Higher-Order Symb Comput (2010) 23:29–86
DOI 10.1007/s10990-010-9063-7

Magic-sets for localised analysis of Java bytecode

Fausto Spoto · Étienne Payet

Published online: 20 August 2010

Abstract Static analyses based on denotational semantics can naturally model functional behaviours of the code in a compositional and completely context and flow sensitive way. But they only model the *functional i.e.,* input/output behaviour of a program P, not enough if one needs P's *internal* behaviours *i.e.,* from the input to some internal program points. This is, however, a frequent requirement for a useful static analysis. In this paper, we overcome this limitation, for the case of mono-threaded Java bytecode, with a technique used up to now for logic programs only. Namely, we define a program transformation that adds new *magic* blocks of code to the program P, whose functional behaviours are the internal behaviours of P. We prove the transformation correct *w.r.t.* an operational semantics and define an equivalent denotational semantics, devised for abstract interpretation, whose denotations for the magic blocks are hence the internal behaviours of P. We implement our transformation and instantiate it with abstract domains modelling *sharing* of two variables, *non-cyclicity* of variables, *nullness* of variables, *class initialisation* information and *size* of the values bound to program variables. We get a static analyser for full mono-threaded Java bytecode that is faster and scales better than another operational pair-sharing analyser. It has the same speed but is more precise than a constraint-based nullness analyser. It makes a polyhedral size analysis of Java bytecode scale up to 1300 methods in a couple of minutes and a zone-based size analysis scale to still larger applications.

Keywords Magic-sets · Abstract interpretation · Static analysis · Denotational semantics

1 Introduction

Static analysis determines at compile-time properties about the run-time behaviour of computer programs. It is used for optimising their compilation [1], deriving loop invariants and

F. Spoto (✉)
Dipartimento di Informatica, Università di Verona, Verona, Italy
e-mail: fausto.spoto@univr.it

É. Payet
IREMIA, Université de la Réunion, Réunion, France

for program verification. Correctness is usually mandatory for static analysis and proved *w.r.t.* a *reference semantics* of the analysed language. Abstract interpretation [15] shows here its strength since it derives static analyses *from the semantics* itself, so that they are by construction correct or even optimal. The derived analyses inherit semantical features such as compositionality, context and flow sensitivity and can only model program properties that can be formalised in terms of the reference semantics. For Java bytecode, possibly downloaded from insecure networks in a machine-independent, non-optimised format, the source code is not available so that a direct analysis of the bytecode is desirable. Static analysis for Java bytecode is however more difficult than for Java, since the former uses an operand stack of temporary values and its code is not explicitly structured. Moreover, analyses that perform well on Java might be quite imprecise for Java bytecode, as formally discussed in [28]. Nevertheless, the analysis of Java bytecode is considered an important topic at the moment, since it is often used in contexts were verification of security properties is desirable [9, 11, 22, 36, 48].

Denotational static analysis Given a piece of code c, *denotational semantics* provides a *denotation i.e.,* a function from the input state provided to c (the computational state before c is executed) to the resulting output state (the computational state after c has been executed), in the style of the denotation of commands at page 58 of [49], that we copy as our Definition 6. Denotational semantics has some good points that speak in favor of its use for static analysis instead, for instance, of an operational semantics:

- if method m (constructor, function, procedure…) is called at program points $p_1, \ldots, p_n$, denotational analyses compute m's denotation *only once* and then *extend* it at each p_i. Hence they can be very fast for analysing complex software where n is often large. Analyses based on operational semantics process instead m from scratch *for every* p_i. *Memoisation*, which takes note of the input states for which m has been already analysed and caches the results, is a partial solution to this problem, since each p_i often calls m with *different* input states;
- denotations *i.e.,* functions from input to output, can be represented as Boolean functions, namely, logical implications from the properties of the input to the properties of the output. Boolean functions have an efficient implementation as binary decision diagrams [10]. Hence there is a potentially very efficient implementation for many denotational static analyses, which is not always the case for operational static analyses;
- denotational semantics is naturally context and flow sensitive *i.e.,* it can distinguish different invocations of the same method from different program points and can distinguish the values of the same variable at different program points. This contributes to the precision of the analyses;
- denotational semantics is *compositional i.e.,* the denotation of a piece of code is computed bottom-up from those of its subcomponents (commands or expressions). The derived static analyses are hence compositional, an invaluable simplification when one formalises, implements and debugs them;
- denotational semantics does not use program counters, nor activation stacks nor return points from calls. Hence it is simpler to abstract than an operational semantics. Note that the operand stack of the Java Virtual Machine is still modelled by a denotational semantics;
- denotational semantics models naturally properties of the functional behaviour of the code, such as information flows [36] or relational information [26, 27, 42]. Operational semantics is very awkward there.

```
public class List {
    private Element head;     private List tail;

    public List(Element head, List tail) {
        this.head = head;    this.tail = tail; }

    public List clone() {
        if (tail == null) return new List(head, null);
        else return new List(head, tail.clone()); }

    public List deepClone() {
        if (tail == null) return new List(head.copy(), null);
        else return new List(head.copy(), tail.deepClone()); }

    public abstract static class Element {
        public abstract Element copy(); }
}

public class Record extends List.Element {
    private int amount;

    public Record(int amount) {
        this.amount = amount; }

    public void update(int amount) {
        this.amount = amount; }

    public List.Element copy() {
        return new Record(amount); }
}

public class Main {
    public static void main(String[] args) {
        List l = new List(new Record(20), null);
        List l1 = l.clone(), l2;
        try { l2 = l.deepClone(); }
        catch (NullPointerException e) {}
    }
}
```

Fig. 1 Our running example

There is however a major drawback in the use of traditional denotational semantics to build static analyses. Namely, denotations only model the *functional i.e.,* input/output behaviour of the code: they do not model the state of the computation at internal program points *i.e.,* before the execution of the code has finished; we call *localised* this missing information about the input/internal program points behaviours. The derived static analyses inherit this drawback, which makes them of little practical use, with some notable exceptions, such as analyses which are not localised at internal program points, as for instance strictness analysis. Consider for instance the Java code in Fig. 1, which implements a list of mutable records. Those lists provide two cloning methods: `clone` returns a shallow copy of a list while `deepClone` performs a deep copy, where also the records are copied. Method `main` builds a list of one element and then calls `clone` and `deepClone`. We have used

a list of one element in order to keep the subsequent examples simple, but nothing would change by using longer lists. In `main`:

1. the return value of `clone` *shares* its record with the list `l`. Moreover, it is not a *cyclical* list, since `l` was not a cyclical list;
2. the return value of `deepClone` does not share anything with `l` nor with `l1`, since it is a deep copy, and it is not cyclical;
3. the calls `head.copy()` and the recursive call `tail.deepClone()` inside method `deepClone()` always find a non-null receiver bound to `head` and `tail`, respectively. Hence the `catch` statement in method `main` is not actually needed.

Sharing analysis of pairs of variables, *cyclicity* analysis of variables and *nullness* analysis of variables, based on denotational semantics and implemented with a *pair-sharing domain* [38], a *cyclicity domain* [35] and a *nullness domain* [41], cannot prove any of these results, since 1 needs information at the internal program point of `main` just after the call to `clone`, 2 needs information at the internal program point of `main` just after the call to `deepClone` and 3 needs information at the internal program point of `deepClone` just before the calls to `copy` and `deepClone`.

Our contributions Our contributions here are the solution of the above limitation of denotational static analyses through the definition of a magic-sets transformation for Java bytecode that lets us apply static analyses for mono-threaded Java bytecode based on denotational semantics and developed through abstract interpretation, still getting localised information at internal program points; the proof of correctness of our magic-sets transformation and its implementation inside our JULIA denotational analyser [47]. To evaluate the resulting analysis framework, we instantiate it with five domains for pair-sharing [38], cyclicity [35], nullness [39, 41], class initialisation and size analysis of program variables [42, 43]. We show that the points 1, 2 and 3 above are solved by the first three of these analyses. We also compare our pair-sharing analysis with an operational analyser for pair-sharing [29]. JULIA is faster and scales much better. We compare our nullness analysis with the domain in [41] with that in [20], based on a constraint-based approach, showing that they scale similarly but ours is more precise. We implement our nullness analysis with the domain in [39], showing that the results are still more precise but the analysis becomes more expensive. We compare our size analysis, using the domain in [43], with a similar one inside the CLOUSOT analyser for `.NET` code [7] and show that it scales in a similar way.

This paper is organised as follows. Section 2 gives an informal introduction to our magic-sets transformation, which is then formalised in Sect. 3. Section 4 defines an operational semantics of Java bytecode which allows us, in Sect. 5, to prove the correctness of the magic-sets transformation. Section 6 defines a denotational semantics of Java bytecode and Sect. 7 proves it equivalent to the operational semantics of Sect. 4. Section 8 presents an example of abstraction of our concrete denotational semantics. The content of this last section has been already published in [41] and we refer to that paper for its proof of correctness. Section 9 shows the experiments with the magic-sets transformation applied to the sharing, cyclicity, nullness, class initialisation and size analysis of Java bytecode, comparing the results with similar analyses. Section 10 concludes the paper. A preliminary version of this paper appeared in the 2007 Static Analysis Symposium [33]. Compared to that previous conference publication, this journal version includes the treatment of exceptions and of virtual calls, provides proofs, has a larger introduction, much more examples and expanded discussions. The experiments with class initialisation analysis are also new.

