to a fresh constraint variable "_" that represents any value. A program P is transformed into an abstract program P^α, that approximates its behaviour w.r.t. a size measure, by abstracting its rules as follows.

Definition 7 (abstract compilation of a rule)**.** *Given $r \equiv m(\mathit{this}, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$, and an identity map ρ_0 over $\mathit{vars}(r) \cup \bar{f}_C$, the abstract compilation of r is $r^\alpha \equiv m(\mathit{this}, \bar{I}, \rho_{n+1}(\bar{O})) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$ where:*

1. *$\mathtt{ABST}(g, \rho_0) = \langle g^\alpha, \rho_1 \rangle$, $\mathtt{ABST}(b_i, \rho_i) = \langle b_i^\alpha, \rho_{i+1} \rangle$, $1 \le i \le n$;*
2. *If m is a block then $\bar{I} = \bar{x} \cdot \bar{f}_C$ and $\bar{O} = \rho_{n+1}(\bar{y} \cdot \bar{f}_C)$; and*
3. *If m is a method then $\bar{I} = \bar{x}$ and $O = \rho_{n+1}(y)$.*

The size abstraction $\mathtt{ABST}(r)$ *for the rule r is $g^\alpha \wedge b_1^\alpha \wedge \ldots \wedge b_n^\alpha$ and $\mathcal{C}(\mathtt{ABST}(r)) = \bigwedge b_i^\alpha$ such that b_i^α is a linear constraint.*

Note that, according to line 8 in Figure 7, when abstracting a rule corresponding to a block (item 2 in Definition 7), we add the fields $\bar{f}_C$ to the input parameters. In what follows, we sometimes represent conjunctions of linear constraints $\varphi_1 \wedge \ldots \wedge \varphi_n$ as sets of the form $\{\varphi_1, \ldots, \varphi_n\}$.

Once the abstract compilation of the rule has finished, the renaming ρ_{n+1} coming from $\mathtt{ABST}(b_n, \rho_n) = \langle \alpha_{\rho_n}(b_n), \rho_{n+1} \rangle$ is applied to the output variables and fields. However, when abstracting a method rule (item (3)) fields are not considered in the head of the method. This matches line 9 in Figure 7.

Example 8 (abstract compilation for the running example)**.** The following is the abstract compilation of the block rule if_c in Figure 2, where $\overline{\mathit{inp}}$, $\overline{\mathit{out}}$ and $\overline{F}$ denote, respectively, the input parameters $\mathit{res}, i, \mathit{incr}, \mathit{pos}$, the output parameters $\mathit{res}'', i'', \mathit{incr}', \mathit{pos}''$ and the fields $\mathit{fp}, \mathit{lth}, \mathit{blockS}$. The substitution ρ_0 stands for the identity mapping on $\mathit{this}, \overline{\mathit{inp}}, \overline{\mathit{out}}$ and $\overline{F}$. The number at the right of each instruction indicates the line in Figure 7 used to compute the abstract compilation.

$if_c(this, \langle \overline{inp}, \overline{F}\rangle, \langle \overline{out}, \overline{F}''\rangle) \leftarrow$	ρ_0	
ⓐ $readContent(this, \langle pos, incr\rangle, \langle f'\rangle)$,	$fresh(f) = f', \rho_1 = \rho_0[f \mapsto f']$	(9)
ⓑ $\perp$,	$fresh(\overline{F}) = \overline{F}', \rho_2 = \rho_1[\overline{F} \mapsto \overline{F}']$	(4)
$v' = f'$,	$fresh(v) = v', \rho_3 = \rho_2[v \mapsto v']$	(5)
$res' = res + v'$	$fresh(res) = res', \rho_4 = \rho_3[res \mapsto res']$	(5)
$i' = i - incr$	$fresh(i) = i', \rho_5 = \rho_4[i \mapsto i']$	(5)
$pos' = pos + incr$	$fresh(pos) = pos', \rho_6 = \rho_5[pos \mapsto pos']$	(5)
ⓒ $while(this, \langle res', i', incr, pos', \overline{F}'\rangle,$	$\rho_7 = \rho_6[res {\mapsto} res'', i {\mapsto} i'', incr {\mapsto} incr',$	
$\langle res'', i'', incr', pos'', \overline{F}''\rangle).$	$pos \mapsto pos'', \bar{F} \mapsto \bar{F}'']$	(8)

where in ⓒ, function $fresh(res, i, inc, pos, \bar{F})$ returns $res'', i'', incr', pos'', \bar{F}''$. Furthermore, the output parameters of if_c are the result of applying ρ_7 to $res, i, incr, pos, fp, lth, blockS$. According to the abstraction in Figure 7, at ⓑ **await** is abstracted to $\perp$ and the information on fields is lost. At ⓒ the fields are added to the call in order to keep track of their values, however, when calling a method at ⓐ, the abstraction "forgets" this information.

Let us consider now method readBlock. The abstract compilation of the method results in:

$readBlock(this, \langle\rangle, \langle r'\rangle) \leftarrow$	ρ_0	
$res' = 0,$	$fresh(res) = res', \rho_1 = \rho_0[res \mapsto res']$	(5)
$i' = lth,$	$fresh(i) = i', \rho_2 = \rho_1[i \mapsto i']$	(5)
$incr' = 0,$	$fresh(incr) = incr', \rho_3 = \rho_2[incr \mapsto incr']$	(5)
$pos' = 0$	$fresh(pos) = pos', \rho_4 = \rho_3[pos \mapsto pos']$	(5)
$while(this, \langle res', i', incr', pos', \overline{F}\rangle,$	$\rho_5 = \rho_4[res \mapsto res'', i \mapsto i'',$	
$\langle res'', i'', incr'', pos'', \overline{F}'\rangle),$	$incr \mapsto incr'', pos \mapsto pos'', \bar{F} \mapsto \bar{F}']$	(8)
$r' = res''.$	$fresh(r) = r', \rho_6 = \rho_5[r \mapsto r']$	(5)

where in ρ_5, function $fresh(res, i, incr, pos, \bar{F})$ returns $res'', i'', incr'', pos'', \bar{F}'$. According to Def. 7, the head of the rule does not contain fields but the call to $while$ does. ■

An abstract program P^α basically abstracts the behaviour of the original program with respect to a size measure α. An abstract state has the form $\mathcal{A} \circ \phi$, where $\mathcal{A} \equiv \{a_1^\alpha, \ldots, a_n^\alpha\}$, a_i^α is an *abstract object* of the form $\langle \bar{b}^\alpha, \bar{\rho}\rangle$ and ϕ is a linear constraint. In order to formalize the operational semantics for an abstract program, we modify the presentation of the abstract rules, by storing also the renamings computed in Def. 6. Therefore abstract rules are now of the form $p(rec, \bar{x}, \bar{y}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha \circ \rho_0 \cdots \rho_{n+1}$, where $\rho_0, \ldots, \rho_{n+1}$ is the tuple of all renamings that were used during abstract compilation of that specific rule.

The operational semantics for an abstract program is given by the transition system in Figure 8, which simply accumulates the constraints (when possible) and proceeds to execute the calls in the body of the rules. Rules $(1)_\alpha$ and $(2)_\alpha$ correspond to calls to blocks and methods respectively. In both cases a renamed apart abstract rule is selected from P^α. For the sake of simplicity, we avoid another renaming step, by assuming that the renamed apart rule is retrieved with the same input $\bar{x}$ and output $\bar{y}$ variables that appear in the call. Rules $(4)_\alpha$ and $(5)_\alpha$ correspond to the abstraction of an **await** instruction since **await** is abstracted to $\perp$ (see Figure 7). In particular, when an **await** instruction is evaluated to *true* (rule (9)) the execution proceeds. This is simulated by rule $(4)_\alpha$. Similarly, if the evaluation of the **await** instruction fails (rule (10)), then the context of the suspended task is introduced in the queue of pending tasks for the current object. Hence, in rule $(5)_\alpha$ the abstraction $\perp$ of the corresponding **await** is kept until the associated **await** succeeds and rule $(4)_\alpha$ can be applied.

Finally note that rules (6) and (7) in Figure 3 handle those cases in which an asynchronous method or block call, respectively, have finished the execution. This fact is captured by rule $(6)_\alpha$ that, as done in rules (6) and (7), simply removes the abstract execution context. Note also that, after applying rules (6) and (7), since the execution context is ϵ, rule (11) can be applied to select

$$(1)_\alpha\ \frac{p(\mathit{this},\bar{x},\bar{y}) \leftarrow g^\alpha, b_1^\alpha,\ldots,b_n^\alpha \circ \rho_0 \cdots \rho_{n+1} \ll P^\alpha, g^\alpha \wedge \phi \not\models \mathit{false}}{\{\langle \mathbf{call}(\mathbf{b}, p(\mathit{this},\bar{x},\bar{y}))\cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \leadsto_\alpha \{\langle b_1^\alpha \cdots b_n^\alpha \cdot \bar{b}^\alpha, \rho_1\cdots\rho_{n+1}\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \wedge g^\alpha}$$

$$(2)_\alpha\ \frac{p(\mathit{rec},\bar{x},y) \leftarrow b_1^\alpha,\ldots,b_n^\alpha \circ \rho_1 \cdots \rho_{n+1} \ll P^\alpha}{\{\langle \mathbf{call}(\mathbf{m}, p(\mathit{rec},\bar{x},y))\cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \leadsto_\alpha \{\langle \bar{b}^\alpha,\bar{\rho}\rangle, \langle b_1^\alpha \cdots b_n^\alpha, \rho_1\cdots\rho_{n+1}\rangle|\mathcal{A}\} \circ \phi}$$

$$(3)_\alpha\ \frac{\varphi \wedge \phi \not\models \mathit{false}}{\{\langle \varphi\cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \leadsto_\alpha \{\langle \bar{b}^\alpha,\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \wedge \varphi}$$

$$(4)_\alpha\ \frac{}{\{\langle \bot\cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \leadsto_\alpha \{\langle \bar{b}^\alpha,\bar{\rho}\rangle|\mathcal{A}\} \circ \phi}$$

$$(5)_\alpha\ \frac{}{\{\langle \bot\cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \leadsto_\alpha \{\langle \bot\cdot\bar{b}^\alpha,\rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi}$$

$$(6)_\alpha\ \frac{}{\{\langle \epsilon, \rho\rangle|\mathcal{A}\} \circ \phi \leadsto_\alpha \{\epsilon|\mathcal{A}\} \circ \phi}$$

Figure 8. Semantics of Abstract Programs

some new task from the queue of the corresponding object for execution. In the case of the abstract semantics, such task exists inside the abstract configuration and thus it can be selected at any time of the computation. Finally, rule $(3)_\alpha$ basically accumulates constraints whenever it is possible. The notation $\varphi \wedge \phi \not\models \mathit{false}$ means that $\varphi \wedge \phi$ is satisfiable.

The following example shows briefly the correspondence between concrete and abstract traces. In order to simplify the example, we ignore renamings in the abstract states and focus on the abstract rules that we apply.

Example 9 (correspondence between concrete and abstract traces)**.** Consider the following IR program and its associated abstract compilation:

$p(\mathit{this},\langle z\rangle,\langle f\rangle) \leftarrow$ $\quad f := z+1.$ $\mathit{main}(\mathit{this},\langle\rangle,\langle\rangle) \leftarrow$ $\quad a := \mathbf{new}\ A(),$ $\quad z := 4,$ $\quad \mathbf{call}(\mathbf{m}, p(a,\langle z\rangle,\langle f\rangle))$ $\quad \mathbf{await}\ f?$	$p(\mathit{this},\langle z\rangle,\langle f_2\rangle) \leftarrow$ $\quad f_2 = z+1.$ $\mathit{main}(\mathit{this},\langle\rangle,\langle\rangle) \leftarrow$ $\quad a_1 = 1,$ $\quad z_1 = 4,$ $\quad p(a_1,\langle z_1\rangle,\langle f_1\rangle),$ $\quad \bot.$

Let us consider the following concrete trace (we show the numbers of the $\leadsto$-rules applied only and the resulting state):

$$\{\mathsf{ob}(\mathsf{main},\bot,\bot,\langle tv_{main}, \mathbf{call}(\mathbf{b}, \mathit{main}(\mathit{this},\langle\rangle,\langle\rangle))\rangle,\emptyset)\} \leadsto^* \quad (4),(3),(1),(5)$$
$$\{\mathsf{ob}(\mathsf{main},\bot,\bot,\langle tv^1_{main}, \mathbf{await}\ f?\rangle,\emptyset), \mathsf{ob}(o_a, A, h, \epsilon, \{\langle tv_a, f_3 := z_2+1\rangle\}),$$
$$\mathtt{fut}(\mathtt{fn},\bot)\}$$

the renaming used in rule (5) is $p(\mathit{this}_1,\langle z_2\rangle,\langle f_3\rangle) \leftarrow f_3 := z_2+1$ and $tv^1_{main}(a) = o_a$, $tv^1_{main}(z) = 4$, $tv^1_{main}(f) = \mathtt{fn}$, $tv_a(\mathit{this}_1) = o_a$, $tv_a(z_2) = 4$, $tv_a(\mathtt{ret}) = (\mathtt{fn}, f_3)$. At this point, we can easily build a $\leadsto_\alpha$ trace as follows:

$$\{\langle \mathbf{call}(\mathbf{b}, \mathit{main}(\mathit{this},\langle\rangle,\langle\rangle)),\_\rangle\} \circ \mathit{true} \leadsto^*_\alpha \quad (1)_\alpha,(3)_\alpha,(3)_\alpha,(2)_\alpha$$
$$\{\langle\bot,\_\rangle,\langle f_3 = z_2+1,\_\rangle\} \circ z = 4 \wedge z_2 = 4$$

Suppose now that we apply rule (10) to the concrete trace on object main. Then, since the future variable is not ready, the task execution context is introduced in the queue of object main, resulting in:

$\{\;\; \texttt{ob}(\mathsf{main}, \bot, \bot, \epsilon, \{\langle tv^1_{main}, \textbf{await } f?\rangle\}),$
$\quad\; \texttt{ob}(o_a, A, h, \epsilon, \{\langle tv_a, f_3 := z_2 + 1\rangle\}), \texttt{fut}(\texttt{fn}, \bot)\}$

The point now is that we can apply rule $(5)_\alpha$ on the abstract state and obtain exactly the same abstract state, i.e., we delay the abstract trace until the **await** succeeds. Let us apply rule (11) on object o_a in order to extract the unique task from its queue, followed by rule (1). The resulting concrete state is:

$$\{\texttt{ob}(\mathsf{main}, \bot, \bot, \epsilon, \{\langle tv^1_{main}, \textbf{await } f?\rangle\}), \texttt{ob}(o_a, A, h, \langle \epsilon, tv^1_a\rangle, \emptyset), \texttt{fut}(\texttt{fn}, \bot)\}$$

where $tv^1_a(f_3) = 5$. On the abstract state we can apply rule $(3)_\alpha$ to compute $\{\langle \bot, \_\rangle, \langle \epsilon, \_\rangle\} \circ z = 4 \wedge z_2 = 4 \wedge f_3 = z_2 + 1$. Note that rule $(3)_\alpha$ corresponds only to the application of rule (1) in Figure 3 but rule (11) has not correspondence with any abstract rule. This is because in abstract states there no exists the notion of queue since all tasks are in the same pool. Hence steps corresponding to the application of rule (11) are simply ignored by the abstract operational semantics. Finally, on the concrete state, using rule (7), we can fix the value of the future variable as follows:

$$\{\texttt{ob}(\mathsf{main}, \bot, \bot, \epsilon, \{\langle tv^1_{main}, \textbf{await } f?\rangle\}), \texttt{ob}(o_a, A, h, \epsilon, \emptyset), \texttt{fut}(\texttt{fn}, 5)\}$$

and afterwards apply rules (11), (9) and (6) and obtain:

$$\{\texttt{ob}(\mathsf{main}, \bot, \bot, \epsilon, \emptyset), \texttt{ob}(o_a, A, h, \epsilon, \emptyset), \texttt{fut}(\texttt{fn}, 5)\}$$

For the case of the abstract state, it is enough to apply, first rule $(6)_\alpha$ (corresponding to the application of rule (7)) to compute $\{\langle \bot, \_\rangle, \epsilon\} \circ z = 4 \wedge z_2 = 4 \wedge f_3 = z_2 + 1$, afterwards rule $(4)_\alpha$ (corresponding to rule (9)) what results in $\{\langle \epsilon, \_\rangle, \epsilon\} \circ z = 4 \wedge z_2 = 4 \wedge f_3 = z_2 + 1$, and finally rule $(6)_\alpha$ to obtain $\{\epsilon, \epsilon\} \circ z = 4 \wedge z_2 = 4 \wedge f_3 = z_2 + 1$. ■

We now establish the *soundness* of the abstract compilation with respect to the chosen size measure α. Intuitively, we prove that the size of the variables in a given concrete trace is computed in a corresponding abstract trace. As notation, given an object $a \equiv \texttt{ob}(o, C, h, \langle tv, \bar{b}\rangle, \mathcal{Q})$, we say that $\langle tv, \bar{b}\rangle$ is its *active task* (denoted by $active(a)$) and we define $pending(a) = \{tk \mid tk \in \mathcal{Q}\}$ as the set of pending tasks of a. Thus, we define the set of tasks for an object a, denoted as $tasks(a)$, as $\{active(a)\} \cup pending(a)$. Finally, given an state $\mathcal{S} = \{a_1, \ldots, a_n\}$, we define the set of tasks in $\mathcal{S}$, denoted as $tasks(\mathcal{S})$, as $\cup_{i=1}^n tasks(a_i)$.

Definition 8 (relation between abstract and concrete states)**.** *Let $\mathcal{S}$ be a concrete state. We say that an abstract state $\mathcal{A} \circ \phi$* approximates *$\mathcal{S}$, denoted as $\mathcal{A} \circ \phi \approx \mathcal{S}$ if an only if:*

1. *ϕ is satisfiable;*
2. *for all $\langle tv, b_1 \cdots b_n\rangle \in tasks(\mathcal{S})$, from an object $\texttt{ob}(o, C, h, \_, \_) \in \mathcal{S}$, there exists $\langle b^\alpha_1 \cdots b^\alpha_n, \rho_1 \cdots \rho_{n+1}\rangle \in \mathcal{A}$ and an assignment $\sigma : vars(tv) \cup \bar{f}_C \mapsto \mathbb{Z}$ such that $\mathit{ABST}(b_i, \rho_i) = \langle b^\alpha_i, \rho_{i+1}\rangle$, $1 \le i \le n$, $\sigma \models \phi$ and:*
   - *For all $x \in dom(tv)$ such that $tv(x) \in \mathbb{Z}$, it holds that $\sigma(\rho_1(x)) = tv(x)$;*
   - *For all $f \in \bar{f}_C$ such that $h(f) \in \mathbb{Z}$, it holds that $\sigma(\rho_1(f)) = h(f)$;*
   - *For all $y \in dom(tv)$, if $tv(y) = \texttt{fn}$, $\texttt{fut}(\texttt{fn}, v) \in \mathcal{S}$ and $v \neq \bot$, then $\sigma(\rho_1(y)) = v$.*

Example 10 (equivalence between concrete and abstract traces)**.** Consider the concrete and abstract states in Example 9, corresponding to the same execution step:

$$\{\texttt{ob}(\mathsf{main}, \bot, \bot, \epsilon, \{\langle tv^1_{main}, \textbf{await } f?\rangle\}), \texttt{ob}(o_a, A, h, \epsilon, \emptyset), \texttt{fut}(\texttt{fn}, 5)\}$$
$$\{\langle \bot, \_\rangle, \epsilon\} \circ \phi$$

where $\phi = z = 4 \wedge z_2 = 4 \wedge f_3 = z_2 + 1$. Let us select σ such that $\sigma \models \phi$, i.e., $\sigma(z) = 4 = \sigma(z_2)$, and $\sigma(f_3) = 5$. Then it holds that $tv^1_{main}(z) = 4 = \sigma(z)$ and $\sigma(z_2) = 4 = \sigma(z)$. Furthermore note that since $tv^1_{main}(f) = \texttt{fn}$ and $\texttt{fut}(\texttt{fn}, 5)$ belongs to the concrete state, σ satisfies that $\sigma(f_3) = 5$. ■

The following theorem establishes that given a concrete trace, we can generate an abstract trace of the same length and instantiate it (i.e., give the integer values to all constraints variables using a consistent assignment σ) in such a way that the size of a variable in the i-th concrete state coincides with the value of the corresponding constraint variable in the i-th abstract state.

Theorem 2 (soundness of abstract compilation)**.** *Let P be an IR program, P^α the abstract compilation of P and $\mathcal{S}_0 = \{\mathsf{ob}(\mathtt{main}, \bot, \bot, \langle tv_0, \boldsymbol{\mathtt{call}}(\boldsymbol{b}, \ \mathsf{main}(this, \langle\rangle, \langle\rangle))\rangle, \emptyset)\}$ an initial state. If $\mathcal{S}_0 \leadsto^n \mathcal{S}_n$ then there exists an abstract trace $\mathcal{A}_0 \circ true \leadsto^n_\alpha \mathcal{A}_n \circ \phi_n$, such that $\mathcal{A}_0 \equiv \{\langle \boldsymbol{\mathtt{call}}(\boldsymbol{b}, \mathsf{main}(this, \langle\rangle, \langle\rangle)), \ \rho_0 \cdot \rho_1\rangle\}$ and for all $\mathcal{S}_i$, it holds that $\phi_n \models \phi_i$ and $\mathcal{S}_i \approx \mathcal{A}_i \circ \phi_i$, $0 \leq i \leq n$.*

5.2. Handling Strings and Algebraic Data-Types

As already mentioned, our implementation handles *String* and *user-defined* algebraic data-types (e.g., lists, trees, etc). Below we describe how such types are handled in the size analysis. For each case (1) we describe corresponding size measures that allow abstracting data of such types to numerical values; and (2) we describe corresponding abstract compilation for instructions that manipulate such types.

String data-type. Strings are abstracted to their length. This is a classical size measure that allows bounding the number of iterations of loops that traverse strings. In the abstract compilation phase, instructions that manipulate strings are abstracted to linear constraints that describe relations between the lengths of the strings on which they operate. For example, the instruction $\mathsf{c} = \mathsf{strapp}(\mathsf{a}, \mathsf{b})$, which concatenates strings a and b into a new one c, is compiled into the constraint $c = a + b$ to indicate that the length of c is as the sum of the lengths of a and b. In addition, we add the constraint $a \geq 0 \wedge b \geq 0 \wedge c \geq 0$ to indicate that the length is a non-negative measure. Other string manipulating instructions are treated similarly.

Algebraic data-types. A classic size measure used for algebraic data-types, mainly in the context of termination analysis, is the *term-size* norm [14] which abstracts data-structures to the number of occurrences of type constructs in the data-structure. For example, the size of the list $\mathsf{Cons}(\mathsf{F}(\mathsf{a}, \mathsf{b}),$ $\mathsf{Cons}(\mathsf{F}(\mathsf{a}, \mathsf{c}), \mathsf{Nil}))$ is 9, where each occurrence of a type construct from $\{\mathsf{Cons}, \ \mathsf{Nil}, \mathsf{F}, \mathsf{a}, \mathsf{b}, \mathsf{c}\}$ contributes 1. In the abstract compilation phase, instructions that manipulate data-structures are abstracted to linear constraints that describe relations between the term-size of the data-structures on which they operate. For example, the instruction $\mathsf{ys} := \mathsf{Cons}(\mathsf{x}, \mathsf{xs})$, which constructs a list whose head is x and whose tail is xs, is abstracted to $ys = 1 + x + xs \wedge ys \geq 0 \wedge x \geq 0 \wedge xs \geq 0$. Our implementation allows choosing between the *term-size* and the *term-depth* measure (which abstracts data-structures to their depth in a similar way). Besides, a recent extension includes state-of-the-art size measures that are automatically extracted from the user-defined types [12]. Such *type-based* norms do not count all type constructs of a given data-structures, but rather only those that are potentially traversed by loops. For example, the size of the list above would be 3 because it will only count `Cons` and `Nil`, which would lead to a more precise bound for a loop that traverses this list but not its internal elements. Moreover, our implementation allows using several type-based size measures simultaneously, which in turn allows abstracting one data-structure using several size measures. This is particularly useful when the different parts of a data-structure are traversed by different parts of the program.

5.3. Class Invariants in Cost Analysis

The accuracy of the size analysis can be improved by using a generalization of *class invariants* (see, e.g., [31]). As discussed in Section 5.1, release points are problematic since at these points other task(s) may modify the values of shared fields. However, it is often possible to gather useful information about the shared variables, in the form of class invariants, which must hold at those points. In sequential programs, class invariants have to be established by constructors and must hold on termination of all (public) methods of the class. They can be assumed at (public) method entry but may not hold temporarily at intermediate states not visible outside the object. In our context, we need that such invariants hold on method termination and also at all release points of all methods. This way, we can use them to improve the abstraction at the release points. In the following, given a class C, Ψ_C denotes the class invariant for class C, which is a set of linear constraints over the fields of C and possibly some constant symbols.

Definition 9 (abstract compilation with class invariants)**.** *Let B be an instruction or a guard and Ψ_C a class invariant for C. We define the* abstract compilation $\mathtt{ABST}^I(B,\rho,\Psi_C)$ *of B w.r.t. a mapping ρ and a class invariant Ψ_C as $\mathtt{ABST}^I(B,\rho,\Psi_C) = \mathtt{ABST}(B,\rho)$, if $B \neq$ **await** $x?$ and $\mathtt{ABST}^I(B,\rho,\Psi_C) = \langle \perp \wedge \Psi_C[\bar{f}_C \mapsto \rho'(\bar{f}_C)], \rho'\rangle$ otherwise, where $\mathtt{ABST}(B,\rho) = \langle \perp, \rho' \rangle$.*

The definition above allows us to define the *abstract compilation* of a rule $r \equiv m(this,\bar{x},\bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$ of class C w.r.t. a class invariant Ψ_C defined in terms of $\bar{f}_C$ as $r^\alpha \equiv m(this,\bar{I},\rho_{n+1}(\bar{O})) \leftarrow \Phi_C \wedge g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$, where r^α is computed as in Def. 7, using $\mathtt{ABST}^I$ instead of $\mathtt{ABST}$.

Example 11 (class invariants for the running example)**.** The following invariants will be required in order to obtain the cost of all methods of our running example: (1) In class Reader, we need to know that field $elems$ is bounded, i.e., $0 \leq elems \leq elems_{max}$ where $elems_{max}$ is a constant symbol that bounds the value of $elems$; and (2) in class FileIS, we also need to know that field lth is bounded, i.e., $0 \leq lth \leq lth_{max}$. Furthermore, we need the invariant $blockS = blockS_{init}$ for the loop in method readBlock. Thus, $\Phi_{Reader} = \{0 \leq elems \leq elems_{max}\}$ and $\Phi_{FileIS} = \{0 \leq lth \leq lth_{max} \wedge blockS = blockS_{init}\}$.
Now, if we consider the abstract compilation of block if_c in Example 8, and we use as invariant Φ_{FileIS}, ⓑ would be replaced $\perp \wedge \rho_2(\Phi_{FileIS})$, where $\rho_2(\Phi_{FileIS}) = 0 \leq lth' \leq lth_{max} \wedge blockS' = blockS_{init}$. ■

The invariants above can be inferred automatically, for instance, by means of a syntactic analysis that simply checks that the corresponding fields are initialized and never updated again. Note that even if several processes modify lth we can still obtain the upper bounds that we have computed before. This is because there is no loop whose termination relies on the value of lth. Observe that the loop in readBlock first copies the value of lth into variable i and then the termination depends on variable i. Thus, if we have several instances of method readBlock interleaving their computations, we can still prove their termination and infer their resource consumption.

Furthermore, even if a field is modified at a release point, we can use the points-to analysis of Section 4 to determine if the field is read by means of references different from those used to write the field. If this is the case, then such a field can be preserved in rule 4 of Figure 7. In the following, given a rule $m(this,\bar{x},\bar{y}) \rightarrow g, b_1, \ldots, b_n$, we use $body(m)$ to refer to the multiset of instructions $\{b_1, \ldots, b_n\}$. Now, we define the set $Read(C,f) = \{o \mid x := this.f \in body(m), o \in \phi_m(this), m$ is a rule in C, $m \not\equiv C_{init}\}$ of references used to read a field f in class C. Similarly, we can define the set $Write(C,f)$ but considering instructions of the form $this.f := x$ in rules of class C. Then we define the set $trackable(C)$ as the set $\{f \in \bar{f}_C \mid Read(C,f) \cap Write(C,f) = \emptyset\}$.

Example 12 (automatic inference of class invariants)**.** Let us consider the field blockS in Figure 1. Then, since $\phi_{readBlock}(this) = \{o_2\}$ (see Example 5), and the field is only read, then $Read(\mathsf{FileIS},\mathsf{blockS}) = \{o_1\}$ but $Write(\mathsf{FileIS},\mathsf{blockS}) = \emptyset$. Hence $\mathsf{blockS} \in trackable(\mathsf{FileIS})$ i.e., this field is not lost when processing the instruction **await** $f?$ inside the while loop in $readBlock$. For fields lth and fp, it holds that $Read(\mathsf{FileIS},\mathsf{lth}) = \{o_2\}$, $Write(\mathsf{FileIS},\mathsf{lth}) = \emptyset$, $Read(\mathsf{FileIS},\mathsf{fp}) = \{o_1,o_2\}$ and $Write(\mathsf{FileIS},\mathsf{fp}) = \emptyset$. Hence both fields belong to $trackable(\mathsf{FileIS})$. Thus $trackable(\mathsf{FileIS}) = \{\mathsf{blockS},\mathsf{lth},\mathsf{fp}\}$. ■

Differently to the sequential setting in [5], a field satisfying that $Read(C,f) = Write(C,f) = \{o\}$, i.e., the field is read and written using the same reference, cannot be considered trackable. The following example illustrates this.

Example 13 (comparison with the sequential setting)**.** Consider the following two methods:

```
void m1(A o){
Fut<Int> x;
  while (this.f > 0) {
    x = o ! p();
    await x?;
    this.f = this.f - 1;
  }
}
```

```
void m2(A o){
Fut<Int> x;
  while (this.f < 0) {
    x = o ! p();
    await x?;
    this.f = this.f + 1;
  }
}
```

that belong to the same class C, where f is a field in C. Assume that the points-to analysis computes $\phi_{m1}(\mathit{this}) = \phi_{m2}(\mathit{this}) = \{o_1\}$. Then $\mathit{Read}(\mathsf{C}, \mathsf{f}) = \mathit{Write}(\mathsf{C}, \mathsf{f}) = \{o_1\}$. However, in a setting in which the execution of m1 and m2 are continuously interleaved because the corresponding **await** instructions do not hold, termination is not guaranteed, since m1 decreases f what endangers the termination of m2. Similarly, as m2 increases f, the termination m1 cannot be guaranteed. ■

The same idea can be also applied to points-to analysis where class invariants can be used to state which fields remain unchanged at release points. In this case, the class invariant Ψ_C^{pt} is a set of field names that are guaranteed to remain unchanged after their initialization. To take this information into account, row 8 of the transfer function in Figure 4 is changed to update ϕ only for those fields that are not in the class invariant, that is:

$$\tau(\mathit{This}, \langle \mathbf{await}\ x? \rangle, \langle \phi, \theta \rangle) = \langle \phi[l.f \mapsto \phi(l.f) \cup \theta(l.f)], \theta \rangle, \forall l \in \mathit{This}, \forall f \in \bar{f}_C \setminus \Psi_C^{pt}$$

Example 14 (class invariants for points-to analysis)**.** If $\Psi_C^{pt} = \{f\}$ was inferred for the program of Example 6, local information for the field f would be valid even after the release point at $\circledast$.