Related work Java bytecode is an object-oriented language. Hence we model late binding, inheritance and overriding by using a list of possible run-time targets after each `call` bytecode, which are filtered by suitable `receiver_is` bytecodes put at the beginning of instance methods. The implementation of an instance method is only executed when its receiver has a class from where dynamic look-up of the method signature actually leads to that implementation. This *compilative* approach to late-binding is also used in [2], although the formalisation is different.

A possible solution to the localisation limitation of denotational semantics is to define a denotational *trace* semantics, in the style of what has been already done for operational semantics [37]. Traces express the sequence of states during the execution of the program and hence contain information about the states arising at internal program points. However, this obliges one to deal with complex semantical structures that implement traces of states, which results in very slow static analyses. The termination of the analyses is not guaranteed even when the domain is finite, since traces can grow indefinitely longer. Some form of *normal form* must be devised for the traces in order to keep the computation finite. Convergence can be forced by using a widening operator on traces, but heavy computations over traces are anyhow required and the definition of the widening operator is not trivial.

Another solution is to look, between the traditional (input/output) denotational semantics and a trace-based semantics, for an *intermediate* denotational semantics. An example is the *watchpoint semantics* defined by one of the authors of this paper [40]. There, traces are projected over the states occurring at a fixed set of program points, called *watchpoints*. The abstraction of those traces is computed through a relational domain binding properties at the input of the traces to properties at the output of the traces and at the watchpoints. Since the set of watchpoints is finite, the computation of the abstract analyses is finite when the abstract domain is finite. However, abstract denotations become very complex objects, dealing with a large amount of variables, because of the high number of watchpoints. Experiments have consequently shown that this does not let analyses scale to large programs.

A further possibility is the transformation of the program into static single assignment form before the analysis is performed. This means that a new fresh variable x' is introduced wherever a program variable x is updated, so that different values of x at different program points can be recovered from x itself or from x', depending on the program point. For instance, the sequence of assignments

```
x:=2
x:=x*3
```

is first transformed into

```
x:=2
x':=x*3
```

and then analysed through any kind of static analysis. A denotational analysis is well possible and gives information at internal program points: the value of x between the two assignments is the value of x at the end of the sequence, since x is not updated anymore by the second statement. To the best of our knowledge, this technique has been applied to Java bytecode in [2] only, followed by an operational analysis. A static single assignment transformation works well for relatively small methods. For larger methods, the number of assignments and hence of new extra variables grows so much that static analyses become too expensive: for most static analyses, the cost grows with the number of variables (this is the case of our cyclicity and sharing analysis, but also of size analyses based on polyhedra or other relational domains). This is very problematic in the case of Java bytecode, where

the operand stack is updated by most bytecodes, so that many extra variables are introduced by the transformation. In [2], for instance, this problem is faced and (partially) solved by program simplifications and by removing variables that are not *significant* for the kind of analysis that is performed. But not all extra variables can be removed this way. Our magic-sets transformation, instead, does not increase the number of variables used in the denotations. Another problem of static single assignment transformations is that the analyses can be easily proved to be correct *w.r.t.* the transformed program, but relating this correctness to the original, untransformed program requires an extra proof, dealing with tedious variable renamings. This extra proof becomes still more complex if some variables have been simplified away. Our magic-sets transformation is proved correct in Sect. 5 for *all* abstract analyses and is relatively simple since the bytecode is not transformed by our magic-sets transformation (only the structure of the blocks of bytecode is transformed); the developer of a new analysis will prove the correctness of its analysis as it is traditional in abstract interpretation [15] and need not even know that a magic-sets transformation is used. Finally, a single static assignment transformation is fine for property of program variables or of the part of the heap that is reachable from each program variable, such as cyclicity, sharing, nullness and size. But it does not work for other properties such as the set of classes that have been already initialised at a given program point (so that their class initialiser is not called anymore, see [25]). Such analysis abstracts the concrete states into the set of classes that have been initialised in those states and does not consider the program variables, so that a single static assignment transformation does not help. That analysis can be used with our magic-sets transformation instead, which works for any abstract domain, without any constraint.

Abstract interpretation has been applied to imperative programs since it was born [15]. In the subsequent years, it has been mainly applied, however, to functional and logic languages and denotational semantics was one of the standard reference semantics. The above problem about internal program points was solved with a *magic-sets transformation* of the program P, similar to ours, specific to logic languages, which adds extra *magic* clauses whose functional behaviours are the internal behaviours of P [5, 8, 13]. Codish [12] kept the overhead of the transformation small by exploiting the large overlapping between the clauses of P and the new magic clauses. Abstract interpretation has moved later towards mainstream imperative languages that use dynamic data structures allocated in a heap, and finally towards low-level programming languages such as Java bytecode. In that context, operational semantics has been *the* reference semantics. This was a consequence of the intuitive definition of operational semantics, very close to an actual implementation of the run-time engine of the language. Recently, the importance of denotational analyses for imperative languages has been rediscovered, when it became necessary to express relational properties of program variables through (some fashion of) polyhedral domains [26, 42] or class invariants [27]. However, those analyses were designed for relational properties (between input and output of methods and programs) rather than for properties localised at internal program points, for which magic-sets are needed.

It should be clear that, although we advocate the importance of the application of denotational semantics for abstract interpretation, we acknowledge as well that other approaches have their good points. For instance, there are formulations of operational semantics which allow the definition of *multi-variant* analyses [29]. This means that the analysis of a method can be different on the basis of the context from where it is called (*context-sensitivity*) but also on the basis of the history of the execution until the point of call. Operational semantics is also the basis of verification techniques based on model-checking, that have shown to be very effective in practice (see, for a notable example, the case of termination analysis [14]).

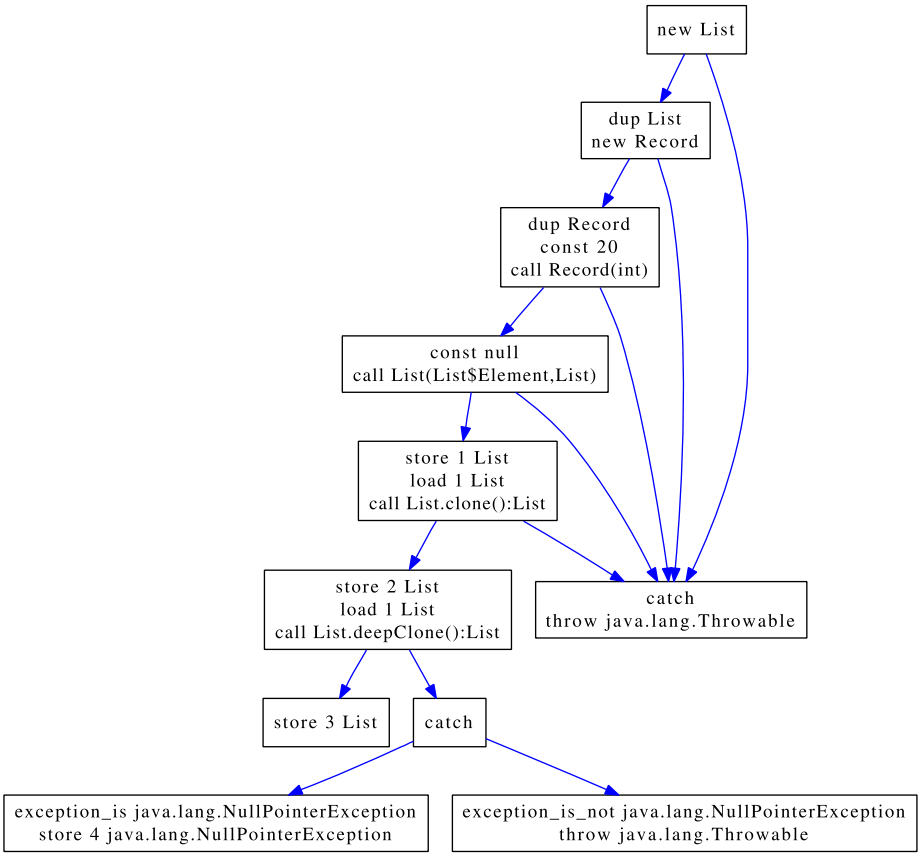

Fig. 2 The Java bytecode of the method `main` in Fig. 1

Axiomatic semantics is (more or less explicitly) the basis for verification techniques based on program annotations, performed through abstract interpretation [7] or through deductive verification by theorem proving [6, 23, 24, 31]. This often leads to the verification of very complex program properties, although a preliminary annotation is needed and the analysis might require human intervention. Moreover, a thorough program annotation is complex and error-prone and, when theorem provers are used, they often run out of memory on very large programs.

Our magic-sets transformation has similarities with the continuation-passing style (CPS) transformation [19, 44, 45] used in the functional programming setting. It is known as a folklore result in abstract interpretation that the CPS transformation and the disjunctive completion [16] of an abstract domain are equivalent, in the sense that any abstraction of the CPS transformation of a program P *w.r.t.* an abstract domain D coincides with the abstraction of P itself *w.r.t.* the disjunctive completion of D. Hence, it is sensible to expect that a domain refinement operator might exist, such that the analysis of a program P with the refined domain yields the same information as the analysis of the magic-sets transformation of P with the original, unrefined domain.

2 Our magic-sets transformation for Java bytecode

In this section we describe, informally, our representation of the Java bytecode as blocks of code, identical to that used in [43], and our magic-sets transformation. We give an intuitive explanation of why input/output information computed for that transformation is actually information at internal points of the original, untransformed program. Subsequent sections will formalise that statement and prove its correctness.

2.1 Java bytecode as blocks of code

Figure 2 reports the Java bytecode for the method `main` in Fig. 1, after a light preprocessing performed by our JULIA analyser. This preprocessing transforms the linear sequence of bytecodes into a graph of *basic blocks* [1]. Moreover, the code in Fig. 2 is *typed i.e.,* instructions are decorated with the type of their operands, and *resolved i.e.,* method and field references are bound to their corresponding definition(s). For type inference and resolution we use the official algorithms [25]. Bytecode instructions that might throw an exception occur at the end of a block linked to another block starting with a `catch` instruction. That instruction is only executed when the Java Virtual Machine has just incurred into an exception (see later for a formal definition). Figure 2 shows that, when an exception handler for the exception exists, it starts with an instruction `exception_is` for the right kind of exception. Default exception handlers, instead, throw back the exception to the caller of the method. Our analyser builds this representation of the exceptional flow automatically, from the low-level information about exception handlers coded into the `.class` file [25]. There is no `return` bytecode in our formalisation. We just assume that a method ends at any block with no outgoing arrows, with the return value (if any) as the only element left on the stack (which is otherwise empty). Method `main` is static. Instead, method `copy` of class `Record` in Fig. 1 is an instance method, that is, it has a receiver object which is stored, by default, in local variable 0. Figure 3 shows the Java bytecode for that method. It has only a default exception handler that throws back all exceptions to the caller. Note the `receiver_is` bytecode, which is put at the beginning of instance methods. It specifies the run-time class of the receiver. Namely, in Fig. 3, it tells that this method is executed only with a receiver of class `Record`. The enumeration of classes that follow the `receiver_is` is built from the class hierarchy of the program. If, for instance, the program included a subclass `SubRecord` of `Record` that does not redefine method `copy`, then a `receiver_is Record, SubRecord` would be used in Fig. 3. But if `SubRecord` redefined `copy`, then a `receiver_is Record` would be used in Fig. 3 and a `receiver_is SubRecord` would be used at the beginning of the code of `copy` inside `SubRecord`. Note the use of a block containing `nop` as final block of the method, that is needed whenever the last bytecode of the method might throw an exception (strictly speaking, this extra block is only needed if the last bytecode of the method is inside an exception handler. For simplicity, we do not distinguish that situation).

A method or constructor implementation in class κ, named m, expecting parameters of types $\vec{\tau}$ and returning a value of type t is written as $\kappa.m(\vec{\tau}) : t$. The `call` instruction implements the four `invoke`'s available in Java bytecode. It reports, as an explicit list, a superset of the target method or constructor implementations that it might call at run-time, accordingly with the semantics of the specific `invoke` that it implements and taking inheritance and overriding into account. We allow more than one implementation for late-binding, although, for simplicity, the examples in Figs. 2 and 3 have one possible target only. Dynamic lookup of the correct implementation of a method is performed by the `receiver_is` instructions put at the beginning of each instance method, as in Fig. 3. They specify for which

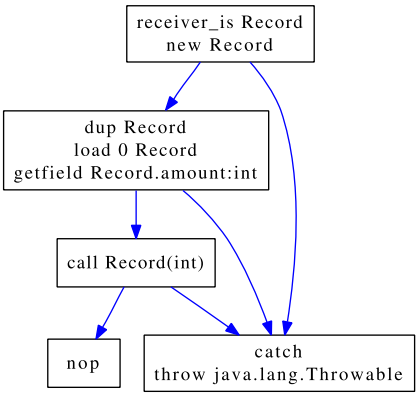

Fig. 3 The Java bytecode of the method `copy` of class `Record` in Fig. 1

run-time class of the receiver the given method is executed. For better efficiency and for better precision of the static analyses, the list of targets of a `call` bytecode is kept as smaller as possible by using a preliminary *class analysis*, that we perform as in [32].

2.2 The magic-sets transformation

In this section, we give an overview of how our magic-sets transformation works.

In order to prove that the return value of `deepClone` inside `main` never shares with local variables 0 and 1, we need information at the program point in Fig. 2 just after the `call List.deepClone():List` bytecode. In order to use a denotational analysis, our magic-sets transformation builds a new *magic* block of code whose functional behaviours are the internal behaviours just after that `call`.

Let us describe this transformation. It starts by splitting the blocks at the program points where the internal information is needed. Since the `call` to `deepClone` in Fig. 2 is already at the end of a block, there is no block to split in this case. For reasons that will be clear soon, it also splits the code before each `call`. This leads to the code in Fig. 4. Then a new *magic* block is built, one for each block of the original code. The result, for method `main`, is in Fig. 5. In that figure, the magic block for block k is marked as mk and filled in grey colour; a dashed arrow connects block k to magic block mk. Block mk contains the same bytecode instructions as block k plus a leading `blockcall` $mp_1 \cdots mp_n$, where $p_1, \ldots, p_n$ are the predecessors of block k, if any. For instance, since block 11 has five predecessors 0, 1, 3, 5 and 7, block $m11$ starts with `blockcall m0 m1 m3 m5 m7` *i.e.,* there are five ways of reaching block 11.

We note that the functional behaviour of magic block mk coincides with the internal behaviour at the end of block k. For instance, the functional behaviours of $m7$ is a denotation from the input state provided to `main` to the intermediate states just after the call to `clone`. To understand why, let us start from $m0$. It is a clone of block 0. At its end, the computation reaches hence the intermediate states at the internal program points between 0 and 1. Block $m1$ executes $m0$ (because of the `blockcall m0` instruction) and then the same instructions as block 1. At its end, the computation reaches hence the intermediate states at the internal program point between 1 and 2. The same reasoning applies to the other magic blocks, until $m7$.

Let us consider method `copy` from Fig. 3 now. Its magic-sets transformation is shown in Fig. 6. Its construction is similar to that of method `main` but for the first magic block $m15$, which starts with a `blockcall mx my` instruction. The latter specifies that method `copy`

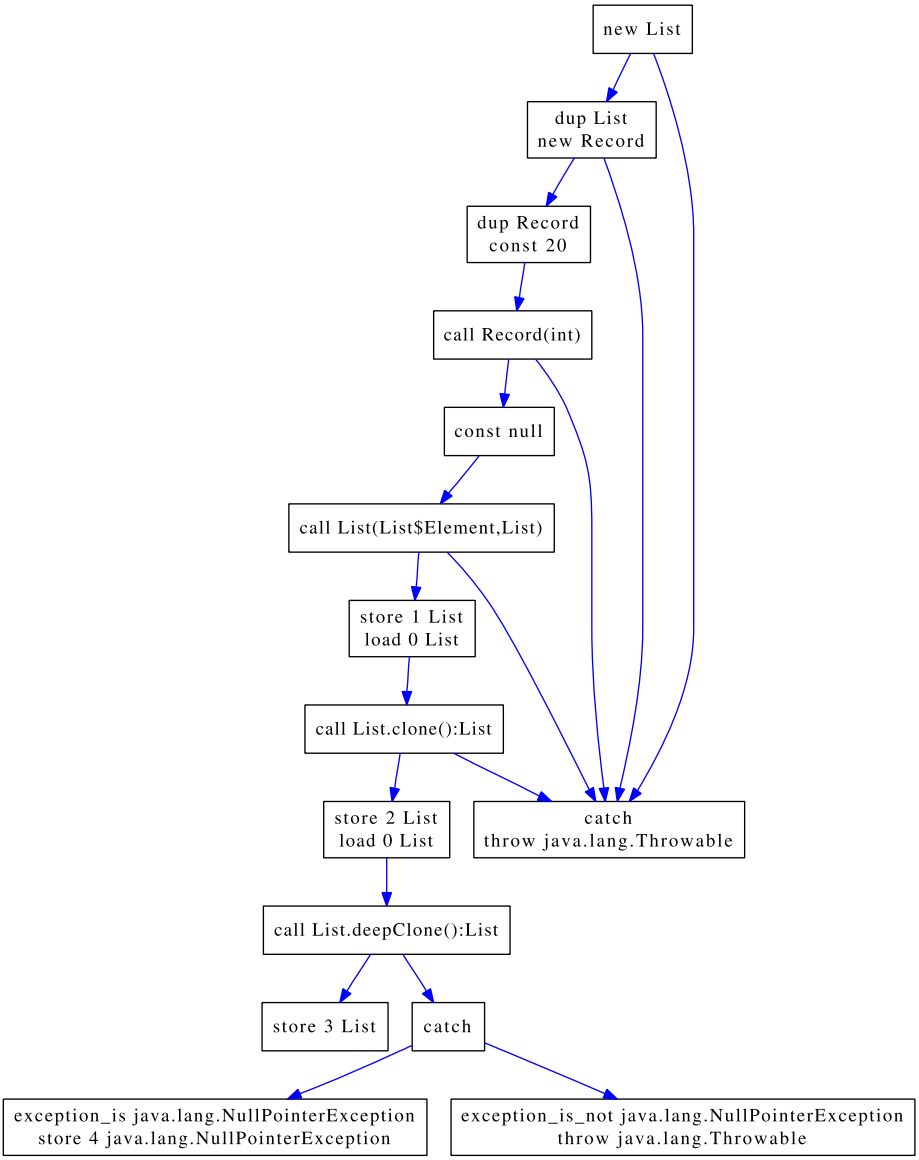

Fig. 4 The Java bytecode of the method `main` in Fig. 1 after the first phase of the magic-sets transformation. Blocks of code have been split at `call` bytecodes and after each internal program point where the abstract information is required by the analysis

is called from a state arising just before one of the calls to `copy` inside method `deepClone` in Fig. 1. Blocks mx and my are the magic-blocks derived from the original blocks x and y of `deepClone` that immediately precede those two calls, respectively. More generally, a method which is not `main` starts with a `blockcall` instruction that enumerates all blocks where a call to the method occurs. This is why we split the code before each `call`: to allow the states of the callers at the call points to flow into the callees. The subsequent