6. OBJECT-SENSITIVE RESOURCE ANALYSIS

In this section, we present the process of obtaining upper bounds on the resource consumption. Our analysis follows the classical two-fold approach to cost analysis [42] in which: (1) a program is first transformed into a set of cost relations [7] which (2) can then be solved into closed-form upper/lower bounds [4], i.e., cost expressions that are not in recursive form. The cost relations we generate can be solved using [4] without requiring any modification; thus we do not describe this second phase and, in what follows, we focus exclusively on the first phase of the resource analysis.

After showing an intuitive example in Section 6.1, the presentation of the analysis is performed in two steps. First, we illustrate in Section 6.2 how an object-insensitive analysis can be defined as in sequential programming, by using the size abstraction computed in Section 5, and point out its limitations. Then, Section 6.3 defines the object-sensitive analysis which, by relying on the object-sensitive points-to information of Section 4, overcomes the limitations of the object-insensitive analysis.

6.1. An Intuitive Example

Let us consider the following simple code (left) and its IR (right):

```
void m(A a,int n) {
  a.p(n);
  n++;
  A b=new A();
  b.p(n);
}
void p(int n){
  n++;
}
```

```
$m(\mathit{this}, \langle a, n \rangle, \langle\rangle) \leftarrow$
  $\mathbf{call}(\mathbf{m}, p(a, \langle n \rangle, \langle\rangle))$,
  $n := n + 1$,
  $b := \mathbf{new}\ A$,
  $\mathbf{call}(\mathbf{m}, p(b, \langle n \rangle, \langle\rangle))$.

$p(\mathit{this}, \langle n \rangle, \langle\rangle) \leftarrow$
  $n := n + 1$.
```

We want to infer automatically the number of instructions executed by m. The most relevant point is that method p is invoked from two different objects and with two different arguments. Intuitively, the transformation of an IR program into cost relations can be formalized by transforming each rule in the program into a cost equation, which accumulates the cost of the instructions in the rule and contains the applicability conditions for the equations resulting from the abstract program. Our analysis is based on the three components introduced in the previous sections as follows. First, (1) the cost models introduced in Section 3 are used by the analysis to determine the cost of each instruction. For instance, we now use the cost model that counts the number of executed instructions and apply it to each rule. We have that the rule for m, denoted m, accumulates 4 instructions and the one for p accumulates 1. Second, (2) the size relations in Section 5 are necessary to generate the applicability conditions (guards) for cost relations and to determine how the size of data changes when the equations are applied. In particular, we infer the equations:

$$\begin{array}{lcll} m(n) & = & 4 + p(n) + p(n') & \{n' = n + 1\} \\ p(n) & = & 1 & \end{array}$$

In the equation for m, we can observe that the size relation $\{n' = n + 1\}$ tells us that the size of n is increased by one in the second call to p. In this case, the equations have no guards as they apply unconditionally.

Up to this point, we have obtained object-insensitive equations, because we do not distinguish the object which is executing the instructions. In the third step, (3) the points-to analysis in Section 4 is necessary to define the *cost centers*, which are the artifacts used by the analysis to separate the cost of the distributed components so that we can then distinguish the resource consumption of each component. In the example, three cost centers are obtained: $c(\mathit{this})$ for the object executing m, $c(a)$ for the object that is passed as parameter to m, and $c(b)$ for the object created in m. Now, when we account for the cost of executing an instruction, we take into account the object that executes it. In particular, we obtain the following equations:

$$\begin{array}{lcll} m(n) & = & 4{*}c(\mathit{this}) + p_a(n) + p_b(n') & \{n' = n + 1\} \\ p_a(n) & = & c(a){*}1 & \\ p_b(n) & = & c(b){*}1 & \end{array}$$

We can observe that attached to the cost we add the cost center for the object executing the instruction. The 4 instructions for method m are attributed to this. When a method is executed from different objects, we create object-sensitive equations which distinguish all possible calling contexts. In the example, we generate two equations for method p, one in which the cost is attributed to a and one to b. Solving the equations results in the upper bound:

$$m(n) = 4 * c(\mathit{this}) + 1{*}c(a) + 1{*}c(b)$$

While the cost in this example is constant, in general the cost is a function of the data input sizes, what makes the problem more interesting and challenging.

6.2. *Object-Insensitive Analysis*

The generation of cost relations from our concurrent and distributed programs, for a generic cost model $\mathcal{M}$, can be done exactly as for sequential programs [7], by using the size abstractions in Section 5 which already take the concurrent behaviour into account, and then simply applying the generic cost model in Section 3 to each instruction of each rule.

Definition 10 (object-insensitive resource analysis)**.** *Let $\mathcal{M}$ be a cost model, P an IR program, $r \equiv m(\mathit{this}, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n$ a rule in P and $r^\alpha \equiv m(\mathit{this}, \bar{x}, y') \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$ its abstract compilation, where $\varphi \equiv \mathcal{C}(g^\alpha \wedge b_1^\alpha \wedge \ldots \wedge b_n^\alpha)$. Let $\mathit{calls}(r^\alpha) = \{p(\bar{w}) \mid b_i^\alpha \equiv$* ***call***$(ct, p(rec, \bar{w}, \_))$*, $1 \leq i \leq n\}$ be the multiset of calls to methods or blocks in r^α. The* cost equation *associated to r is defined as:*

$$m(\bar{x}) = \sum_{i=1}^{n} \mathcal{M}(b_i) + \sum_{p(\bar{w}) \in \mathit{calls}(r^\alpha)} p(\bar{w}), \quad \varphi$$

Given a program P, its cost relation system (CRS for short) is obtained by applying the above definition to all rules. The CRS is like a standard CRS for sequential programs with the following features: (i) equations do not have output arguments, as we aim at obtaining the cost as a function of the input argument sizes; (ii) given a rule being analyzed, its cost equation is obtained by applying the cost model $\mathcal{M}$ to each of the basic instructions in the body (first summation in the equation); (iii) a call in the program is substituted by a call to its corresponding cost equation (second summation in the equation); (iv) the linear constraints φ obtained from the size abstraction of the rule are attached to the rule to define its applicability conditions and the size relations among the variables in the equation. When we have class invariants available, they are added to the constraints in the equations. Finally, the CRS is called *object-insensitive* because it does not separate the cost per object, but rather it accumulates the cost carried out by all objects in the program.

Example 15 (object-insensitive cost equations)**.** Let us see the application of Definition 10 to the rule $readBlock$ (shown in Figure 2) w.r.t. the cost model $\mathcal{M}_i$ (see Section 3) that counts the number of executed instructions. First we apply $\mathcal{M}_i$ to all instructions in the abstract body of $readBlock$ (see Example 8), what results in the constant 6. In addition, we have to include the cost of the **`while`** loop, i.e., we add the call to $while$ with its corresponding arguments:

$$readBlock() \quad = \quad 6{+}while(i', blockS), \qquad \varphi$$

where $\varphi \equiv \{i'{=}lth, 0 \leq lth \leq lth_{max}, blockS = blockS_{init}\}$ and φ comes from the abstract compilation in Example 8 and the class invariant Φ_{FileIS} in Example 11. In what follows we only include the constraints of the class invariants that are relevant for obtaining the upper bounds. For the rule $readBlock$ we only include the part of the class invariant Φ_{FileIS} that is needed for obtaining the upper bound, $0 \leq lth \leq lth_{max}$. Moreover, note that, according to the abstract compilation of $readBlock$ in Example 8, the call to $while$ should have as input arguments $\langle res', i', incr', pos', fp, lth, blockS\rangle$. For the sake of clarity, we only include those arguments that are relevant for obtaining the cost, i.e., those involved in guards (see [6] for more details). In the case of $while$, we only include i' and $blockS$. Now, by applying Definition 10 to all rules in the IR of the running example, we obtain the following CRS:

$$\begin{array}{lcll}
main() & = & 14{+}readOnce() + readBlock() & \{\} \\
readBlock() & = & 6{+}while(i', blockS) & \{i'{=}lth, 0 \leq lth \leq lth_{max}\}, \\
while(i, blockS) & = & 0 & \{i \leq 0\} \\
while(i, blockS) & = & 1{+}if(i, blockS) & \{i{>}0, blockS = blockS_{init}\} \\
if(i, blockS) & = & 2{+}if^{c}(i, blockS, incr) & \{blockS > i, incr = i\} \\
if(i, blockS) & = & 2{+}if^{c}(i, blockS, incr) & \{blockS \leq i, incr = blockS\} \\
if^{c}(i, blockS, incr) & = & 7{+}readContent(){+} & \{i' = i{-}incr\} \\
 & & while(i', blockS) & \\
readOnce() & = & 4{+}readContent() & \{\} \\
readContent() & = & 7{+}process() & \{\} \\
process() & = & 4 + while_1(i, elems) & \{i{=}0, 0 \leq elems{\leq}elems_{max}\} \\
while_1(i, elems) & = & 0 & \{i{\geq}elems\} \\
while_1(i, elems) & = & 15{+}while_1(i', elems') & \{i{<}elems, i'{=}i+1\}
\end{array}$$

We assume that the executions of methods hdRead and update have constant costs, which are accounted in the constant 15 of the second equation for $while_1$. Likewise, the constant 7 in the equation for $readContent$ includes the cost of executing the constructor of class Reader which is assumed to be 2 (two fields are initialized). Note that the constraints of the equation $process$ include the class invariant of the class Reader, i.e., $\Phi_{Reader} = \{0 \leq elems \leq elems_{max}\}$. For brevity, we do not include Φ_{FileIS} in the equations $readContent$ and $readOnce$ as it is not relevant for solving the equations. The constraint $blockS = blockS_{init}$ from Φ_{FileIS} is only relevant in the equation $while$. Observe that the constraints capture (1) the conditions required to apply the rule, as well as (2) the constraints that state how their values are modified along the program execution, e.g., in the first equation for if we see that, (1) when $blockS > i$, (2) the value of i is updated, $i' = i - incr$. This CRS is solved using [4] (without requiring any modification to the solving process) into the following closed-form upper bounds:

$$
\begin{array}{lcl}
UB_{process}() & = & 4 + 15 * \mathsf{nat}(elems_{max}) \\
UB_{readContent}() & = & 11 + 15 * \mathsf{nat}(elems_{max}) \\
UB_{readOnce}() & = & 15{+}15 * \mathsf{nat}(elems_{max}) \\
UB_{readBlock}() & = & 6{+}\mathsf{nat}(lth_{max}){*}(21{+}15{*}\mathsf{nat}(elems_{max}))
\end{array}
$$

Let us explain the different parts of the upper bound computed for readBlock. The constant 6 comes from the constant in the equation for $readBlock$. We use the nat operator to avoid negative evaluations. Concretely $\mathsf{nat}(e)$ is defined as e if $e > 0$, and 0, otherwise. The cost of the loop is the following quadratic expression:

$$\mathsf{nat}(lth_{max}){*}(21{+}15{*}\mathsf{nat}(elems_{max}))$$

where $\mathsf{nat}(lth_{max})$ is an upper bound on the number of iterations of the loop and $21{+}15{*}\mathsf{nat}(elems_{max})$ is the worst-case cost of each iteration. At each iteration, method $readContent$ is invoked. This method contains a loop whose cost is linear on $elems_{max}$. Thus, the component $\mathsf{nat}(elems_{max})$ in the upper bound above is due to such method invocation.
As the cost equation for main includes the cost of $readOnce$ and $readBlock$, its upper bound is:

$$
\begin{array}{lcl}
UB_{main}() & = & 14 + \underbrace{15{+}15 * \mathsf{nat}(elems_{max})}_{readOnce} + \\
 & & \underbrace{6{+}\mathsf{nat}(lth_{max}){*}(21{+}15{*}\mathsf{nat}(elems_{max}))}_{readBlock}
\end{array}
$$ ■

The above analysis has a main drawback: it is not capable of distinguishing the different distributed components. Instead, the resource usage contributed by all objects is accumulated in a single cost center which corresponds to the whole execution of the distributed system.

6.3. Adding Cost Centers to the Equations

As its main novelty, CRS in our object-sensitive resource analysis uses cost centers in order to keep the resource usage corresponding to the different components separate. The main idea is to take advantage of the object-sensitive points-to information to generate cost equations for all possible contexts (and thus objects). In particular, the object-sensitive equations will allow us to count separately the cost that corresponds to different instances of objects that are created at the same allocation site but correspond to different object names and potentially different distributed components. Given a program rule whose first instruction is at program point h, we annotate the rule as follows:

$$r \equiv [m(this, \bar{x}, \bar{y})]^{\phi_h(this)} \leftarrow g, [b_1]^{O_1}, \ldots, [b_n]^{O_n} \in P$$

where O_i is defined as follows:

1. if b_i is a call of the form $\mathbf{call}(\mathbf{m}, m(rec, \bar{w}, z))$ occurring at program point j, then $O_i = \{\phi_j(rec^{o_l})^{o_l} | o_l \in \phi_h(this)\}$;
2. Otherwise, i.e., b_i is not a method call, then O_i is the empty set.

Observe that we are annotating the head of the rule with set the object names that might be $this$ for this rule, that is, $\phi_h(this)$. In addition we annotated the calls to methods with a set of sets: for each element in $\phi_h(this)$ we obtain the set of possible object names that might be pointed by rec, and, in addition, we keep as super-index the value of $this$ used for obtaining such set, $\phi_j(rec^{o_l})^{o_l}$.

Example 16 (annotated rules)**.** The annotated rules using the information gathered by points-to analysis with $k = 2$ are of this form:

$$[main(this, \langle\rangle, \langle\rangle)]^{\{\epsilon\}} \leftarrow \ldots,$$
$$\quad [\texttt{call}(\mathbf{m}, readOnce(ob_1, \langle\rangle, \langle f_1 \rangle))]^{\{\{o_1\}\}}, [\texttt{call}(\mathbf{m}, readBlock(ob_2, \langle\rangle, \langle f_2 \rangle))]^{\{\{o_2\}\}}, \ldots$$

$$[readBlock(this, \langle\rangle, \langle r \rangle)]^{\{o_2\}} \leftarrow \ldots, [\texttt{call}(\mathbf{b}, while(\overline{inp}, \overline{out}))]^{\{\{o_2\}\}}, \ldots$$

$$[if^{\,c}(\overline{inp}, \overline{out})]^{\{o_2\}} \leftarrow \ldots,$$
$$\quad [\texttt{call}(\mathbf{m}, readContent(this, \langle pos, incr \rangle, \langle f \rangle))]^{\{\{o_2\}\}}, \ldots$$

$$[readOnce(this, \langle\rangle, \langle r \rangle)]^{\{o_1\}} \leftarrow \ldots,$$
$$\quad [\texttt{call}(\mathbf{m}, readContent(this, \langle 0, lth \rangle, \langle f \rangle))]^{\{\{o_1\}\}}, \ldots$$

$$[readContent(this, \langle pos, incr \rangle, \langle f \rangle)]^{\{o_1, o_2\}} \leftarrow \ldots,$$
$$\quad [\texttt{call}(\mathbf{m}, process(rd, \langle pos \rangle, \langle f \rangle))]^{\{\{o_{13}\}^{o_1}, \{o_{23}\}^{o_2}\}}, \ldots$$

$$[process(this, \langle pos \rangle, \langle r \rangle)]^{\{o_{13}, o_{23}\}} \leftarrow \ldots$$

Some blocks of method readBlock are omitted since all of them are annotated with $\{o_2\}$, and also some irrelevant calls are omitted. As it is shown in Figure 5, at program point marked with $\textcircled{r}$, $\phi_{\textcircled{r}}(this) = \{o_1, o_2\}$, then we annotate the rule readContent with the set $\{o_1, o_2\}$. Similarly, process are annotated with $\{o_{13}, o_{23}\}$. At program point $\textcircled{p}$, shown in Figure 5, depending on the value of *this*, we have different object names which might be pointed by rd, $\phi_{\textcircled{p}}(rd^{o_1}) = o_{13}$ and $\phi_{\textcircled{p}}(rd^{o_2}) = o_{23}$. Then we annotate the call to process with $\{\{o_{13}\}^{o_1}, \{o_{23}\}^{o_2}\}$. ■

Now, given an annotated rule $r \equiv [m(this, \bar{x}, \bar{y})]^{This} \leftarrow g, [b_1]^{O_1}, \ldots, [b_n]^{O_n}$, we also annotate its abstract compilation as $r^\alpha \equiv [m(this, \bar{x}, \_)]^{This} \leftarrow g, [b_1^\alpha]^{O_1}, \ldots, [b_n^\alpha]^{O_n}$, and we use the following functions:

- $methods(r^\alpha)$ to obtain the multiset of annotated elements $[q(rec, \bar{w}, z)]^O$ which correspond to calls to methods of the form $[\texttt{call}(\mathbf{m}, q(rec, \bar{w}, z))]^O$ in the body of r^α;
- $blocks(r^\alpha)$ to refer to the multiset of elements $p(this, \bar{w}, \bar{z})$ which are calls to intermediate rules of the form $\texttt{call}(\mathbf{b}, p(this, \bar{w}, \bar{z}))$ in the body of r^α.

Definition 11 (object-sensitive resource analysis)**.** *Let $\mathcal{M}$ be a cost model, P an IR program, $r \equiv m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n$ a rule in P and $r^\alpha \equiv m(this, \bar{x}, y') \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$ its abstract compilation, where $\varphi \equiv \mathcal{C}(g^\alpha \wedge b_1^\alpha \wedge \ldots \wedge b_n^\alpha)$. Let us consider the annotated abstract rule $[m(this, \bar{x}, \bar{y})]^{This} \leftarrow \_$ for r, where $methods(r^\alpha) = \{[m_1(y_1, \bar{w}_1, z_1)]^{O_1}, \ldots, [m_k(y_k, \bar{w}_k, z_k)]^{O_k}\}$. Then the following set of equations defines the cost of r: for each $o \in This$, and for each $\langle o_1, \ldots, o_k \rangle \in O_1^o \times \cdots \times O_k^o$ such that $O_i^o \in O_i$, $1 \leq i \leq k$, we generate the equation:*

$$m\_o(\bar{x}) = \sum_{b \in body(r)} c(o) * \mathcal{M}(b) + \sum_{p(this, \bar{w}, \bar{z}) \in blocks(r^\alpha)} p\_o(\bar{w}) + m_1\_o_1(\bar{w}_1) + \cdots + m_k\_o_k(\bar{w}_k), \quad \varphi$$

where $m_i\_o_i$ is the name of the equation that represents a call to method m_i from object o_i, and $c(o)$ denotes the cost center associated to o.

Intuitively, the above definition generates, from one rule, as many equations as needed for defining its cost such that all possible contexts (i.e., object names of callees) are considered. The new names are obtained by concatenating the corresponding object name to the rule name. Besides, as regards method invocations, all combinations have to be generated. This is done in the definition by means of the Cartesian product $O_1^o \times \cdots \times O_k^o$ which gives us all possible combinations for the elements in the sets. The cost expressions we accumulate are multiplied by a symbolic expression $c(o)$ which denotes the cost center of the object on which the call is performed. As an example, if we have a rule, where m_2 and m_3 are calls to methods:

$$\begin{aligned} [m_1(this, \langle x \rangle, \langle y \rangle)]^{\{o_1, o_2\}} \quad \leftarrow \quad & [m_2(\_, \langle x, u \rangle, \langle\rangle)]^{\{\{o_3\}^{o_1}, \{o_4, o_5\}^{o_2}\}} + \\ & [m_3(\_, \langle u, y \rangle, \langle\rangle)]^{\{\{o_6, o_7\}^{o_1}, \{o_8\}^{o_2}\}} \end{aligned}$$

Intuitively, when *this* points to o_1, then x (in m_2) may point to o_3 and u to o_6 or o_7 (in m_3). Similarly, when *this* points to o_2, then x (in m_2) may point to o_4 or o_5 and u (in m_3) may point to o_8. From the above rule, the following four equations are generated to cover all cases:

$$\begin{array}{l} m_1\_o_1(x) = m_2\_o_3(x,u) + m_3\_o_6(u,y) \\ m_1\_o_1(x) = m_2\_o_3(x,u) + m_3\_o_7(u,y) \\ m_1\_o_2(x) = m_2\_o_4(x,u) + m_3\_o_8(u,y) \\ m_1\_o_2(x) = m_2\_o_5(x,u) + m_3\_o_8(u,y) \end{array}$$

Multiple rules for the same procedure are interpreted as non-deterministic choices and the upper bound solver computes the maximum over them. Therefore, the fact that multiple non-deterministic rules are introduced (e.g., two rules for $m_1\_o_1(x)$) does not degrade the quality of the upper bound obtained.

Example 17 (object sensitive cost equations for the running example)**.** In the running example, method readContent is executed by two different objects, and consequently. This is captured in the points-to analysis by means of two object names, o_1 and o_2 for the *this* reference of readContent (see Example 5). Nevertheless, the upper bound for main shown in Example 15, accumulates the cost executed by all objects together. By applying Definition 11 to the annotated rules in Example 16, the equations in the CRS (Example 15) are replicated for all possible object names that could execute the equations. For example, as the rule readContent is annotated with $\{o_1, o_2\}$ and the call to process is annotated with $\{\{o_{13}\}^{o_1}, \{o_{23}\}^{o_2}\}$, the replicated equations obtained by the object sensitive cost analysis are as follows:

$$\begin{array}{lcll} readContent\_o_1() & = & c(o_1)*7+process\_o_{13}() & \{\} \\ readContent\_o_2() & = & c(o_2)*7+process\_o_{23}() & \{\} \\ process\_o_{13}() & = & c(o_{13})*4+while_1\_o_{13}(i,elems) & \{i{=}0,\, 0 \leq elems{\leq}elems_{max}\} \\ process\_o_{23}() & = & c(o_{23})*4+while_1\_o_{23}(i,elems) & \{i{=}0,\, 0 \leq elems{\leq}elems_{max}\} \end{array}$$

Note that the replication of the rule *process* is analogous, but for the set $\{o_{13}, o_{23}\}$. The equations include the cost center corresponding to the equation responsible of accumulating such cost, i.e., the cost accumulated by the equation $readContent\_o_1$ is multiplied by $c(o_1)$. Using such cost centers, the closed-form upper bounds now keep separate the resource consumption associated to each cost center o_i by means of a symbolic constant $c(o_i)$. From the above equations for readContent the solver obtains the following upper bounds:

$$\begin{array}{lcl} UB_{readContent\_o_1}() & = & c(o_1)*7+\underbrace{c(o_{13})*(4+15*\mathsf{nat}(elems_{max}))}_{process\_o_{13}} \\ UB_{readContent\_o_2}() & = & c(o_2)*7+\underbrace{c(o_{23})*(4+15*\mathsf{nat}(elems_{max}))}_{process\_o_{23}} \end{array}$$

Similarly for *readOnce* and *readBlock*, which are annotated with o_1 and o_2 (respectively), we obtain the upper bounds:

$$\begin{array}{lcl} UB_{readOnce\_o_1}() & = & c(o_1)*4+\underbrace{c(o_1)*7+c(o_{13})*(4+15*\mathsf{nat}(elems_{max}))}_{readContent\_o_1} \\ UB_{readBlock\_o_2}() & = & c(o_2)*6+c(o_2)*\mathsf{nat}(lth_{max})*10+ \\ & & \mathsf{nat}(lth_{max})*(\underbrace{c(o_2)*7+c(o_{23})*(4+15*\mathsf{nat}(elems_{max}))}_{readContent\_o_2}) \end{array}$$

With the object sensitive cost analysis, in contrast to the upper bound obtained in Example 15, the cost centers added in the cost expressions the closed-form upper bound for main keeps the number of instructions executed on each object multiplied by its corresponding cost center:

$$\begin{array}{lcl} UB_{main}() & = & c(\epsilon)*14+\underbrace{c(o_1)*4+c(o_1)*7+c(o_{13})*(4+15*\mathsf{nat}(elems_{max}))}_{readOnce\_o_1} \\ & & \underbrace{c(o_2)*6+c(o_2)*\mathsf{nat}(lth_{max})*10+}_{readBlock\_o_2} \\ & & \underbrace{\mathsf{nat}(lth_{max})*(c(o_2)*7+c(o_{23})*(4+15*\mathsf{nat}(elems_{max})))}_{readBlock\_o_2} \end{array}$$

■

The upper bound for a set of objects $\mathcal{O}$, $UB_p|_{\mathcal{O}}$, is obtained by setting $c(o)$ to 1 for all object names $o \in \mathcal{O}$ and to 0 for the remaining ones. Note that, if we replace $c(o)$ by 1 (for all object names o), the accuracy of object-insensitive CRS in Def. 10 coincides with that of object-sensitive CRS.

Example 18 (object sensitive upper bound accuracy)**.** If we are interested in the number of instructions performed by the cost centers $\{o_{13}\}$ and by $\{o_{23}\}$, we replace the symbolic expression $c(o_{13})$, $c(o_{23})$, respectively, by 1 and the rest of cost centers by 0. Then,

$$UB_{main}()|_{\{o_{13}\}} = 4 + 15 * \mathsf{nat}(elems_{max})$$
$$UB_{main}()|_{\{o_{23}\}} = \mathsf{nat}(lth_{max}) * (4 + 15 * \mathsf{nat}(elems_{max}))$$

Such upper bound captures the instructions executed by process when we call it from readOnce. The main observation is that the accuracy of the upper bound for main is significantly better when the analysis is performed with $k = 2$ than with $k = 1$. If the points-to analysis is performed for $k = 1$, the object names o_{13} and o_{23} are merged in a single object name o_3, resulting in the upper bound $UB^{k=1}_{main}$:

$$\begin{aligned} UB^{k=1}_{main}() \quad = \quad & c(\epsilon) * 14 + c(o_1) * 4 + c(o_1) * 7 + \underline{c(o_3)} * (4 + 15 * \mathsf{nat}(elems_{max})) \\ & c(o_2) * 6 + c(o_2) * \mathsf{nat}(lth_{max}) * 10 + \\ & \mathsf{nat}(lth_{max}) * (c(o_2) * 7 + \underline{c(o_3)} * (4 + 15 * \mathsf{nat}(elems_{max}))) \end{aligned}$$

Therefore, it would not be possible to distinguish between the objects created at program point ③ and the costs are aggregated together resulting in a less precise upper bound that accumulates the expressions for $c(o_{13})$ and for $c(o_{23})$:

$$UB^{k=1}_{main}()|_{\{o_3\}} \quad = \quad (4 + 15 * \mathsf{nat}(elems_{max})) + \mathsf{nat}(lth_{max}) * (4 + 15{*}\mathsf{nat}(elems_{max})) \qquad \blacksquare$$

The following theorem relates the concrete cost $\mathcal{C}(\mathcal{T}, o, \mathcal{M})$ defined in Definition 3 with the upper bound inferred by the analysis.

Theorem 3 (soundness)**.** *Let P be a program and $\mathcal{S}_0$ an initial state. If $\mathcal{T} \equiv \mathcal{S}_0 \leadsto^n \mathcal{S}_n$, then for all object identifier o such that $\mathtt{ob}(o, \_, \_, \_, \_) \in \mathcal{S}_i$, $0 \le i \le n$, it holds that $\mathcal{C}(\mathcal{T}, o, \mathcal{M}) \le UB_{\mathsf{main}}()|_{\{name(o)\}}$.*

Given the soundness of the size and points-to analyses used to generate the equations, soundness of the object-sensitive cost analysis is proved by simply showing that the above CRS can be obtained by cloning the program as many times as determined by the number of object names computed by the points-to analysis and applying the standard object-insensitive cost analysis to each of the versions.

Finally, the use of cost centers easily allows us to instantiate our analysis with different deployment strategies. Such strategies determine the groups of objects that share the processor (see, e.g., JCobox [35]). The resource consumption of each group can be obtained by our approach by setting $c(o)$ to 1 for all object names o that belong to the group, and to 0 for the remaining ones.

7. EXPERIMENTAL EVALUATION

We have implemented our analysis in SACO [2], an analyzer of ABS programs which can be tried out online at: `http://costa.ls.fi.upm.es/saco/web/`. This section presents our experimental evaluation using the SACO system with a set of typical concurrent programs as benchmarks. The overall goal of the evaluation is to measure the accuracy and performance of our cost analysis. First, in Section 7.1 we evaluate the accuracy of the object insensitive analysis, by evaluating the obtained upper bounds against the actual cost obtained in real runs using a profiler. Then, Section 7.2 evaluates the accuracy and performance of the object sensitive analysis and the impact of using more precise points-to analyses. Finally, we evaluate and discuss the potential applications of our cost analysis using a larger and real application, namely, the TradingSystem, a case study developed within the FP7 HATS project `http://www.hats-project.eu`.

7.1. Object-Insensitive Experiments

In this section we evaluate the accuracy of the object insensitive analysis. This is done by comparing the actual number of executed instructions in real runs with a series inputs using the aPET system [9]

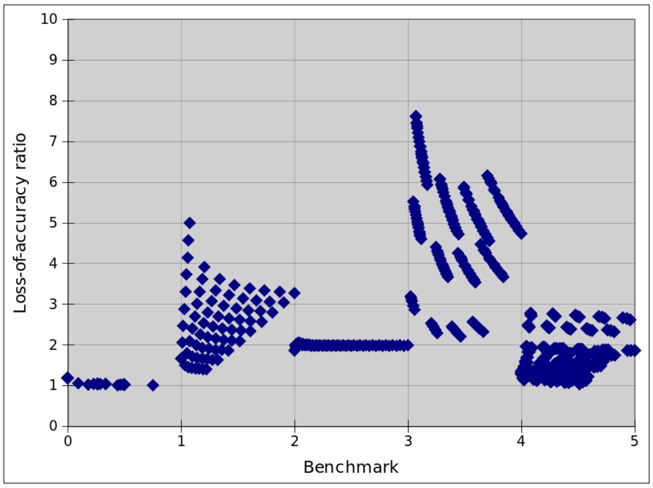


Figure 9. Evaluating precision of object-insensitive analysis

as profiler, against the estimated cost obtained by evaluating the generated upper bounds with the $\mathcal{M}_i$ cost model for the corresponding abstractions of the inputs. E.g., if a concrete list in algebraic form is used as input for one run, the corresponding evaluation of the upper bound is done with its term-size abstraction (see Section 5.2). The following typical concurrent applications have been used as benchmarks: PeerToPeer, a pure peer-to-peer file sharing application; BBuffer, a classical bounded-buffer for communicating several producers; BookShop, a web shop client-server application; Mail, a simple model of a Mail server; and DistHT, a distributed implementation of a hash table. The source code of all benchmarks is available at the SACO web page.

The chart in Figure 9 depicts the *loss-of-accuracy ratio* (Y axis) for each input of each benchmark (X axis). The loss-of-accuracy ratio is obtained by dividing the estimated cost by the real cost. Thus, the higher the ratio the less accurate the upper bound is, and the closer the ratio to one, the more precise it is. In the X axis, the range 0-1 corresponds to the different runs with the BBuffer benchmark, the range 1-2 to Mail, the range 2-3 to DistHT, the range 3-4 to PeerToPeer, and the range 4-5 to BookShop. Within each range, higher values correspond to higher real costs. The loss-of-accuracy ratio seems reasonable, namely from 1.1 to 7.7. For most runs it is less than 5, except for some runs of the PeerToPeer benchmark. It is important to note that we infer upper bounds on the worst-case cost, and thus it is clear that their quality varies when applying them to different inputs (because they must over-approximate the cost of a program for all possible inputs). Apart from this, there are several reasons why our analysis can lose precision, namely, the size measures for algebraic data structures (see Section 5.2), field maximizations, the precision of the underlying analyses, etc. All these factors directly affect the precision of the inferred loop bounds, which are the basic ingredients used to build upper bounds. It is important to note, however, that the ratio stays reasonably stable within each benchmark, and more importantly, that it does not increase for executions of higher cost, but it fluctuates with different combinations of inputs. In the case of the PeerToPeer benchmark it can be observed that the ratio tends to decrease for executions of higher cost.