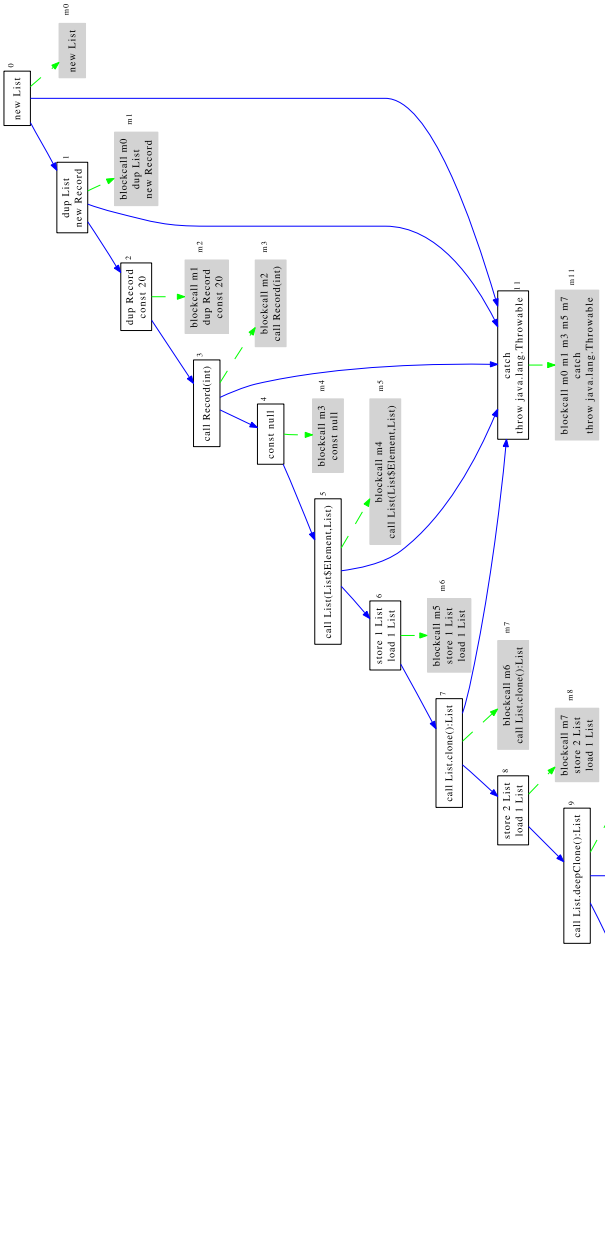

Fig. 5 The magic-sets transformation of the Java bytecode for method `main` in Fig. 2

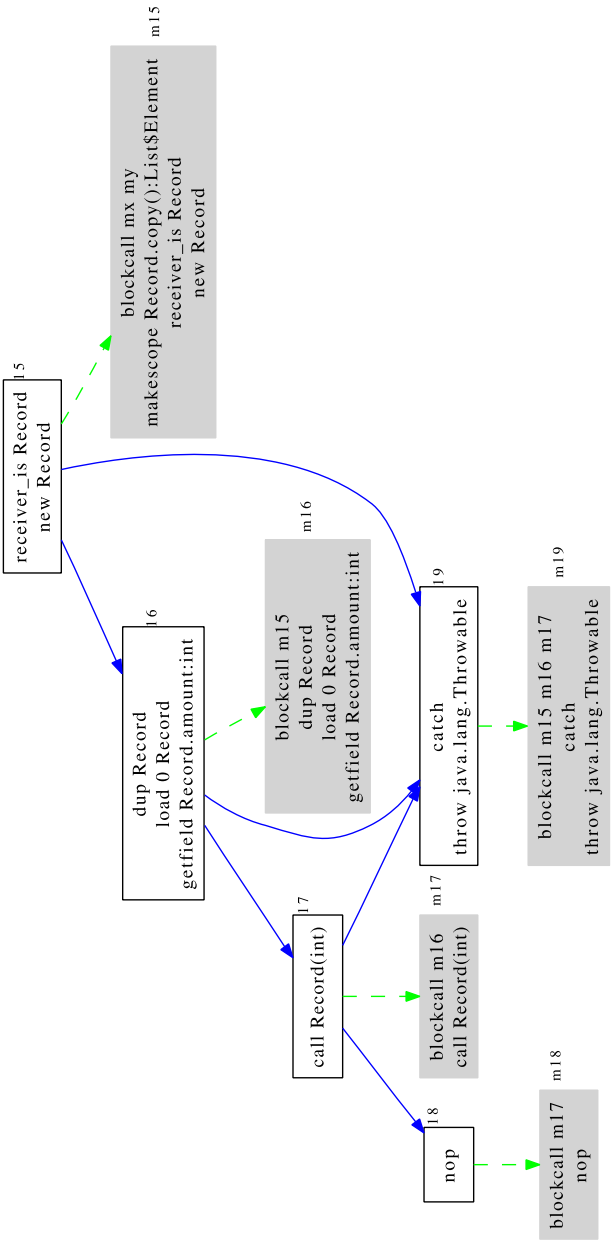

Fig. 6 The magic-sets transformation of the Java bytecode for method `copy` of class `Record` in Fig. 2

instruction `makescope Record.copy():List$Element` builds the initial state for `copy`: in Java bytecode, the caller stores the actual arguments on the operand stack and the callee retrieves them from the local variables [25]. Hence this `makescope` copies the only argument of the method, that is the implicit `this` parameter left on top of the stack by the callers of this method, into local variable 0 and clears the operand stack of `copy`.

3 A formalisation of our magic-sets transformation

We formalise here the magic-sets transformation. From now on we assume that P is a program *i.e.,* a set of blocks as those in Fig. 5. This partition is arbitrary, as long as the following constraints are satisfied. In general, however, it is better to keep the number of blocks small, since this decreases the memory requirement for the representation of the program. In our implementation, we use a partition of the code of the program into *basic blocks* [1], which gives the largest possible extension to the blocks. The choice of the partition, however, has no effect on the precision of the analyses.

We make the following assumptions about the blocks of P:

1. The starting block of a method has no predecessors and does not start with a `call` bytecode. This does not introduce any loss of generality since it is always possible to add an extra initial block containing `nop`, whenever the first bytecode of a method is a `call` or has some predecessor;
2. The blocks that do not start a method have at least a predecessor, since otherwise they would be dead-code and could be eliminated;
3. Each `call` bytecode is at the beginning of a block;
4. Each `return` ends a block with no successors;
5. The `main` method is not called from inside the program. This does not introduce any loss of generality since we can always rename `main` into `main'` wherever in the program and add a new `main` that wraps a call to `main'`.

Original blocks are labelled with k and magic blocks with mk with $k \in \mathbb{N}$. If ℓ is a label, $P(\ell)$ is block ℓ of P. We write block ℓ with n bytecode instructions and m immediate successor blocks $b_1, \ldots, b_m$, with $m, n \geq 0$, as

$$\boxed{\begin{array}{l}\mathtt{ins}_1\\ \mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_n\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \overset{\cdots}{b_m}\end{array} \quad \text{or just as} \quad \boxed{\begin{array}{l}\mathtt{ins}_1\\ \mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_n\end{array}}^{\ell} \quad \text{when } m = 0.$$

The *magic-sets transformation* of P builds a *magic block* mk for each block k.

Definition 1 The *magic block* mk, with $k \in \mathbb{N}$, is built from $P(k)$ as

$$magic\Big(\underbrace{\boxed{code}^{k} \begin{array}{l}\rightarrow b_1\\ \rightarrow \overset{\cdots}{b_m}\end{array}}_{P(k)}\Big) = \begin{cases} \boxed{\begin{array}{l}\mathtt{blockcall}\ mp_1 \cdots mp_l\\ code\end{array}}^{mk} & \text{if } l > 0 \\ \boxed{\begin{array}{l}\mathtt{blockcall}\ mq_1 \cdots mq_u\\ \mathtt{makescope}\ \kappa.m(\vec{\tau}) : t\\ code\end{array}}^{mk} & \text{if } l = 0 \text{ and } u > 0 \\ \boxed{code}^{mk} & \text{if } l = 0 \text{ and } u = 0 \end{cases}$$

where $p_1, \ldots, p_l$ are the predecessors of $P(k)$ and $q_1, \ldots, q_u$ those of the blocks of P that begin with a `call` $\ldots\kappa.m(\vec{\tau}) : t \ldots$ and method $\kappa.m(\vec{\tau}) : t$ starts at block k.