7.2. *Object-Sensitive Experiments*

In this section we evaluate the accuracy and performance of object-sensitive cost analysis. We use the same benchmarks as in Section 7.1, plus an additional one, namely Chat, a chat application, which could not be used in the previous section since it could not be handled by the aPET profiler. In general, the more precise the points-to analysis is, the results of the cost analysis can be more precise as well. In this experimental evaluation, our objectives are: (1) to experimentally measure how an improvement in the precision of points-to analysis affects the precision of cost analysis; (2) to evaluate the impact of using more precise (and thus more costly) points-to analysis on the

Benchmark	#ev	k=1			k=2						
		#c	#e	T	#c	#e	T	$\%_c$	$\%_m$	$\%_M$	$\%_A$
BBuffer	1280	15	96	602	21	143	1078	40.0	0.2	36.2	12.9
DistHT	1458	10	78	1036	17	131	1808	70.0	20.4	44.0	30.8
MailServer	1250	7	93	644	13	159	1012	71.4	3.3	42.9	19.1
Chat	1250	12	81	231	15	104	275	25.0	13.3	20.6	16.4
BookShop	3072	7	116	1485	10	162	1969	42.9	1.8	17.4	10.5
PeerToPeer	864	11	306	14648	19	581	28019	72.7	39.4	48.5	45.0

Benchmark	k=3							k=4						
	#c	#e	T	$\%_c$	$\%_m$	$\%_M$	$\%_A$	#c	#e	T	$\%_c$	$\%_m$	$\%_M$	$\%_A$
BBuffer	35	253	1620	66.7	3.9	48.8	21.8	47	301	1645	68.6	2.5	47.0	24.4
DistHT	27	229	3324	58.8	13.4	44.0	32.9	39	385	4165	44.4	7.7	39.9	29.1
MailServer	13	159	1023	0.0	0.0	0.0	0.0	13	159	1027	0.0	0.0	0.0	0.0
Chat	17	118	288	76.0	3.8	19.2	10.0	19	118	305	2.4	0.4	0.5	0.4
BookShop	11	195	2047	40.0	5.6	24.3	18.2	11	195	2042	0.0	0.0	0.0	0.0
PeerToPeer	19	581	28312	0.0	0.0	0.0	0.0	19	581	28371	0.0	0.0	0.0	0.0

Table I. Statistics about the Object-Sensitive Resource Analysis (times in ms.)

efficiency of the overall cost-analysis, (3) to find out which value, or values, of k achieve the best balance between precision and performance. In order to tackle such questions, we have applied object-sensitive cost analysis with four different values of k (see Section 4), $k = 1$, $k = 2$, $k = 3$ and $k = 4$.

Table I summarizes the results obtained on an Intel Core 2 Duo at 2.53GHz with 4GB of RAM, running Linux 3.2.0. Columns **#c**, **#e** show, for each value of k, the number of cost centers identified by points-to analysis and the number of equations of the CRS, respectively. Column **T** shows the time taken to apply the overall cost analysis, including the generation of the CRS and the time to solve the CRS into a closed-form upper bound. In order to measure the accuracy gain when incrementing the value of k, we have evaluated the upper bound for different combinations of the input arguments and computed the average. Column **#ev** shows the number of different combinations evaluated for the benchmark. The result of each evaluation is a positive integer value for each cost center identified by the points-to analysis. To measure the accuracy gain, we compare such value for a given k_i with the value (upper bound) obtained for the next k_{i+1} for each cost center. Observe that a cost center obtained in $k = i$ might be split in two (or more) cost centers for $k = i + 1$, and such partition might result in a smaller (more precise) upper bound. $\%_c$ shows the percentage of cost centers for $k = i$ for which the upper bound of its corresponding cost center(s) with $k = i + 1$ is smaller. The gain for a cost center c for $k = i$, for which c' is its corresponding cost center for $k = i + 1$, is obtained as $(1 - UB^{k=i+1}|_{c'} \ / \ UB^{k=i}|_{c}) * 100$ where $UB^{k=i+1}|_{c'}$ is the upper bound obtained with $k = i + 1$ for the cost center c'. $UB^{k=i}|_{c'}$ is the upper bound obtained with $k = i$ for c'. Columns $\%_m$ and $\%_M$ show, respectively, the minimum and the maximum gains, and column $\%_A$ shows the average of the gains obtained for all cost centers that improve their results.

Let us start by discussing benchmarks BBuffer and DistHT. In their main methods, a structure of objects is created from which the different methods are invoked. The same method is often called from different objects and thus replication of the equations is required. In BBuffer, it can be seen that the number of equations increases from 96, with $k = 1$ up to 301 with $k = 4$. As a consequence, analysis time increases from 602ms to 1645ms. We have a similar behaviour in DistHT, it goes from 78 to 385 equations, and the time goes from 1036 ms to 4165ms. In both benchmarks, the accuracy is improved when k grows. In particular, the number of cost centers which improve their precision ranges between 40% and 70% and the actual gain ranges from 13.4% to 33%. In summary, as expected, an increment on the number of cost centers found by the points-to analysis multiplies the number of equations, leading to more precise bounds and requiring larger analysis time.

As regards MailServer and PeerToPeer, the best precision is achieved with $k = 2$, i.e., incrementing the value of k does not lead to further improvements in the cost analysis. In particular, 72% of the cost centers improve their precision and, for the MailServer the gain is on an average 19.1%, while for PeerToPeer it is 45.0%. For both benchmarks, the time taken by the analysis increases with the number of new equations created. In MailServer, we need from 644ms for 93 equations to 1027ms for 159 equations and in the PeerToPeer, from 306 equations in 14,6s to 581 equations in 28s. For BookShop, we obtain the best precision with $k = 3$, achieving an improvement in 40% of the cost centers, and a gain of 18.2% in the upper bounds, which is higher than the improvement obtained from $k = 1$ to $k = 2$, on average 10.5% for 42.9% of the cost centers. Note that, while the efficiency of the analysis significantly degrades when going from $k = 1$ to $k = 2$, the performance is not significantly affected by the increment in the precision from $k = 2$ to $k = 3$. Regarding Chat, we can see that the analysis obtains its best precision with $k = 4$. Despite that, the improvement from $k = 3$ to $k = 4$ is not significant, as only 2.4% of the cost centers improve their precision, and the improvement is not relevant, 0.5%.

All in all, we argue that our experimental evaluation shows that object-sensitive cost analysis is feasible and accurate. As expected, the more precise the points-to analysis is, the more precise the upper-bounds obtained are. According to the experiments, the best value for k is between 2 and 3, but we point out that this is quite dependent on the benchmarks. In our benchmarks, the application of the points-to analysis with $k = 2$ is precise enough to obtain a good balance between precision and performance. Another interesting conclusion is that incrementing the value of k does not degrade the performance when there is no accuracy to be gained. This can be observed in MailServer, BookShop and PeerToPeer as, when the increment in the value of k does not produce new cost centers, the analysis is almost the same as for the previous value of k.

7.3. *Case Study: Trading System*

In this section, we aim at evaluating object sensitive cost analysis on a larger application, namely on the TradingSystem case study. Our objective is to use the analysis results to identify potential bottlenecks related to the high resource consumption in some components (objects) of the distributed system and be able to give some hints on the deployment of the application (i.e., how to allocate objects to machines for the actual deployment). The TradingSystem models a supermarket cash desk line: it includes the processes at a single cash desk (e.g., scanning products using a bar code scanner, paying by cash or by credit card); it also handles bill printing, as well as other administrative tasks. A store consists of an arbitrary number of cash desks. Each of them is connected to the store server, holding store-local product data such as inventory stock, prices, etc. The system is divided into two main parts, the CashDeskInstallation and the CashDeskEnvironment. The CashDeskInstallation contains those classes that are in charge of modeling the hardware behaviour and the CashDeskEnvironment, which models at higher level the behaviour of the system. Furthermore, the TradingSystem includes a class for modeling a bank implementation and another one that models an inventory system. The program has 1350 lines of code. Some minor modifications have been done on the source code of the program in order to handle some loops whose number of iterations could not be bounded, such as loops whose number of iterations depends on one particular keystroke, or loops that terminate when the credit card is read properly. Besides, we have added some class invariants to specify that fields are unchanged at release points (these invariants could be automatically obtained by using [10]).

One interesting aspect of the TradingSystem is that the program points at which the objects that compose the system are created are always placed in object initializations and our points-to analysis identifies all of them with $k = 1$. Thus, no gain is obtained for greater values of k. Points-to analysis identifies 23 cost centers and all of them correspond to a different class which models a different element that composes the system, e.g., a printer, a light display, a bar code scanners or a card reader. The application of the object-insensitive cost analysis takes 689 seconds and the time taken by the object-sensitive analysis with $k = 1$ is 875 seconds. This is an expected time due to the complexity and the size of the application. The number of equations goes from 343 for the object insensitive CRS to 413 equations in the object-sensitive approach. This slight increment is due to the fact that each object created in the program is responsible of modeling a concrete part of the system, and we only have one instance of each object and, in most cases, its methods are invoked only once. In order to see which objects can very overloaded and which objects do not have much computation (and thus can be grouped together and share the processor), we have applied the object-sensitive cost analysis and evaluated the upper bound for concrete values of the input arguments. In this case, we have obtained the percentage of instructions attributed to each cost center with respect to the total number of instructions. According to the results obtained we see three main groups of objects: (1) the objects responsible of creating the system structure, the bank and the inventory, accumulate a low number of instructions, namely 9 different objects that accumulate less than 0.7% of the instructions per object; (2) the objects that belong to the CashDeskEnvironment which accumulate, on average, around 9% of the total number of instructions per object; and, (3) those objects that correspond to the CashDeskInstallation which accumulate a quite significant part of the total number of instructions, because they include the most complex parts of the system. Such information can be useful for determining the number of servers that are needed to deploy the system. As the objects that belong to the first group execute a very low number of instructions, we can suggest that their tasks be executed in only one processor. For the second set of objects, we would recommend that the objects that model the CashDeskEnvironment run in a single server. Such server should have more capacity than the one for the first type of objects. For the objects that interact with the hardware in CashDeskInstallation, we suggest that they have their own processor to avoid contention in this part of the system.

8. RELATED WORK

Our work is closely related to other resource usage analysis frameworks [24, 25]. Most of such frameworks assume a sequential execution model and thus do not deal with the main challenges addressed in this paper. Notable exceptions are [29, 20]. In particular, a live heap space analysis for a concurrent language is proposed in [29]. This analysis is proposed for a simple model of shared memory and besides only considers a particular type of resource (memory) while we use a generic notion of cost. The approach in [20] is completely different to ours, and thus not directly comparable. It is based on the use of *dynamic matrices* for modeling cost analysis of concurrent programs. The use of cost centers has been proposed in the context of profiling, but to our knowledge, its use in the context of static analysis is new.

The termination of multi-threaded programs presented in [15] is based on inferring conditions on the global state which are sufficient to guarantee termination and are similar to our class invariants. Observe that such conditions are only one component within our cost analysis framework, which additionally requires the generation of a new form of recurrence relations and the definition of cost models for the concurrent setting. The particular case of occurrence counting analysis in mobile systems of processes, which in our proposal can be obtained using a particular cost model, has been addressed by several contributions in the literature, although they focus on high-level models, such as the π-calculus and BioAmbients [19, 23].

When considering cumulative cost models, as we do in this paper, asynchronous calls can be handled exactly as synchronous calls without sacrifying precision. This is because, in such cost models, what is important is to approximate the number of times a method is executed (i.e., called), and not how many of them might be running in parallel. In contrast, when considering noncumulative cost models, information on the lifetime of each task is important, since it might directly affect the peak consumption of the corresponding resource. As future work, we plan to integrate in our framework cost models that are noncumulative [8].

There exist other analyses for ABS programs that infer *liveness* properties (namely deadlock freeness), and they are thus complementary to ours. Recent work [22, 21] studies the problem of inferring deadlock freeness, i.e., there is no state in which a non-empty set of tasks cannot progress because all tasks are waiting for the termination of other tasks in the set, or otherwise we show the tasks involved in a potential deadlock set. In this case, the analysis tries to infer dependencies among instructions which may lead to deadlocks. As the goal of this analysis is different from ours, the basic techniques used in deadlock analysis are unrelated to those used in resource and termination analysis. However, both deadlock and resource analyses can benefit from the same underlying analysis. In both cases, *points-to analysis* and *may-happen-in-parallel* are auxiliary analysis that can greatly improve their efficiency. As we have seen, points-to analysis allows us to approximate to which objects a reference variable might be pointing. We can have an object-sensitive deadlock analysis which uses the information inferred by the points-to analysis in a similar way as we do. In the cases of may-happen-in-parallel, the deadlock analysis in [21] shows how it can greatly increase the accuracy of the analysis. For termination and resource consumption, a recent extension of our framework [10] proposes to rely on may-happen-in-parallel relations in order to automatically infer class invariants (like those defined in Section 5.3) which allow us to reason on the values of fields at processor release points.

An alternative approach to *static* cost analysis is the *measurement-based* approach [43], where the program is first executed on a set of input values in order to measure the cost of interest (e.g., execution time) for some code fragments, and then the results are combined to generate an estimation of the overall cost. This measurement can also be used in a probabilistic model to infer properties such average cost or its distribution [36]. The measurement-based approach is particularly useful when the cost of interest depends on external factors other than the program instructions. This is the case, for example, of timing analysis or energy consumption, where the time or energy required for executing an instruction depends on the current state of the underlying machine (e.g., the state of the cache). It is clearly less effective when analyzing a modeling languages like ours where the underlying execution architecture is not know. In such setting we typically concentrate on cost models that depend only on the program instructions and data, independently from the environment on which they will be deployed. In a recent work [11] we have explored the combination of static analysis of ABS with a simulation based approach, where the upper-bounds inferred for the (sequential) functional part of ABS where used to estimate the imperative part of the model by means of simulation, under some higher-level assumptions on the resources available in the underlying deployment component.

9. CONCLUSIONS, CURRENT AND FUTURE WORK

We have presented a novel cost analysis framework for concurrent and distributed programs based on actor-based concurrency. In summary, our main results are: (1) a sound size analysis for concurrent execution which is *field-sensitive*, i.e., it tracks data stored in the heap whenever it is sound to do so. The size analysis can be used in combination with *class invariants* which contain information on the shared memory; (2) an extension of the notion of cost used in sequential programming to the distributed setting by relying on the notion of *cost centers*, which represent the (distributed) components and allow separating their costs; (3) a flow-sensitive object-sensitive points-to analysis for the concurrent objects setting setting; (4) a novel form of *object-sensitive* recurrence relations which relies on information gathered by the object-sensitive points-to analysis in order to generate the cost equations; (5) a prototype implementation of a cost analyzer for programs written in the ABS language.

To develop the analysis, we have considered an object-oriented language based on the notion of concurrent objects which live in a distributed environment with asynchronous communication. The basics of our techniques could be adapted to other concurrent programming languages. In particular, the idea of having equations parametric on the cost centers is of general applicability. The size analysis is tailored for the concurrency primitives of our language, but similar abstractions could be developed for other languages which use monitors, and an analogous abstraction would be directly applicable to other actor-based languages (e.g., Scala or Erlang).

Current work is focused on the automatic inference of class invariants which can be used to know the values of fields at processor release points. The challenge is on being able to infer the resource consumption and prove termination even in cases in which fields involved in the loop conditions are modified by several methods. For instance, consider the following two methods which belong to the same class (elems is a field):

```
Int m() {
    Int i = 0;
    while (i < elems) {
        f = o ! remoteCall();          void inc() {
        await f?;                          elems = elems + 1;
        i = i + 1;                     }
    }
    return i;
}
```

An interleaved execution of them would not allow us to prove termination. Recent work [10] proposes the use of a may-happen-in-parallel analysis to detect whether at the **await** instruction in the body of the loop, we might have an instance of method inc pending to be executed. If this is not the case, we can safely prove termination of the loop in m. Even more, even if there might be an instance of inc in the object queue when the processor is released, we would be able to prove termination, as the value of elems will be incremented once, but it will then remain stable. The general reasoning proposed in [10] is to prove that we have a finite number of instructions which update the field of interest in the queue. If this is the case, the value of the field will eventually not be modified any longer. Thus, the required invariants should establish the boundness of the field values.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.
3. Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. Cost Analysis of Concurrent OO programs. In Hongseok Yang, editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2011.
4. Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
5. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Diana Ramírez-Deantes. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2010.
6. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 368–375. ACM, 2008.
7. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.
8. Elvira Albert, Puri Arenas, Samir Genaim, and Damiano Zanardini. Task-Level Analysis for a Language with Async-Finish Parallelism. *ACM SIGPLAN Notices*, 46(5):21–30, 2011.
9. Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Peter Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 595–598. ACM, 2013.
10. Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2013.
11. Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368, 2011.
12. Elvira Albert, Samir Genaim, and Raúl Gutiérrez. Towards a Transformational Approach to Resource Analysis with Typed-Norms (Extended Abstract). In *23rd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'13)*, pages 85–96, September 2013.

13. Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
14. Annalisa Bossi, Nicoletta Cocco, and Massimo Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In Samson Abramsky and T. S. E. Maibaum, editors, *APSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer, 1991.
15. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving Thread Termination. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 320–330. ACM, 2007.
16. Karl Crary and Stephanie Weirich. Resource Bound Certification. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 184–198. ACM, 2000.
17. Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A Complete Guide to the Future. In Rocco de Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, March 2007.
18. Manuel Fähndrich. Static Verification for Code Contracts. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2010.
19. Jérôme Feret. Occurrence Counting Analysis for the Pi-Calculus. *Electronic Notes in Theoretical Computer Science*, 39(2):1–18, 2001.
20. Gian Luigi Ferrari and Ugo Montanari. Dynamic Matrices and the Cost Analysis of Concurrent Programs. In Vangalur S. Alagar and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, Canada, July 3-7, 1995, Proceedings*, volume 936 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 1995.
21. Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems (FMOODS/FORTE 2013)*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, June 2013.
22. Elena Giachino and Cosimo Laneve. Analysis of Deadlocks in Object Groups. In Roberto Bruni and Jürgen Dingel, editors, *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6722 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2011.
23. Roberta Gori and Francesca Levi. A New Occurrence Counting Analysis for Bioambients. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 381–400. Springer, 2005.
24. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009.
25. Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.
26. E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):35–58, 2007.
27. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.
28. Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and System Modeling*, 6(1):39–58, 2007.
29. Martin Kero, Pawel Pietrzak, and Johan Nordlander. Live Heap Space Bounds for Real-Time Systems. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010.
30. Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In David S. Rosenblum and Gabriele Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.
31. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.
32. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering Methodology*, 14:1–41, 2005.
33. Richard G. Morgan and Stephen A. Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing*, 8(3):201–237, 1998.
34. Benjamin C. Pierce. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer, 1994.

35. Jan Schäfer and Arnd Poetzsch-Heffter. JCobox: Generalizing Active Objects to Concurrent Components. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, Lecture Notes in Computer Science, pages 275–299. Springer, 2010.
36. Sanjit A. Seshia and Alexander Rakhlin. Quantitative Analysis of Systems using Game-Theoretic Learning. *ACM Transactions on Embedded Computing Systems*, 11(S2):55:1–55:27, August 2012.
37. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your Contexts Well: Understanding Object-Sensitivity. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 17–30. ACM, 2011.
38. Fausto Spoto, Fred Mesnard, and Étienne Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.
39. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer, 2008.
40. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Optimization Framework. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, pages 125–135. IBM, 1999.
41. Vasco Thudichum Vasconcelos. *A Process-Calculus Approach to Typed Concurrent Objects*. PhD thesis, Keio University, 1994.
42. Ben Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.
43. Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, volume 17 of *Communications in Computer and Information Science*, pages 430–444. Springer, 2008.

PROOFS (added for reviewers' convenience and to appear in an electronic appendix)

A. SOUNDNESS (PROOF SKETCH FOR THEOREM 1)

In the following, when we talk about $\leadsto$-traces, we refer to traces in which the application of rule (11) in Figure 3 is not considered. This is because such a rule only selects tasks from a queue but does not make any computation.

In order to prove the Theorem, we can reason by induction on the length n of the trace $\mathcal{S}_0 \leadsto^n \mathcal{S}_n$.

Base Case: If the trace is of length 0, i.e., $(n = 0)$ then

$$\mathcal{S}_0 \equiv \{\texttt{ob}(\langle id_0, \epsilon\rangle, \bot, \bot, \langle tv_0, \mathbf{call}(\mathbf{b}, \mathsf{main}(\langle this\rangle, \langle\rangle))\rangle, \{\})\}$$

and Theorem 1 trivially holds.

Inductive Case: Let us consider traces of length $n + 1 > 0$. Assuming that the theorem holds for all $\leadsto$-traces of length $n \geq 0$ (the induction hypothesis), we show that it also holds for traces that consist of $n + 1$ steps. Consider a $\leadsto$-trace of length n, $\mathcal{S}_0 \leadsto \ldots \leadsto \mathcal{S}_n$. By the induction hypothesis, Theorem 1 holds for all $\mathcal{S}_j$, $0 \leq j \leq n$. In what follows, i stands for the program point of the instruction executed in state $\mathcal{S}_{n+1}$. We use ϕ_i, θ_i to refer to the components of the abstract state after instruction i, and $\phi_{pred(i)}, \theta_{pred(i)}$ to refer to the abstract state before instruction i. We will also use *name*(o) to refer to the object name of an object identifier o, as described in Section 4.2. To extend the Theorem to traces of length $n + 1$ we reason for all possible cases in Figure 3. In all cases except Rule (5), Theorem 1 holds for $\mathcal{S}_n$ and $tv(this)$ is not modified in $\mathcal{S}_{n+1}$, then $name(o) \in \phi_i(this)$ holds and therefore case a) of Theorem 1 holds for $\mathcal{S}_{n+1}$. We assume we select non-deterministically one $a \in \mathcal{S}_n$ as follows.

[Rule (1)] $a \equiv \texttt{ob}(o, C, h, \langle tv, x := e \cdot \bar{b}\rangle, \mathcal{Q})$. If x is a reference variable, e may be one of the following:

- $e \equiv y$, where y is a local variable (different from x). Points-to analysis applies row 2 of the transfer function τ shown in Figure 4, updating ϕ_i as follows:

  $$\phi_i = \phi_{pred(i)}[x^l \mapsto \phi_{pred(i)}(y^l)], \forall l \in \phi_{pred(i)}(this)$$

  By the induction hypothesis, $name(tv(y)) \in \phi_{pred(i)}(y^{name(o)})$ holds. At state $\mathcal{S}_{n+1}$ Rule (1) sets $tv(x) = tv(y)$. As state $\mathcal{S}_{n+1}$ does not change $tv(y)$ and τ sets $\phi_i(x^{name(o)}) = \phi_{pred(i)}(y^{name(o)})$, then $name(tv(x)) \in \phi_i(x^{name(o)})$ holds.
- $e \equiv this.f$, where f is a field of class C. This case is similar to the previous case, applied to fields. Points-to analysis applies row 3 of the transfer function τ shown in Fig. 4, updating ϕ_i as follows:

  $$\phi_i = \phi_{pred(i)}[x^l \mapsto \phi_{pred(i)}(l.f)], \forall l \in \phi_{pred(i)}(this)$$

  By the induction hypothesis $name(h(f)) \in \phi_{pred(i)}(name(o).f)$ holds. At state $\mathcal{S}_{n+1}$ Rule (1) sets $tv(x) = h(f)$. As state $\mathcal{S}_{n+1}$ does not change $h(f)$ and τ sets $\phi_i(x^{name(o)}) = \phi_{pred(i)}(name(o).f)$, then $name(tv(x)) \in \phi_i(x^{name(o)})$ holds.
- $e \equiv \mathsf{null}$. In this case Theorem 1 is not applicable.

In all applicable cases, case b) of Theorem 1 holds. Case c) also holds, as $\phi_i(name(o).f)$ does not change for any $f \in \bar{f}_C$.

[Rule (2)] $a \equiv \texttt{ob}(o, C, h, \langle tv, this.f := y \cdot \bar{b}\rangle, \mathcal{Q})$. If f is a reference field, the points-to analysis applies row 5 of the transfer function τ shown in Figure 4, updating ϕ and θ as follows:

$$\phi_i = \phi_{pred(i)}[l.f \mapsto \phi_{pred(i)}(y^l)], \forall l \in \phi_{pred(i)}(this) \quad (1)$$
$$\theta_i = \theta_{pred(i)}[l.f \mapsto \phi_{pred(i)}(y^l) \cup \theta_{pred(i)}(l.f)], \forall l \in \phi_{pred(i)}(this) \quad (2)$$

By the induction hypothesis, $\mathit{name}(tv(y)) \in \phi_{pred(i)}(y^{\mathit{name}(o)})$. At state $\mathcal{S}_{n+1}$ Rule (2) sets $h(o.f) = tv(y)$. As state $\mathcal{S}_{n+1}$ does not change $tv(y)$ and τ sets $\phi_i(\mathit{name}(o).f) = \phi_i(y^{\mathit{name}(o)})$, $\mathit{name}(h(f)) \in \phi_i(\mathit{name}(o).f)$ holds. Therefore, case c) of Theorem 1 holds. Case b) also holds, as $\phi_i(x^{\mathit{name}(o)})$ does not change for any $x \in \mathit{dom}(tv)$.

Observe that, as a result of the application of τ, $\phi_i(l.f) \subseteq \theta_i(l.f), \forall f \in \phi(\mathit{this}), \forall f \in \bar{f}_C$ (1 and 2). Flow-insensitive analysis information stored in θ is propagated along all program points, and it only grows as a result of the application of row 5 of the transfer function. Flow-insensitive information for fields will be used in rows 7 and 8 (Rules (5) and (9) of the semantics, respectively).
[Rule (3)] $a \equiv \mathtt{ob}(o, C, h, \langle tv, x := \mathbf{new}\ D \cdot \bar{b}\rangle, \mathcal{Q})$. As a result of the application of this rule, a fresh object identifier o_1 is created and stored in $tv(x)$ by means of $\mathit{newRef}(i, o)$.
Points-to analysis applies row 1 of the transfer function τ shown in Figure 4, updating ϕ_i as follows:

$$\phi_i = \phi_{pred(i)}[x^l \mapsto \{l \oplus_k i\}], \forall l \in \phi_{pred(i)}(\mathit{this})$$

$\phi_i(x^{\mathit{name}(o)})$ is a set with a single object name which is the abstraction produced by $\mathit{name}(o_1)$. Therefore, case b) of Theorem 1 holds.
[Rule (4)] $a \equiv \mathtt{ob}(o, C, h, \langle tv, \mathbf{call}(\mathbf{b}, p(\mathit{this}, \bar{x}, \bar{y})) \cdot \bar{b}\rangle, \mathcal{Q})$. This rule generates a fresh renaming of program rule p and creates a new mapping in which reference variables are initialized to $\mathbf{null}$. Then the fresh variables $\bar{x}'$ that correspond to formal parameters are assigned the values of the actual parameters in the call, $tv(\bar{x})$.
Row 6 in Table 4 describes calls to blocks in the transfer function τ. The function interp projects the abstractions of actual input parameters stored in ϕ into formal parameters, resulting in a new mapping ϕ'. Therefore, for each $x_j \in \bar{x}$ and $x'_j \in \bar{x}'$, $\phi'_i = \phi_{pred(i)}[(x'_j)^l \mapsto \phi_{pred(i)}(x^l_j)], \forall l \in \phi_{pred(i)}(\mathit{this})$. Since $\mathit{name}(o) \in \phi_{pred(i)}(\mathit{this})$, $\mathit{name}(tv(x'_j)) = \mathit{name}(tv(x_j))$ and $\mathit{name}(tv(x_j)) \in \phi_{pred(i)}(x_j^{\mathit{name}(o)})$ hold, then $\mathit{name}(tv(x'_j)) \in \phi_i((x'_j)^{\mathit{name}(o)})$ holds and also case b) of Theorem 1.
[Rule (5)] $a \equiv \mathtt{ob}(o, C, h, \langle tv, \mathbf{call}(\mathbf{m}, p(\mathit{rec}, \bar{x}, y)) \cdot \bar{b}\rangle, \mathcal{Q})$. This rule corresponds to an asynchronous call to a method on object $o_1 = tv(\mathit{rec})$ that adds a new task to object o_1 with the formal parameters $\bar{x}'$ initialized to the actual parameters $\bar{x}$ in the call, i.e.,

$$\mathtt{ob}(o_1, D, h_1, \_, \{\langle tv_3, b'_1 \cdots b'_n\rangle\} \cup \mathcal{Q}')$$

where D is the class that o_1 belongs to, h_1 is the current mapping of fields to values in o_1 local heap, and tv_3 is a mapping of local variables, initialized to 0 or $\mathbf{null}$, and parameters, that take their values from tv, namely $tv_3(\bar{x}') = tv(\bar{x})$ and $tv_3(\mathit{this}) = tv(\mathit{rec}) = o_1$.
The points-to analysis uses function interp for handling method calls in row 7 of Table 4 as follows:

$$\langle \phi_i, \theta_i\rangle = \mathit{interp}(\langle \phi_{pred(i)}, \theta_{pred(i)}\rangle, \phi_{pred(i)}(\mathit{rec}^l), m(\mathit{rec}, \bar{x}, y)), \forall l \in \phi_{pred(i)}(\mathit{this})$$

It projects the abstractions of rec and actual parameters stored in $\phi_{pred(i)}$ into this and formal parameters, resulting in a new mapping ϕ', which is then used to analyze m. By the induction hypothesis, $\mathit{name}(o) \in \phi_{pred(i)}(\mathit{this})$ and $\mathit{name}(tv(\mathit{rec})) \in \phi_{pred(i)}(\mathit{rec}^{\mathit{name}(o)})$. For each $l \in \phi_{pred(i)}(\mathit{this})$, function interp sets $\phi'(\mathit{this})$ to $\phi_{pred(i)}(\mathit{rec}^l)$. Therefore, there is one l for which $\mathit{name}(o_1) \in \phi'(\mathit{this})$ and case a) of Theorem 1 holds.
Regarding the parameters, for each $l \in \phi_{pred(i)}(\mathit{this})$ and for each $x_j \in \bar{x}$ and $x'_j \in \bar{x}'$,

$$\phi' = \phi_{\perp}[(x'_j)^r \mapsto \phi_{pred(i)}(x^l_j)], \forall r \in \phi'(\mathit{this})$$

Since $\mathit{name}(tv_3(x'_j)) = \mathit{name}(tv(x_j))$ and by the induction hypothesis $\mathit{name}(tv(x_j)) \in \phi_{pred(i)}(x_j^{\mathit{name}(o)})$ holds, then $\mathit{name}(tv_3(x'_j)) \in \phi'((x'_j)^{\mathit{name}(o_1)})$ also holds and thus case b) of Theorem 1 holds.

Fields in ϕ' are updated with the flow-insensitive information in $\theta_{pred(i)}$ as follows:

$$\phi' = \phi'[r.f \mapsto \theta_{pred(i)}(r.f)], \forall r \in \phi'(this)$$

As mentioned in the case of rule (2), flow-insensitive analysis information stored in θ is propagated along the program, and it is updated when a field $r.f$ is assigned a value, adding the new abstract value to $\theta(r.f)$. Therefore, $\phi'(\mathit{name}(o_1).f)$ correctly stores all possible values that field f may be assigned to by other tasks of the same object, and thus case c) of Theorem 1 holds.

Finally, after the analysis of method m function $interp$ stores the analysis results of the returning value of m in the future variable y, i.e.,

$$\phi_i = \phi_{pred(i)}[y^l \mapsto \cup\{\phi''((y')^r) \mid r \in \phi_{pred(i)}(rec^l)\}], \forall l \in \phi_{pred(i)}(this)$$

where ϕ'' is the transfer function at the end of method m. Note that in ϕ'' there are as many replicas of y' as elements in the set $\phi_{pred(i)}(rec^l)$ that are joined to produce a result to assign to y^l. The value stored in y^l will be used when retrieving the result of the method by means of a **get** instruction, as it is detailed in Rule (8) below.