Definition 1 has three cases. In the first case, block k does not start a method (nor a constructor). Hence it has $l > 0$ predecessors and magic block mk begins with a `blockcall` to the magic blocks of these predecessors.

Example 1 Block 11 in Fig. 5 has $l = 5$ predecessors 0, 1, 3, 5 and 7. Hence block $m11$ in Fig. 5 is derived from block 11:

$$magic\left(\boxed{\begin{array}{l}\texttt{catch}\\ \texttt{throw java.lang.Throwable}\end{array}}^{11}\right) = \boxed{\begin{array}{l}\texttt{blockcall m0 m1 m3 m5 m7}\\ \texttt{catch}\\ \texttt{throw java.lang.Throwble}\end{array}}^{m11}.$$

In the second and third case of Definition 1, block k starts a method or constructor $\kappa.m(\vec{\tau}) : t$; hence that block has no predecessors. If the program P calls $\kappa.m(\vec{\tau}) : t$ (second case), those `calls` have $u > 0$ predecessors, since we assume that `call` does not start a method. Magic block mk calls those predecessors and then uses `makescope` to build the scope for $\kappa.m(\vec{\tau}) : t$.

Example 2 Block 15 in Fig. 6 starts the method `copy`, thus $l = 0$. The latter is called at the beginning of two blocks of `deepClone` whose only predecessors are some blocks mx and my, respectively. Hence $u = 2$ and we have

$$magic\left(\boxed{\begin{array}{l}\texttt{receiver\_is Record}\\ \texttt{new Record}\end{array}}^{15} \begin{array}{l}\rightarrow 16\\ \rightarrow 19\end{array}\right) = \boxed{\begin{array}{l}\texttt{blockcall mx my}\\ \texttt{makescope Record.copy() : List\$Element}\\ \texttt{receiver\_is Record}\\ \texttt{new Record}\end{array}}^{m15}.$$

Otherwise (third case), P never calls $\kappa.m(\vec{\tau}) : t$ and mk is a clone of k. This would be the case of the first block of method `main` (yielding block $m0$ in Fig. 5).

Definition 2 The *magic-sets transformation* of P is obtained by adding to P all blocks mk where $k \in \mathbb{N}$ and $P(k)$ is a block of P.

A nice property of our magic-sets transformation of a program P is that the strongly-connected components of its blocks are never larger than those of P itself. This is important since, otherwise, the cost of a bottom-up static analysis, as that we are going to define in Sect. 8, based on the denotational semantics of Sect. 6, might easily explode.

Lemma 1 *Let P be a program and P' its magic-sets transformation. If two magic blocks $mk_1, mk_2 \in P'$ belong to the same strongly-connected component of blocks of P', then $k_1, k_2 \in P$ belong to the same strongly-connected component of blocks of P.*

Proof The successors of a block $b \rightarrow \begin{array}{l} b_1 \\ \vdots \\ b_m \end{array}$ are $b_1, \ldots, b_m$ but also the initial block of every method called inside b and blocks $mp_1, \ldots, mp_l$ if b contains an instruction `blockcall` $mp_1 \cdots mp_l$. Note that the last case is only possible when b is a magic block, since original blocks in P do not contain `blockcall`'s. Also, when b is a magic block we

have $m = 0$ (Definition 1). We write $b_1 < b_2$ to mean that b_2 is a successor of b_1 and we allow that relationship to be transitive and reflexive. Note that the original blocks in P are not modified by the magic-sets transformation and do not contain `blockcall`'s, so that the successors of an original, non-magic block are always original, non-magic blocks. Consider hence mk_1 and mk_2. Since they belong to the same strongly-connected component of P', there is a sequence of blocks $mk'_1 < \cdots < mk'_s$ where mk'_{j+1} is an immediate successor of mk'_j for every $1 \leq j < s$, $mk_1 = mk'_1$ and $mk_2 = mk'_s$ and there is a sequence of blocks $mk''_1 < \cdots < mk''_w$ such that mk''_{j+1} is an immediate successor of mk''_j for every $1 \leq j < w$, $mk_2 = mk''_1$ and $mk_1 = mk''_w$. Consider the first sequence. All its blocks are magic, since otherwise, as said before, mk'_s should be an original, non-magic block, but we know that $mk'_s = mk_2$, which is magic. The only way, for a magic block mk'_{j+1}, to be a successor of another magic block mk'_j is that mk'_j starts with an instruction `blockcall` $mp_1 \cdots mp_x$ such that mk'_{j+1} belongs to $mp_1 \cdots mp_x$. By Definition 1, this entails that whether the original blocks k'_j and k'_{j+1} are such that $k'_{j+1} < k'_j$ (first case of Definition 1) or rather k'_{j+1} is the immediate predecessor of a block c that begins with a `call` $\cdots \kappa.m(\vec{\tau}) : t \cdots$ and block k'_j is the first block of method $\kappa.m(\vec{\tau}) : t$ (second case of Definition 1). In that last case, we have $k'_{j+1} < c < k'_j$. In all cases, we conclude that $k'_s < \cdots < k'_1$, with $k_1 = k'_1$ and $k_2 = k'_s$. By considering the second sequence, we similarly conclude that $k''_w < \cdots < k''_1$, with $k_2 = k''_1$ and $k_1 = k''_w$. It follows that k_1 and k_2 must belong to the same strongly-connected component of blocks of P (since, again, the successors of original blocks are always original blocks). □

Proposition 1 *The strongly-connected components of the blocks of the magic-sets transformation P' of P are never larger than those of the blocks of P itself.*

Proof Let c be a strongly-connected component of blocks of P'. It cannot contain both original and magic blocks, since a magic block is never a successor of an original block, so they cannot belong to the same component. Hence whether c contains original blocks only, but then it is a component of P, or c contains magic blocks only, but then all those magic blocks are derived from distinct original blocks of P (Definition 1) and those original blocks must belong to the same strongly-connected component (Lemma 1). In conclusion, there is always a component c' of blocks of P that is at least as large as c. □

4 Operational semantics of the Java bytecode

In this section we describe an operational semantics of the Java bytecode, that we use in Sect. 5 to prove our magic-sets transformation correct. We are aware that there are other operational semantics for Java bytecode already. However, we need an operational semantics exclusively defined in terms of *state transformers* or *denotations*, since it will be matched later to a denotational semantics. Our formalisation is nevertheless indebted to [21], where Java and Java bytecode are mathematically formalised and the compilation of Java into bytecode and its type-safeness are machine-proved. Our formalisation of the state of the Java Virtual Machine (Definition 4) is similar to theirs, with the exception that we do not use a program counter nor keep the name of the current method and class inside the state. We avoid program counters by using blocks of code linked by arrows as concrete representation of the structure of the bytecode. Also our formalisation of the heap and of the objects inside the heap is identical to theirs.

Our operational semantics keeps a *state*, providing values for the variables of the program. An *activation stack* of states is used to model the method call mechanism, exactly as in an actual implementation of the Java Virtual Machine. For simplicity, we assume that the only primitive type is `int` and the only reference types are the *classes*; we only allow *instance* fields and methods. Our implementation deals instead with full sequential Java bytecode and all Java types.

Definition 3 (Classes) The set of *classes* $\mathbb{K}$ in program P is partially ordered *w.r.t.* $\leq$, that expresses the subclass relationship. A *type* is an element of $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}\}$. A class $\kappa \in \mathbb{K}$ has *instance fields* $\kappa.f : t$ (field f of type $t \in \mathbb{T}$ defined in class κ) and *instance methods* $\kappa.m(\vec{\tau}) : t$ (method m with arguments of type $\vec{\tau} \subseteq \mathbb{T}$, returning a value of type $t \in \mathbb{T} \cup \{\texttt{void}\}$, defined in class κ). We consider constructors as methods returning `void`.

A *state* provides *values* to program variables.

Definition 4 (State) A *value* is an element of $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$, where $\mathbb{L}$ is an infinite set of *memory locations*. A *state* is a triple $\langle l \parallel s \parallel \mu \rangle$ where l is an array of values (the *local variables*), s a stack of values (the *operand stack*), that grows leftwards, and μ a *memory*, or *heap*, that binds locations to *objects*. The empty stack is written ε. An object o belongs to class $o.\kappa \in \mathbb{K}$ (is an *instance* of $o.\kappa$) and maps identifiers (the fields f of class $o.\kappa$ and of its superclasses) into values $o.f$. The set of states is Ξ. We write $\Xi_{i,j}$ when we want to fix the number i of local variables and j of stack elements. A value v *has type* t in a state $\langle l \parallel s \parallel \mu \rangle$ if $v \in \mathbb{Z}$ and $t = \texttt{int}$, or $v = \texttt{null}$ and $t \in \mathbb{K}$, or $v \in \mathbb{L}$, $t \in \mathbb{K}$ and $\mu(v).\kappa \leq t$.

Example 3 State $\sigma = \langle [\ell] \parallel \ell' \parallel \mu \rangle \in \Xi_{1,1}$ is a possible state at the beginning of the execution of block 16 in Fig. 6. Location ℓ must be bound to the receiver object `this` in μ. Location ℓ' must point to an object, freshly created and pushed on the stack by the `new Record` instruction at the end of block 15. It must be the case that $\mu(\ell').\texttt{amount} = 0$, since the constructor of class `Record` has not been called yet and the field `amount` still holds its default value 0.