[Rules (6) and **(7)]** $a \equiv \mathtt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q})$. Both rules correspond to the end of a block or a method, respectively. By the induction hypothesis, Theorem 1 holds for $\mathcal{S}_n$. Rules (6) and (7) do not change $this$ nor local variables or fields, therefore Theorem 1 also holds for $\mathcal{S}_{n+1}$. Observe that rule (7) stores the value returned by an asynchronous execution of a method in $\mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}_{n+1}$. This value is used by rule (8) below.

[Rule (8)] $a \equiv \mathtt{ob}(o, C, h, \langle tv, x := y.\mathbf{get} \cdot \bar{b}\rangle, \mathcal{Q})$. This rule corresponds to the execution of a **get** instruction. By the induction hypothesis, Theorem 1 holds for $\mathcal{S}_n$ and previous states. The application of this rule requires the application of rule (7) in a previous step $\mathcal{S}_j$ for some $j \leq n$, in order to set a value for the future variable $\mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}_j$. Since future variables are not removed from the states of a trace, in any trace $k > j$, $\mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}_k$ holds, and in particular $\mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}_n$.

If x is a reference variable and the method whose call is linked to the future variable y returns a reference, the points-to analysis stores the returning value of the method in the local future variable y, i.e., $\phi(y^l), \forall l \in \phi_{pred(i)}(this)$, by means of function $interp$ used in row 7 of Table 4. This value is then used in row 9 of Table 4 for **get**:

$$\phi_i = \phi_{pred(i)}[x^l \mapsto \phi_{pred(i)}(y^l)], \forall l \in \phi_{pred(i)}(this)$$

Since $\mathit{name}(o) \in \phi_{pred(i)}(this)$, $\phi_i(x^{\mathit{name}(o)}) = \phi_{pred(i)}(y^{\mathit{name}(o)})$ holds. As seen in rule (5), the returning value is properly stored in $\phi(y^{\mathit{name}(o)})$. Therefore, $\mathit{name}(tv(x)) \in \phi_i(x^{\mathit{name}(o)})$ holds, and also case b) of Theorem 1. Case c) also holds, as $\phi_i(\mathit{name}(o).f)$ does not change for any $f \in \bar{f}_C$.

[Rules (9) and **(10)]** $a \equiv \mathtt{ob}(o, C, h, \langle tv, \mathbf{await}\ x? \cdot \bar{b}\rangle, \mathcal{Q})$. These rules correspond to the execution of an **await** instruction. First rule considers the case in which the future variable is ready, and the task continues executing. In the case of rule (10), the condition of the await does not hold and the processor is released. Any other task in o may continue executing, possibly changing field values in h.

When an **await** instruction is found, the points-to analysis applies row 8 of Table 4 as follows:

$$\phi_i = \phi_{pred(i)}[l.f \mapsto \phi_{pred(i)}(l.f) \cup \theta_{pred(i)}(l.f)], \forall l \in \phi_{pred(i)}(this)$$

As it has been mentioned in the case of rule (2), flow-insensitive analysis information stored in θ is propagated along the program, and it is updated when a field $l.f$ is assigned a value, adding the new abstract value to $\theta(l.f)$.

Since $\mathit{name}(o) \in \phi_{pred(i)}(this)$, $\phi_i(\mathit{name}(o).f)$ correctly stores all possible values that field f may be assigned to by other tasks of the same object, and thus case c) of Theorem 1 holds. Case b) also holds, as $\phi_i(x^{\mathit{name}(o)})$ does not change for any $x \in dom(tv)$.

B. SOUNDNESS (PROOF SKETCH FOR THEOREM 2)

In the following, when we talk about $\rightsquigarrow$-traces, we refer to traces in which the application of rule (11) in Figure 3 is not considered. This is because such a rule only selects tasks from a queue but does not make any computation. Note that, each time we give a $\rightsquigarrow^{\alpha}$-step, of the form $\mathcal{A} \circ \phi \rightsquigarrow^{\alpha} \mathcal{A}' \circ \phi'$, it holds that $\phi' = \phi \wedge \varphi$. This means that $\phi' \models \phi$ trivially. Hence it is trivial that in any α-trace of the form $\mathcal{A}_0 \circ \phi_0 \rightsquigarrow^{\alpha} \ldots \rightsquigarrow^{\alpha} \mathcal{A}_n \circ \phi_n$ it holds that $\phi_n \models \phi_i$, for all $1 \leq i \leq n$. The rest of conditions of the Theorem are proved by the induction on the length n of the trace $\mathcal{S}_0 \rightsquigarrow^{n} \mathcal{S}_n$.

Base Case: If the trace is of length 0, i.e., $n = 0$, then

$$\mathcal{S}_0 \equiv \{\texttt{ob}(\texttt{main}, \bot, \bot, \langle tv_0, \textbf{call}(\mathbf{b}, \mathsf{main}(this, \langle\rangle, \langle\rangle))\rangle, \{\})\}$$

and we can take ρ_0 as the identity mapping, and ρ_1 and $\mathtt{ABST}(\textbf{call}(\mathbf{b}, \mathsf{main}(this, \langle\rangle, \langle\rangle)), \rho_0) = \langle \textbf{call}(\mathbf{b}, \mathsf{main}(this, \langle\rangle, \langle\rangle)), \rho_1\rangle$.

Inductive Case: Let us consider traces of length $n + 1 > 0$. Assuming that the theorem holds for all $\rightsquigarrow$-traces of length $n \geq 0$ (the induction hypothesis), we show that it also holds for traces that consist of $n + 1$ steps. Consider a $\rightsquigarrow$-trace of length n:

$$\mathcal{S}_0 \equiv \{\texttt{ob}(\texttt{main}, \bot, \bot, \langle tv_0, \textbf{call}(\mathbf{b}, \mathsf{main}(this, \langle\rangle, \langle\rangle))\rangle, \{\})\} \rightsquigarrow \mathcal{S}_n$$

By the induction hypothesis, there exists an abstract trace:

$$\mathcal{A}_0 \equiv \langle \textbf{call}(\mathbf{b}, \mathsf{main}(this, \langle\rangle, \langle\rangle)), \rho \cdot \rho'\rangle \circ true \quad \rightsquigarrow^{n}_{\alpha} \mathcal{A}_n \circ \phi_n$$

such that $S_i \approx \mathcal{A}_i \circ \phi_i$, $1 \leq i \leq n$. Let us analyze how the theorem extends to all possible $\rightsquigarrow$-traces of length $n + 1$ generated from the above concrete and abstract traces. We reason for all possible cases in Figure 3, by assuming we select non-deterministically one $a \in S_n$ as follows:

[**Rule (1)**] $a \equiv \texttt{ob}(o, C, h, \langle tv, x := e \cdot \bar{b}\rangle, \mathcal{Q})$. Then $\mathcal{S}_{n+1}$ is equals to $\mathcal{S}_n$ by replacing object a by the new one $\texttt{ob}(o, C, h, \langle tv[x \mapsto v], \bar{b}\rangle, \mathcal{Q})$. Now by the induction hypothesis, it holds that $a^{\alpha} \equiv \langle \varphi \cdot \bar{b}^{\alpha}, \rho_1 \cdot \rho_2 \cdot \bar{\rho}\rangle \in \mathcal{A}_n \circ \phi_n$. Let us now analyze all possible forms of expression e:

- $e = y$. Then $eval_e(e, h, tv) = tv(y)$ and $\varphi = \rho_2(x) = \rho_1(y)$. Let us σ be the assignment satisfying point 2 of Def. 8. Let us define σ' as σ together with $\sigma(\rho_2(x)) = v$. Then by the induction hypothesis ϕ_n is satisfiable, and thus, since $\rho_2(x)$ is a fresh variable, then $\phi_{n+1} \equiv \phi_n \wedge \rho_2(x) = \rho_1(y)$ is also satisfiable and we can apply the $\rightsquigarrow_{\alpha}$ rule $(1)_{\alpha}$. Consider σ' defined as σ together with $\sigma'(\rho_2(x)) = v$. The only points to be proved are $tv[x \mapsto v](x) = v = \sigma'(\rho_2(x))$ and $\sigma' \models \phi_{n+1}$. The first point is trivial. The second one can be deduced by the induction hypothesis together with the following $v = tv(y) = \sigma(\rho_2(y)) = \sigma(\rho_1(y)) = \sigma'(\rho_1(y))$.
- $e = this.f$, $eval_e(e, h, tv) = h(f) = v$ and $\texttt{ob}(o, C, h, \langle tv[x \mapsto v], \bar{b}\rangle, \mathcal{Q}) \in \mathcal{S}_{n+1}$. By the induction hypothesis it holds that $\langle \rho_2(x) = \rho_1(f), \rho_1 \cdot \rho_2 \cdot \bar{\rho}\rangle \in \mathcal{A}_n \circ \phi_n$. For the same reason as in the previous case $\phi_{n+1} \equiv \phi_n \wedge \rho_2(x) = \rho_1(f)$ is satisfiable, and we can apply rule $(1)_{\alpha}$ to compute $\mathcal{A}_{n+1} \circ \phi_{n+1}$ satisfying that $\langle \bar{b}^{\alpha}, \rho_2 \cdot \bar{\rho}\rangle \in \mathcal{A}_{n+1} \circ \phi_{n+1}$. Let us define σ' as σ extended with $\sigma'(\rho_2(x)) = v$ and $\sigma'(\rho_2(f)) = v$. Then it is enough to note first that $\sigma'(\rho_2(x)) = v = tv[x \mapsto v](x)$. And second, since by the induction hypothesis $\sigma(\rho_1(f)) = h(f) = v$, then $\sigma'(\rho_2(f)) = \sigma(\rho_1(f))$.
- The remaining cases can be reasoned similarly as ones above.

[**Rule (2)**] Then $\texttt{ob}(o, C, h, \langle tv, this.f := y \cdot \bar{b}\rangle, \mathcal{Q}) \in \mathcal{S}_n$ and $\texttt{ob}(o, C, h[f \mapsto v], \langle tv, \bar{b}\rangle, \mathcal{Q}) \in \mathcal{S}_{n+1}$. After applying the induction hypothesis we compute $\langle \rho_2(f) = \rho_1(y) \cdot \bar{b}^{\alpha}, \rho_1 \cdot \rho_2 \cdot \bar{\rho}\rangle \circ \phi_n \in \mathcal{A}_n$ and the conditions of the Theorem hold. Concretely there exists σ satisfying ϕ_n which trivially satisfies also $\phi_{n+1} \equiv \phi_n \wedge \rho_2(f) = \rho_1(y)$. Then we can apply rule $(3)_{\alpha}$ to compute $\langle \bar{b}^{\alpha}, \rho_2 \cdot \bar{\rho}\rangle \circ \phi_{n+1} \in \mathcal{A}_{n+1}$. Let us consider σ' defined as σ and extended with

$\sigma'(\rho_2(f)) = v$. In order to prove this case it is enough to prove that $\sigma'(\rho_2(f)) = h(f)$, what trivially holds because of the definition of σ'.

[Rule (3)] $a \equiv \texttt{ob}(o, C, h, \langle tv, x := \textbf{new}\, D \cdot \bar{b}\rangle, \mathcal{Q})$ and $\langle o, C.h, \langle tv[x \mapsto o_1], \bar{b}\rangle, \mathcal{Q}\rangle$ $\langle o_1, D, h_1, \epsilon, \emptyset\rangle$ belongs to $\mathcal{S}_{n+1}$. By the induction hypothesis $\langle \rho_2(x) = 1 \cdot \bar{b}^\alpha, \rho_1 \cdot \rho_2 \cdot \bar{\rho}\rangle \in \mathcal{A}_n \circ \phi_n$, and Def. 8 holds. Note that since $\rho_2(x)$ is a fresh variable and ϕ_n is satisfiable (by the induction hypothesis), then $\phi_{n+1} \equiv \phi_n \wedge \rho_2(x) = 1$ is also satisfiable. Let σ be an assignment ensuring $\mathcal{S}_n \approx \mathcal{A}_n \circ \phi_n$. Let us consider σ' defined as σ together with $\sigma'(\rho_2(x)) = 1$. Then $\mathcal{S}_{n+1} \approx \mathcal{A}_{n+1} \circ \phi_{n+1}$ follows from the induction hypothesis and by the definition of σ'. Note that since $\mathit{tasks}(\texttt{ob}(o_1, D, h_1, \epsilon, \emptyset)) = \emptyset$, this object has not to be considered.

[Rule (4)] For this case it holds that $\texttt{ob}(o, C, h, \langle tv, \texttt{call}(\textbf{b}, p(\mathit{this}, \bar{x}, \bar{y})) \cdot \bar{b}\rangle, \mathcal{Q}) \in \mathcal{S}_n$ and $\texttt{ob}(o, C, h, \langle tv \cup tv_2, b'_1 \cdots b'_n \cdot \bar{b}\rangle, \mathcal{Q}) \in \mathcal{S}_{n+1}$ and $r \equiv p(\mathit{this}', \bar{x}', \bar{y}) \leftarrow g',\ b'_1, \ldots, b'_n \ll P,\ tv_1 = \mathit{newEnv}(r - \{\bar{y}\}),\ tv_2 = tv_1[\mathit{this}' \mapsto o, \bar{x}' \mapsto tv(\bar{x})],\ \mathit{eval}_{gd}(g', tv_2) = \mathit{true}$. By the induction hypothesis, we can build the abstract trace $\mathcal{A}_0 \leadsto^n_\alpha \mathcal{A}_{n+1}$, where $\langle \texttt{call}(\textbf{b}, p(\mathit{this}_1, \bar{x}_1, \bar{y}_1)) \cdot \bar{b}^\alpha, \rho_0 \cdot \rho_1 \cdot \bar{\rho}\rangle \in \mathcal{A}_n \circ \phi_n$. and the conditions of the Theorem hold. Concretely, $\rho_0(\mathit{this}) = \mathit{this}_1$, $\rho_0(\bar{x}) = \bar{x}_1$ and $\rho_1(\bar{y}) = \bar{y}_1$. Furthermore, if σ is the assignment satisfying the Theorem, then $\sigma(\rho_0(\mathit{this})) = tv(\mathit{this}) = tv_2(\mathit{this}')$ and $\sigma(\rho_0(\bar{x})) = tv(\bar{x}) = tv_2(\bar{x})$. Let us select r^α as $p(\mathit{this}_1, \bar{x}_1, \bar{y}_1) \rightarrow g_1, b^\alpha_1, \ldots, b^\alpha_n \circ \rho_0 \cdots \rho_1$. Suppose now that $g' \equiv z' \diamond w'$ and $g' \equiv z'' \diamond w''$. Then since $\sigma(\rho_0(z'')) = tv_2(z')$ and $\sigma(\rho_0(w'')) = tv_q(w')$ and $\mathit{eval}_{gd}(z' \diamond w', tv_2)$, then $\sigma \models \rho_0(z'') \diamond \rho_0(w'')$. Hence $\sigma \models \phi_{n+1}$, where $\phi_{n+1} \equiv \phi_n \wedge z'' \diamond w''$ and we can apply rule $(2)_\alpha$ to get $\langle b^\alpha_1 \cdots b^\alpha_n \cdot \bar{b}^\alpha, \rho_0 \cdots \rho_1 \cdot \rho_1 \cdots \bar{\rho}\rangle \circ \phi_{n+1}$. The only point to prove now is that $\mathtt{ABST}(b'_i, \rho_i) = \langle b^\alpha_i, \rho_{i+1}\rangle$. The result trivially holds (by the induction hypothesis) for $\bar{x}, \bar{y}$ and this. For the rest of variables which are fresh, it is enough to select the corresponding renamings satisfying the condition.