The Java Virtual Machine allows exceptions. Hence we distinguish *normal* states Ξ *i.e.,* those arising during the normal execution of a piece of code, from *exceptional* states $\underline{\Xi}$, arising *just after* a bytecode that throws an exception. States in $\underline{\Xi}$ have always a stack of height 1 containing a location (bound to the thrown exception object). We write them underlined in order to distinguish them from the normal states.

Definition 5 (Java Virtual Machine State) The set of *Java Virtual Machine states* (from now on just *states*) with i local variables and j stack elements is $\Sigma_{i,j} = \Xi_{i,j} \cup \underline{\Xi}_{i,1}$.

When we denote a state by σ, we mean that we do not specify if it is normal or exceptional. If we want to stress that we deal with a normal or with an exceptional state, then we write $\langle l \parallel s \parallel \mu \rangle$ in the first case and $\underline{\langle l \parallel s \parallel \mu \rangle}$ in the second.

The semantics of a bytecode `ins` is a partial map $ins : \Sigma_{i_1,j_1} \to \Sigma_{i_2,j_2}$ from an *initial* to a *final* state *i.e.,* a *denotation* [49]. The indices i_1, j_1, i_2, j_2 depend on the program point where the bytecode occurs.

Definition 6 (Denotation) A *denotation* is a partial map from an *input* or *initial* state to an *output* or *final* state; the set of denotations is Δ. If we want to stress how many local

variables and stack elements are in the states, we write $\Delta_{i_1,j_1 \to i_2,j_2} = \Sigma_{i_1,j_1} \to \Sigma_{i_2,j_2}$. The *sequential composition* of $\delta_1 \in \Delta_{i_1,j_1 \to i_2,j_2}$ and $\delta_2 \in \Delta_{i_2,j_2 \to i_3,j_3}$ is $\delta_1; \delta_2 \in \Delta_{i_1,j_1 \to i_3,j_3}$, where $\delta_1; \delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$, that is undefined when $\delta_1(\sigma)$ or $\delta_2(\delta_1(\sigma))$ is undefined.

In the composition of denotations $\delta_1; \delta_2$, the idea is that δ_1 describes the behaviour of an instruction ins_1, δ_2 that of an instruction ins_2 and $\delta_1; \delta_2$ describes the behaviour of the sequential execution of ins_1 followed by ins_2. The fact that we allow partial denotations is important since we model conditional jumps as an (apparently) non-deterministic choice between two continuations. However, only one of the bytecodes at the beginning of the two continuations will be defined.

Size and type of local variables and stack elements at each program point are statically known [25]. In the following we silently assume that the bytecodes are run in a program point with i local variables and j stack elements and that the semantics of the bytecodes is undefined for input states of wrong sizes or types. These assumptions are required by [25]. Code that does not satisfy them cannot be run and is not legal Java bytecode.

We now give examples of the specification of the semantics of some bytecodes.

Basic instructions

Bytecode `const` v pushes $v \in \mathbb{Z} \cup \{\texttt{null}\}$ on the stack. Formally, its semantics is the denotation

$$const\ v = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel v :: s \parallel \mu\rangle.$$

The λ-notation defines a partial map. Since $\langle l \parallel s \parallel \mu\rangle$ (where s might be ε) is not underlined, the map is undefined on exceptional states *i.e.,* the bytecode is executed when the Java Virtual Machine is not in an exceptional state. This is the case of *all* bytecodes but `catch`, that starts the exceptional handlers from an exceptional state. Bytecode `dup` t duplicates the top of the stack, that must have type t:

$$dup\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l \parallel top :: top :: s \parallel \mu\rangle.$$

Bytecode `load` i t pushes on the stack the value of local variable number i, that must exist and have type t:

$$load\ i\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel l[i] :: s \parallel \mu\rangle.$$

Conversely, bytecode `store` i t pops the top of the stack of type t and writes it in local variable i:

$$store\ i\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle.\langle l[i \mapsto top] \parallel s \parallel \mu\rangle.$$

If l contains less than $i + 1$ variables, the resulting set of local variables gets expanded. The semantics of a conditional bytecode is undefined when its condition is false. For instance, `if_ne` t checks if the top of the stack, that must have type t, is not 0 when $t =$ `int` or is not `null` otherwise:

$$if\_ne\ t = \lambda\langle l \parallel top :: s \parallel \mu\rangle. \begin{cases} \langle l \parallel s \parallel \mu\rangle & \text{if } top \neq 0 \text{ and } top \neq \texttt{null}, \\ undefined & \text{otherwise.} \end{cases}$$

Undefined here means that the denotation is a partial rather than total function. It corresponds to the fact that the Java Virtual Machine does not continue the execution of the code if the condition is false. Note that, in our formalisation, conditional bytecodes are

used in complementary pairs (for instance, `if_ne` and `if_eq`), at the beginning of the two branches of a condition, so that only one of them is defined for each given state. This is exactly as for the selection of the right exception handler, shown in Fig. 2.

Object-manipulating instructions

Some bytecodes create or access objects in memory. Bytecode `new` κ pushes on the stack a reference to a new object n of class κ, with reference fields initialised to `null`:

$$new\ \kappa = \lambda\langle l \parallel s \parallel \mu\rangle.\begin{cases}\langle l \parallel \ell :: s \parallel \mu[\ell \mapsto n]\rangle & \text{if there is enough memory,}\\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto oome]\rangle} & \text{otherwise}\end{cases}$$

with $\ell \in \mathbb{L}$ fresh and *oome* new instance of `java.lang.OutOfMemoryError`. This is the first example of a bytecode that can throw an exception. Bytecode `getfield` $\kappa.f:t$ reads field $\kappa.f:t$ of the object pointed by the top *rec* (the *receiver*) of the stack, that has type κ:

$$getfield\ \kappa.f:t = \lambda\langle l \parallel rec::s \parallel \mu\rangle.\begin{cases}\langle l \parallel \mu(rec).f :: s \parallel \mu\rangle & \text{if } rec \neq \texttt{null},\\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto npe]\rangle} & \text{otherwise}\end{cases}$$

with $\ell \in \mathbb{L}$ fresh and *npe* new instance of `java.lang.NullPointerException`. This is the first example of a bytecode that dereferences a location (*rec*) and might hence throw an exception. Another is `putfield` $\kappa.f:t$, that writes the top of the stack, of type t, inside field $\kappa.f:t$ of the object pointed by a value *rec* below the top of the stack, of type κ (ℓ and *npe* are as before):

$$putfield\ \kappa.f:t = \lambda\langle l \parallel top::rec::s \parallel \mu\rangle.\begin{cases}\langle l \parallel s \parallel \mu[\mu(rec).f \mapsto top]\rangle\\ \text{if } rec \neq \texttt{null},\\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto npe]\rangle}\\ \text{otherwise.}\end{cases}$$

Exception handling instructions

Bytecode `throw` κ throws, explicitly, the object pointed by the top of the stack, of type $\kappa \leq$ `java.lang.Throwable` (ℓ and *npe* are as before):

$$throw\ \kappa = \lambda\langle l \parallel top::s \parallel \mu\rangle.\begin{cases}\underline{\langle l \parallel top \parallel \mu\rangle} & \text{if } top \neq \texttt{null},\\ \underline{\langle l \parallel \ell \parallel \mu[\ell \mapsto npe]\rangle} & \text{if } top = \texttt{null}.\end{cases}$$

Bytecode `catch` starts an exception handler. It takes an exceptional state and transforms it into a normal state, subsequently used by the bytecodes implementing the handler:

$$catch = \lambda\underline{\langle l \parallel top \parallel \mu\rangle}.\langle l \parallel top \parallel \mu\rangle$$

where $top \in \mathbb{L}$ has type `java.lang.Throwable`. Note that `catch` is undefined on all normal states, so that, for instance, in Fig. 2, the computation can enter the exception handlers only if the previous instruction yields an exceptional state. After `catch`, the appropriate exception handler is selected on the basis of the run-time class of the exception. To

that purpose, we use a bytecode `exception_is` K that filters those states whose top of the stack points to an instance of a class in $K \subseteq \mathbb{K}$:

$$exception\_is\ K = \lambda\langle l \parallel top \parallel \mu\rangle.\ \begin{cases} \langle l \parallel top \parallel \mu\rangle & \text{if } top \in \mathbb{L} \text{ and } \mu(top).\kappa \in K, \\ undefined & \text{otherwise.} \end{cases}$$

We also use `exception_is_not` K, that can be seen as a shortcut for `exception_is` H, where H is the set of exception classes that are not instances of some class in K. Alternatively, an explicit definition of that instruction is also possible.

Method calls

When a caller transfers the control to a callee $\kappa.m(\vec{\tau}) : t$, the Java Virtual Machine performs an operation *makescope* $\kappa.m(\vec{\tau}) : t$ that copies the topmost stack elements into the corresponding local variables and clears the stack.

Definition 7 Let $\kappa.m(\vec{\tau}) : t$ be a method or constructor and p the number of stack elements needed to hold its actual parameters, including the implicit parameter `this`, if any. We define $(makescope\ \kappa.m(\vec{\tau}) : t) : \Sigma \to \Sigma$ as

$$makescope\ \kappa.m(\vec{\tau}) : t = \lambda\langle l \parallel v_{p-1} :: \cdots :: v_0 :: s \parallel \mu\rangle.\langle [v_0, \ldots, v_{p-1}] \parallel \varepsilon \parallel \mu\rangle.$$

Definition 7 formalises the fact that the ith local variable of the callee is a copy of the element located at $(p-1)-i$ positions down the top of the stack of the caller.

The operational semantics

We can now define the operational semantics of our language.

Definition 8 A *configuration* is a pair $\langle b \parallel \sigma\rangle$ of a block b (not necessarily in P) and a state σ. It represents the fact that the Java Virtual Machine is going to execute b in state σ. An *activation stack* is a stack $c_1 :: c_2 :: \cdots :: c_n$ of configurations, where c_1 is the topmost, *current* or *active* configuration.

We can define now the *operational semantics* of a Java bytecode program. It is a relation between activation stacks. It models the transformation of the activation stack induced by the execution of each single bytecode.

Definition 9 The (small step) operational semantics of a Java bytecode program P is a relation $a' \Rightarrow_P a''$ (P is usually omitted) providing the immediate successor activation stack a'' of an activation stack a'. It is defined by the rules:

$$\frac{\texttt{ins} \text{ is not a } \texttt{call} \text{ nor a } \texttt{blockcall},\ ins(\sigma) \text{ is defined}}{\left\langle \boxed{\begin{matrix}\texttt{ins} \\ rest\end{matrix}} \begin{matrix}\overset{\ell}{\to}\ b_1 \\ \to\ \vdots \\ \to\ b_m\end{matrix} \parallel \sigma\right\rangle :: a \Rightarrow \left\langle \boxed{rest} \begin{matrix}\overset{\ell}{\to}\ b_1 \\ \to\ \vdots \\ \to\ b_m\end{matrix} \parallel ins(\sigma)\right\rangle :: a} \tag{1}$$

$$\frac{\begin{array}{c} 1 \le i \le n,\ b \text{ is the block where method } \kappa_i.m(\vec{\tau}) : t \text{ starts} \\ \sigma = \langle l \parallel pars :: s \parallel \mu\rangle, \quad pars \text{ are the actual parameters of the call} \\ \sigma' = (makescope\ \kappa_i.m(\vec{\tau}) : t)(\sigma) \end{array}}{\left\langle \boxed{\begin{matrix}\texttt{call}\ \kappa_1.m(\vec{\tau}) : t, \ldots, \kappa_n.m(\vec{\tau}) : t \\ rest\end{matrix}} \begin{matrix}\overset{\ell}{\to}\ b_1 \\ \to\ \vdots \\ \to\ b_m\end{matrix} \parallel \sigma\right\rangle :: a \Rightarrow \langle b \parallel \sigma'\rangle :: \left\langle \boxed{rest} \begin{matrix}\overset{\ell}{\to}\ b_1 \\ \to\ \vdots \\ \to\ b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu\rangle\right\rangle :: a} \tag{2}$$

$$\frac{}{\langle \boxed{\ }^{k} \| \langle l \parallel vs \parallel \mu \rangle \rangle :: \langle b \parallel \langle l' \parallel s' \parallel \mu' \rangle \rangle :: a \Rightarrow \langle b \parallel \langle l' \parallel vs :: s' \parallel \mu \rangle \rangle :: a} \tag{3}$$

$$\frac{}{\langle \boxed{\ }^{k} \| \underline{\langle l \parallel e \parallel \mu \rangle} \rangle :: \langle b \parallel \langle l' \parallel s' \parallel \mu' \rangle \rangle :: a \Rightarrow \langle b \parallel \underline{\langle l' \parallel e \parallel \mu \rangle} \rangle :: a} \tag{4}$$

$$\frac{1 \leq i \leq m}{\langle \boxed{\begin{array}{c} \rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m \end{array}}^{k} \| \sigma \rangle :: a \Rightarrow \langle b_i \parallel \sigma \rangle :: a} \tag{5}$$

$$\frac{1 \leq i \leq l}{\langle \boxed{\begin{array}{c} \texttt{blockcall}\ mp_1 \cdots mp_l \\ rest \end{array}}^{mk} \| \sigma \rangle :: a \Rightarrow \langle P(mp_i) \parallel \sigma \rangle :: \langle \boxed{rest}^{mk} \| \sigma \rangle :: a} \tag{6}$$

$$\frac{}{\langle \boxed{\ }^{mk} \| \sigma \rangle :: \langle b \parallel \sigma' \rangle :: a \Rightarrow \langle b \parallel \sigma \rangle :: a}. \tag{7}$$

Rule (1) executes an instruction `ins`, different from `call` and `blockcall`, by using its semantics *ins*. The Java Virtual Machine moves then forward to run the rest of the instructions. Instruction `ins` might be here a `makescope`, whose semantics is given in Definition 7. Rule (2) calls a method. It looks for the block b where the latter starts and builds its initial state σ', by using *makescope*. It creates a new current configuration containing b and σ'. It removes the actual arguments from the old current configuration and the call from the instructions still to be executed at return time. Since a method call can actually call many implementations, depending on the run-time class of the receiver, this rule is apparently non-deterministic. However, only one thread of execution will continue, the one starting with the `receiver_is` bytecode for the right run-time class of the receiver. Control returns to the caller by rule (3), that rehabilitates the configuration of the caller but forces the memory to be that at the end of the execution of the callee. The return value of the callee is pushed on the stack of the caller. This rule is executed if the state reached at the end of the caller is a normal state. If it is an exceptional state, rule (4) is executed instead, that propagates the exception back to the caller. Rule (5) applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. This rule is normally deterministic, since if a block of our formalisation of the Java bytecode has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed. Rule (6) runs a `blockcall` by choosing one of the called blocks mp_i and creating a new configuration where it can run. This is true non-determinism, corresponding to the fact that there might be more ways of reaching a magic block and hence more intermediate states at an internal program point. Rule (7) applies at the end of the execution of a magic block mk. It returns the control to the caller of mk and keeps the state reached at the end of the execution of mk. Rules (1) and (2) can be used both for the original and for the magic blocks of the program; rules (3), (4) and (5) only for the original blocks, which is expressed by the use of k as label of the blocks; rules (6) and (7) only for the magic ones, which is expressed by the use of mk as label of the blocks.

From now on, when we use the notation $\Rightarrow$, we often specify the rule of Definition 9 that is used at each derivation step; for instance, we write $\underset{(1)}{\Rightarrow}$ to mean a derivation step through rule (1).

Example 4 Let $\sigma = \langle [\ell] \parallel \ell' \parallel \mu \rangle$ be the state in Example 3 and consider an arbitrary activation stack a. The first steps of the execution of block 16 in Fig. 6 are

$$\langle \boxed{\begin{array}{l}\texttt{dup Record}\\ \texttt{load 0 Record}\\ \texttt{getfield Record.amount : int}\end{array}}^{16} \rightrightarrows \begin{smallmatrix}17\\19\end{smallmatrix} \parallel \sigma \rangle :: a$$

$$\underset{(1)}{\Rightarrow} \langle \boxed{\begin{array}{l}\texttt{load 0 Record}\\ \texttt{getfield Record.amount : int}\end{array}}^{16} \rightrightarrows \begin{smallmatrix}17\\19\end{smallmatrix} \parallel \langle [\ell] \parallel \ell' :: \ell' \parallel \mu \rangle \rangle :: a$$

$$\underset{(1)}{\Rightarrow} \langle \boxed{\texttt{getfield Record.amount : int}}^{16} \rightrightarrows \begin{smallmatrix}17\\19\end{smallmatrix} \parallel \langle [\ell] \parallel \ell :: \ell' :: \ell' \parallel \mu \rangle \rangle :: a$$

$$\underset{(1)}{\Rightarrow} \langle \boxed{}^{16} \rightrightarrows \begin{smallmatrix}17\\19\end{smallmatrix} \parallel \langle [\ell] \parallel \mu(\ell).\texttt{amount} :: \ell' :: \ell' \parallel \mu \rangle \rangle :: a$$

$$\underset{(5)}{\Rightarrow} \langle \boxed{\texttt{call Record(int)}}^{17} \rightrightarrows \begin{smallmatrix}18\\19\end{smallmatrix} \parallel \langle [\ell] \parallel \mu(\ell).\texttt{amount} :: \ell' :: \ell' \parallel \mu \rangle :: a$$

$$\underset{(2)}{\Rightarrow} \langle b \parallel \langle [\ell', \mu(\ell).\texttt{amount}] \parallel \varepsilon \parallel \mu \rangle :: \langle \boxed{}^{17} \rightrightarrows \begin{smallmatrix}18\\19\end{smallmatrix} \parallel \langle [\ell] \parallel \ell' \parallel \mu \rangle :: a$$

where block b is the beginning of the constructor `Record(int)`. When rule (5) is applied, we could have continued with block 19 as well. But that execution thread would stop immediately since $\langle [\ell] \parallel \mu(\ell).\texttt{amount} :: \ell' :: \ell' \parallel \mu \rangle$ is a normal state and the `catch` instruction that is at the beginning of block 19 is undefined on that state.

Our small step operational semantics allows us to define the set of intermediate states at a given, internal program point p, provided that p ends a block k. This is not restrictive, since one can always split after p the block where p occurs. The idea of Definition 10 is to execute the program from the `main` method until the computation reaches block k and all its instructions have been executed. Hence collect all the resulting states σ.