[Rule (5)] Then $\texttt{ob}(o, C, h, \langle tv, \texttt{call}(\textbf{m}, p(rec, \bar{x}, y)) \cdot \bar{b}\rangle, \mathcal{Q}), \texttt{ob}(o_1, D, h_1, \langle tv_1, \bar{b}_1\rangle, \mathcal{Q}') \in \mathcal{S}_n$, where $r \equiv p(\mathit{this}', \bar{x}', y') \leftarrow b'_1, \ldots, b'_n \ll P, tv(rec) = o_1$, $\texttt{fn} = \mathit{newFut}()$, $tv_2 = \mathit{newEnv}(r)$, $tv_3 = tv_2[\mathit{this}' \mapsto o_1, \bar{x}' \mapsto tv(\bar{x}), \texttt{ret} \mapsto (y', \texttt{fn})]$. The application of rule (5) generates $\texttt{ob}(o, C, h, \langle tv[y \mapsto \texttt{fn}], \bar{b}\rangle, \mathcal{Q})$, $\langle o_1, D, h_1, \langle tv_1, \bar{b}_1\rangle, \{\langle tv_3, b'_1 \cdots b'_n\rangle\} \cup \mathcal{Q}'\rangle$, $\texttt{fut}(\texttt{fn}, \bot) \in \mathcal{S}_{n+1}$. By the induction hypothesis, it holds that $\langle \texttt{call}(\textbf{m}, p(rec_1,\ \bar{x}_1, y_1)), \cdot \bar{b}^\alpha,\ \ \rho_1 \cdot \rho_2 \circ \bar{\rho}\rangle, \langle \bar{b}^\alpha_1, \bar{\rho}_1\rangle \in \mathcal{A}_n \circ \phi_n$ where $\rho_1(rec) = \mathit{rec}_1$, $\rho_1(\bar{x}) = \bar{x}_1$, $\rho_2(y) = y_1$ and the conditions of the Theorem holds. Let us select the abstract rule $r^\alpha \equiv p(rec_1, \bar{x}_1, y_1) \rightarrow b^\alpha_1, \ldots, b^\alpha_n \circ \rho_1 \cdots \rho_2$, coming from abstracting rule r. If we apply rule $(2)_\alpha$, then $\langle \bar{b}^\alpha,\ \rho_2 \circ \bar{\rho}\rangle, \langle b^\alpha_1, \ldots, b^\alpha_n \cdot \bar{b}^\alpha_1, \rho_1 \cdots \rho_2 \cdot \bar{\rho}_1\rangle \in \mathcal{A}_n \circ \phi_n$. Let σ be the assignment satisfying $\mathcal{S}_n \approx \mathcal{A}_n \circ \phi_n$. The important points to prove are $\sigma(\rho_1(\bar{x}_1)) = tv_3(\bar{x}')$, $\sigma(\rho_1(rec)) = tv_3(\mathit{this}')$ and $\sigma(\rho_1(y) = tv_3(y)$. But by the induction hypothesis it holds that $\sigma(\rho_1(\bar{x}_1)) = tv(\bar{x}) = tv_3(\bar{x}')$, $\sigma(\rho_1(rec)) = tv(rec) = tv_3(\mathit{this}')$ and $\sigma(\rho_1(y)) = tv(y) = tv_3(y')$.

[Rule (6)] In this case the result trivially holds by the induction hypothesis and the application of rule $(6)_\alpha$.

[Rule (7)] In this case it holds that $\texttt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q}), \texttt{fut}(\texttt{fn}, \bot) \in \mathcal{S}_n$, $\texttt{ret} \in \mathit{dom}(tv), (y,\ \texttt{fn}) = tv(\texttt{ret}), v = tv(y)$ and $\texttt{ob}(o, C, h, \epsilon, \mathcal{Q}), \texttt{fut}(\texttt{fn}, v) \in \mathcal{S}_{n+1}$. The application of rule (7) comes from the complete execution of some method m. Let us select the state S_k in which the call to m is selected to be evaluated. Then we have the following situation: $\texttt{ob}(o_1, D, h_1, \langle tv_1, \texttt{call}(\textbf{m}, m(rec, \_, y) \cdot \bar{b}\rangle, \mathcal{Q}_1), \texttt{ob}(o, C, h_2, \langle tv_2, \bar{b}_2\rangle, \mathcal{Q}_2) \in \mathcal{S}_k$ and $\texttt{ob}(o_1, D, h_3, \langle tv_3, \bar{b}\rangle, \mathcal{Q}_3), \texttt{ob}(o, C, h_2, \langle tv_2, \bar{b}_2\rangle, \mathcal{Q}_3 \cup \{\langle tv_4, \mathit{body}(m)\rangle\}) \in \mathcal{S}_{k_{k+1}}$, where $tv_1(rec) = \mathit{0}$, $tv_3(y) = \texttt{fn}$, $tv_4(\texttt{ret}) = (\texttt{fn}, y')$, $\mathit{body}(m)$ is the renamed rule selected to give the $\leadsto$-step, where y' is the corresponding renaming of y. Now let us consider the sub-trace $\mathcal{S}_{k+1} \leadsto \mathcal{S}_n$, where the task $\langle tv_4, \mathit{body}(m)\rangle$ is now completely resolved. Then $\texttt{ob}(o_1, D, h_6, \langle tv_6, \bar{b}_6\rangle, \mathcal{Q}_6) \in \mathcal{S}_n$ and $tv(\texttt{ret}) = tv_4(\texttt{ret}) = (\texttt{fn}, y')$ and $tv_6(y) = tv_3(y) = \texttt{fn}$.

By the induction hypothesis we can build the α-trace $\mathcal{A}_0 \leadsto^n_\alpha \mathcal{A}_n$ satisfying the conditions of the Theorem. Concretely $\langle \epsilon, \rho \rangle \in \mathcal{A}_n \circ \phi_n$, where ϕ_n is satisfiable. Rule $(6)_\alpha$ can be applied on $\mathcal{A}_n$ to get $\mathcal{A}_{n+1} = \langle \epsilon, \rho \rangle$. The only point to prove is that since $y \in dom(tv_6)$, $tv_6(y) = \mathtt{fn}$ and $(\mathtt{fn}, v) \in \mathcal{S}_{n+1}$, then $\sigma(\rho(y)) = v$, where σ is the assignment satisfying the conditions of the Theorem on $\mathcal{S}_n$ and $\mathcal{A}_n$. However, since $tv(y') = v$, then by the induction hypothesis $\sigma(\rho(y')) = v$. By construction of the α-trace, it holds $\rho(y') = \rho(y)$.

[Rule (8)] In this case $\mathtt{ob}(o, C, h, \langle tv, x := y.\mathbf{get} \cdot \bar{b} \rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}_n$, where $\mathtt{fn} = tv(y)$, $v \neq \bot$. By the induction hypothesis, $\langle \rho_2(x) = \rho_1(y) \cdot \bar{b}^\alpha, \rho_1 \cdot \rho_2 \cdot \bar{\rho} \rangle \in \mathcal{A}_n \circ \phi_n$, where ϕ_n is satisfiable, and there exists σ such that $\sigma \models \phi_n$ and $\sigma(\rho_1(y)) = v$. Now we apply rule (8) on $\mathcal{S}_n$, what transforms $\mathcal{S}_n$ as $\mathtt{ob}(o, C, h, \langle tv[x \mapsto v], \bar{b} \rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v) \in \mathcal{S}_{n+1}$. Since ϕ_n is satisfiable and $\rho_2(x)$ is a fresh variable, then $\phi_{n+1} \equiv \phi_n \wedge \rho_2(x) = \rho_1(y)$ is satisfiable and we can apply rule $(1)_\alpha$ to compute $\langle b^\alpha, \rho_2 \cdot \bar{\rho} \rangle \in \mathcal{A}_{n+1}$. Let us choose an assignment σ' defined as σ but extended with $\sigma'(\rho_2(x)) = v$. This assignment satisfies that $\sigma' \models \phi_{n+1}$. Furthermore $\sigma'(\rho_2(x)) = tv[x \mapsto v](x) = v$. Thus the result holds.

[Rules (9) and (10)] In both cases the proof follows immediately from the induction hypothesis by applying respectively, rules $(4)_\alpha$ and $(5)_\alpha$. Note that for rule (10) we have that $tasks(\mathcal{S}_n) = tasks(\mathcal{S}_{n+1})$ since the task is only introduced in the queue of the current object.

C. SOUNDNESS (PROOF SKETCH FOR THEOREM 3)

We sketch the main ideas of the proof for the object-insensitive analysis, and then we comment on the straightforward changes required to handle the object-sensitive case. The proof sketch consists of two parts:

- In the first one, we instrument the abstract states in the abstract operational semantics of Figure 8 with a cost component that measures the cost of abstract executions; and show that each concrete trace has a corresponding abstract one with the same cost.
- Then, in a second part, we show that the cost relations generated from the abstract program indeed approximate the resource consumption behaviour of the abstract program.

Figure 10 depicts an abstract operational semantics derived from the one of Figure 8 by instrumenting the abstract states with a component that accumulates cost. Namely, an abstract state now has the form $\mathcal{A} \circ \phi \circ e$ where e is the amount of resources consumed so far. The instrumentation is straightforward: when executing b^α we simply accumulate the cost $\mathcal{M}(b^\alpha)$ that, by abusing of notation, we assume to be equivalent to $\mathcal{M}(b)$, i.e., to the cost of the original instruction from which b^α originate. Given an abstract trace $\mathcal{T}^\alpha \equiv \mathcal{A}_0 \circ true \circ 0 \leadsto^n_\alpha \mathcal{A}_n \circ \phi_n \circ e_n$, its cost is defined as $\mathcal{M}(\mathcal{T}^\alpha) = e_n$. Note that this instrumentation has no effect on the abstract executions, i.e., we still have the same abstract traces as those generated using the abstract semantics of Figure 8, and, moreover, Theorem 2 holds for the instrumented abstract semantics of Figure 10.

Now let $\mathcal{S}_0$ be an initial state, and let $\mathcal{T} \equiv \mathcal{S}_0 \leadsto^n \mathcal{S}_n$ be a concrete trace that starts from $\mathcal{S}_0$. Recall that the cost of $\mathcal{T}$, denoted $\mathcal{M}(\mathcal{T})$, is defined as the sum of all $\mathcal{M}(b)$ for each instruction b used in an execution step of $\mathcal{T}$. From Theorem 2, it immediately follows that there is a corresponding abstract trace $\mathcal{T}^\alpha \equiv \mathcal{A}_0 \circ true \circ \leadsto^n_\alpha \mathcal{A}_n \circ \phi_n \circ e_n$ such that $\mathcal{M}(\mathcal{T}^\alpha) = e_n = \mathcal{M}(\mathcal{T})$. This is because according to the proof in Appendix B, the instructions of $\mathcal{T}$ and $\mathcal{T}^\alpha$ coincide, i.e., whenever we make a concrete step that uses an instruction b, we can also make an abstract step that uses b^α. This means that any concrete trace $\mathcal{T}$, has a corresponding abstract one with the same cost.

Next we briefly explain why the cost relations generated from the abstract rules approximate the resource consumption of the abstract program, and thus the resource consumption of the original program. We do this by starting from the abstract program and the abstract semantics of Figure 10, and then modify them several times until we obtain the corresponding cost relations and the corresponding semantics [4].

$$(1)_\alpha \; \frac{p(\mathit{this},\bar{x},\bar{y}) \leftarrow g^\alpha, b_1^\alpha,\ldots,b_n^\alpha \circ \rho_0 \cdots \rho_{n+1} \ll P^\alpha, g^\alpha \wedge \phi \not\models \mathit{false}, e' = \mathcal{M}(p(\mathit{this},\bar{x},\bar{y})) + e}{\{\langle \mathtt{call}(\mathtt{b}, p(\mathit{this},\bar{x},\bar{y})) \cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e \rightsquigarrow_\alpha \{\langle b_1^\alpha \cdots b_n^\alpha \cdot \bar{b}^\alpha, \rho_1 \cdots \rho_{n+1}\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \wedge g^\alpha \circ e'}$$

$$(2)_\alpha \; \frac{p(\mathit{rec},\bar{x},y) \leftarrow b_1^\alpha,\ldots,b_n^\alpha \circ \rho_1 \cdots \rho_{n+1} \ll P^\alpha, e' = \mathcal{M}(p(\mathit{rec},\bar{x},\bar{y})) + e}{\{\langle \mathtt{call}(\mathtt{m}, p(\mathit{rec},\bar{x},y)) \cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e \rightsquigarrow_\alpha \{\langle \bar{b}^\alpha, \bar{\rho}\rangle, \langle b_1^\alpha \cdots b_n^\alpha, \rho_1 \cdots \rho_{n+1}\rangle|\mathcal{A}\} \circ \phi \circ e'}$$

$$(3)_\alpha \; \frac{\varphi \wedge \phi \not\models \mathit{false}, e' = \mathcal{M}(\varphi) + e}{\{\langle \varphi \cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e \rightsquigarrow_\alpha \{\langle \bar{b}^\alpha, \bar{\rho}\rangle|\mathcal{A}\} \circ \phi \wedge \varphi \circ e'}$$

$$(4)_\alpha \; \frac{e' = \mathcal{M}(\bot) + e}{\{\langle \bot \cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e \rightsquigarrow_\alpha \{\langle \bar{b}^\alpha, \bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e'}$$

$$(5)_\alpha \; \frac{}{\{\langle \bot \cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e \rightsquigarrow_\alpha \{\langle \bot \cdot \bar{b}^\alpha, \rho\cdot\bar{\rho}\rangle|\mathcal{A}\} \circ \phi \circ e}$$

$$(6)_\alpha \; \frac{}{\{\langle \epsilon, \rho\rangle|\mathcal{A}\} \circ \phi \circ e \rightsquigarrow_\alpha \{\epsilon|\mathcal{A}\} \circ \phi \circ e}$$

Figure 10. Semantics of Abstract Programs with Cost Annotations

In the first step, we consider a program that is obtained from the abstract program by removing all output variables, we refer to this program as output-free program. Clearly, any trace obtained using the abstract program has a corresponding trace that is obtained using the output-free program with the same resource consumption. This is true since the only difference is that in each step we might add less constraints to the store (we do not add those that match the formal and actual output parameters).

In the second step, we change the abstract semantics such that instead of accumulating the resource consumption of each execution step, it accumulates the resource consumption of all abstract instructions immediately when they are added to the abstract state in rules $(1)_\alpha$ and $(2)_\alpha$. This change amounts to: (i) changing rules $(3)_\alpha$ and $(4)_\alpha$ such that they do not accumulate any cost, and (ii) changing rules $(1)_\alpha$ and $(2)_\alpha$ to accumulate also $c = \mathcal{M}(b_1^\alpha) + \cdots + \mathcal{M}(b_n^\alpha)$. Clearly, this change only anticipates the consumption of resources, and thus for any abstract trace that is obtained using the output-free program and the abstract semantics of Figure 10, we can generate a corresponding abstract trace using the same program and the modified abstract semantics such that it consumes at least the same amount of resources.

In the third step, we eliminate Rules $(3)_\alpha$-$(6)_\alpha$ from the abstract semantics and we modify rules $(1)_\alpha$ and $(2)_\alpha$ such that (i) they add all constraints that appear in the body of the selected rules (let us call them $\varphi = \varphi_1 \wedge \varphi_k$) to the store, and the rest, which are calls, are added as usual to the corresponding task. It is still guaranteed that using this abstract semantics we can reproduce the resource consumption of any trace generated in the above step. This is because the constraints in the body are obtained by applying a single static assignment transformation, thus for any $i > j$ the constraint φ_i does not restrict the values of the variables in φ_j.

Now let us consider an equation $p(\bar{x}) = c + \Sigma q_i(\bar{w}_i), \varphi$ in the cost relation. Here c and φ are the total resource consumption and the constraints of a given rule respectively (as above). It is easy to see that this equation is just a denotational form of the resource consumption as developed in the third step above. Thus, any upper-bound of the cost relation is also an upper bound in the resource consumption of the corresponding abstract traces.

The correctness for the object-sensitive case is straightforward given the soundness of the points-to analysis. The above proof can be adapted to the object-sensitive case by: (i) modifying the abstract program such that it includes corresponding points-to annotations; and (ii) change the abstract semantics in order to accumulate expressions of the form $c(o) * \mathcal{M}(b)$. The correctness of the points-to analysis guarantees that if in the concrete setting we accumulate $\mathcal{M}(b)$ when executing within object o', then in the abstract setting we accumulate $c(o) * \mathcal{M}(b)$ where o is the approximation of the o' inferred by the points-to analysis. Finally, cloning the equation as done in Definition 11 just makes the points-to information explicit in the rules names.