Definition 10 (Intermediate states at a program point) Let σ_{in} be the initial state provided to the method `main` of P starting at block b_{in}. The *intermediate states* at the end of block $k \in \mathbb{N}$ during the execution of P from σ_{in} are

$$\Sigma_k^{\sigma_{in}, b_{in}} = \{\sigma \mid \langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{}^{k \rightarrow} \begin{smallmatrix} b_1 \\ \vdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a\}.$$

When σ_{in} and b_{in} are clear from the context, we write Σ_k instead of $\Sigma_k^{\sigma_{in}, b_{in}}$.

In general, Σ_k is a set since there might be more ways of reaching block k, for instance through loops or recursion.

5 Correctness of the magic-sets transformation

By using the operational semantics of Sect. 4, we show that the final states reached at the end of the execution of a magic block mk are exactly the intermediate states reached at the end of block k, before executing its successors: the functional behaviour of mk coincides hence with the internal behaviour at the end of k.

Theorem 1 (Correctness of the magic-sets transformation) *Let σ_{in} be the initial state provided to the method* `main` *of P and $k \in \mathbb{N}$ be a block of P. We have*

$$\Sigma_k = \left\{ \sigma \,\middle|\, \langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{}^{mk} \parallel \sigma \rangle \right\}.$$

The proof follows from the next two propositions. They entail the double inclusion between the left and right-hand sides of the equality in Theorem 1. In order to keep an inductive invariant, they prove stronger results than those needed for Theorem 1. Namely, they prove the inclusions when block mk is not necessarily empty at the end of the derivation, but contains instead some bytecode instructions *rest*, still to be executed.

Proposition 2 *Let P be a program and $k \in \mathbb{N}$. If*

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k \to} {}^{b_1}_{\to\, b_m} \parallel \sigma \rangle :: a$$

then

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma \rangle.$$

Proof For any $n \in \mathbb{N}$, we let $Prop_{\subseteq}(n)$ denote the property:

for any program P and any $k \in \mathbb{N}$, if

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^n \langle \boxed{rest}^{k \to} {}^{b_1}_{\to\, b_m} \parallel \sigma \rangle :: a$$

then

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma \rangle.$$

We prove by induction on n that $Prop_{\subseteq}(n)$ holds for any $n \in \mathbb{N}$.

– (Basis) We prove that $Prop_{\subseteq}(0)$ holds. Let

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^0 \langle \boxed{rest}^{k \to} {}^{b_1}_{\to\, b_m} \parallel \sigma \rangle :: a.$$

Then, $b_{in} = \boxed{rest}^{k \to} {}^{b_1}_{\to\, b_m}$, $\sigma_{in} = \sigma$ and a is empty. Notice that $P(mk) = magic(P(k))$ with $P(k) = b_{in}$. Since the first block of method `main` has no predecessors and is not called by any method, by the third case of Definition 1, we have $P(mk) = \boxed{rest}^{mk}$. As $\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle P(mk) \parallel \sigma_{in} \rangle$ we have $\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma \rangle$.

– (Induction) Suppose that for each $i \leq n$, $Prop_{\subseteq}(i)$ holds. We prove that $Prop_{\subseteq}(n+1)$ also holds. Assume that

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^{n+1} \langle \boxed{rest}^{k \to} {}^{b_1}_{\to\, b_m} \parallel \sigma \rangle :: a.$$

Then $\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^n a_n \Rightarrow \langle \boxed{rest}^{k \to} {}^{b_1}_{\to\, b_m} \parallel \sigma \rangle :: a$. Let us consider the rule of Definition 9 that is used in the last derivation step, from a_n.

1. If rule (1) is used then $a_n = \langle \boxed{\begin{array}{l}\texttt{ins}\\ rest\end{array}}^{k} \begin{array}{l}\rightarrow\\ \rightarrow\\ \vdots\\ \rightarrow\end{array}\begin{array}{l}b_1\\ \\ \\ b_m\end{array} \parallel \sigma' \rangle :: a$ and $\sigma = ins(\sigma')$. By inductive hypothesis,

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\begin{array}{l}\texttt{ins}\\ rest\end{array}}^{mk} \parallel \sigma' \rangle.$$

Moreover,

$$\langle \boxed{\begin{array}{l}\texttt{ins}\\ rest\end{array}}^{mk} \parallel \sigma' \rangle \underset{(1)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel ins(\sigma') \rangle.$$

Consequently, as $ins(\sigma') = \sigma$,

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma \rangle.$$

2. If rule (2) is used then a_n is

$$\langle \boxed{\begin{array}{l}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k'} \begin{array}{l}\rightarrow\\ \rightarrow\end{array}\cdots \parallel \sigma' \rangle :: a'$$

where $\sigma' = \langle l \parallel pars :: s \parallel \mu \rangle$ and

$$\sigma = (makescope\ \kappa.m(\vec{\tau}) : t)(\sigma').$$

Moreover, a has the form $\langle \boxed{rest}^{k'} \begin{array}{l}\rightarrow\\ \rightarrow\end{array}\cdots \parallel \langle l \parallel s \parallel \mu \rangle \rangle :: a'$. Notice that we have assumed that only the starting blocks of the methods have no predecessor and that such blocks do not start with a `call`. Consequently, $\boxed{\begin{array}{l}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k'} \begin{array}{l}\rightarrow\\ \rightarrow\end{array}\cdots$ has some predecessors, say $p_1, \ldots, p_l$. So, the derivation from $\langle b_{in} \parallel \sigma_{in} \rangle$ to a_n has the form

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \square^{p_i} \begin{array}{l}\rightarrow\\ \rightarrow\end{array}\cdots \parallel \sigma' \rangle :: a'$$

$$\underset{(5)}{\Rightarrow} \underbrace{\langle \boxed{\begin{array}{l}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k'} \begin{array}{l}\rightarrow\\ \rightarrow\end{array}\cdots \parallel \sigma' \rangle :: a'}_{a_n}$$

with $1 \leq i \leq l$.

As $\boxed{rest}^{k} \begin{array}{l}\rightarrow\\ \rightarrow\\ \vdots\\ \rightarrow\end{array}\begin{array}{l}b_1\\ \\ \\ b_m\end{array}$ is the first block of method $\kappa.m(\vec{\tau}) : t$, then it has no predecessor and $P(k) = \boxed{rest}^{k} \begin{array}{l}\rightarrow\\ \rightarrow\\ \vdots\\ \rightarrow\end{array}\begin{array}{l}b_1\\ \\ \\ b_m\end{array}$. Consequently, by the second case of Definition 1, we have

$$P(mk) = magic(P(k)) = \boxed{\begin{array}{l}\texttt{blockcall}\cdots mp_i\cdots\\ \texttt{makescope}\ \kappa.m(\vec{\tau}):t\\ rest\end{array}}^{mk}.$$

So, we have:

$$\langle P(mk) \parallel \sigma_{in} \rangle \underset{(6)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{\begin{array}{l}\texttt{makescope}\ \kappa.m(\vec{\tau}):t\\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle$$

and, by inductive hypothesis,

$$\langle P(mp_i) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{}^{mp_i} \parallel \sigma' \rangle.$$

Consequently,

$$\begin{aligned}\langle P(mk) \parallel \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{}^{mp_i} \parallel \sigma' \rangle :: \langle \boxed{\begin{array}{l}\texttt{makescope}\ \kappa.m(\vec{\tau}) : t \\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle \\ &\underset{(7)}{\Rightarrow} \langle \boxed{\begin{array}{l}\texttt{makescope}\ \kappa.m(\vec{\tau}) : t \\ rest\end{array}}^{mk} \parallel \sigma' \rangle \\ &\underset{(1)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel \underbrace{(makescope\ \kappa.m(\vec{\tau}) : t)(\sigma')}_{\sigma} \rangle.\end{aligned}$$

3. If rule (3) is used then a_n has the form

$$\langle \boxed{}^{k'} \parallel \langle l' \parallel vs \parallel \mu' \rangle \rangle :: \langle \boxed{rest}^{k \to \atop \to} {b_1 \atop \vdots \atop b_m} \parallel \langle l \parallel s \parallel \mu \rangle \rangle :: a$$

and $\sigma = \langle l \parallel vs :: s \parallel \mu' \rangle$. Notice that rule (3) corresponds to a situation when control returns to the caller $P(k)$, since the only rule that can create a new top configuration with an original (*i.e.*, non-magic) block is rule (2). As a `call` instruction is always located at the beginning of a block, we have

$$P(k) = \boxed{\begin{array}{l}\texttt{call}\ \ldots\kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{k \to \atop \to} {b_1 \atop \vdots \atop b_m}.$$

Hence $P(k)$ has some predecessors (because we have assumed that only the starting blocks of the methods have no predecessor and that such blocks do not start with a `call`), say $p_1, \ldots, p_x$. So, the derivation from $\langle b_{in} \parallel \sigma_{in} \rangle$ to a_n has the form

$$\begin{aligned}\langle b_{in} \parallel \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{}^{p_i \to \atop \to} \cdots \parallel \sigma_i \rangle :: a \\ &\underset{(5)}{\Rightarrow} \langle\, P(k) \parallel \sigma_i \,\rangle :: a \\ &\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_i' \rangle :: \langle \boxed{rest}^{k \to \atop \to} {b_1 \atop \vdots \atop b_m} \parallel \langle l \parallel s \parallel \mu \rangle \rangle :: a \\ &\Rightarrow^* \underbrace{\langle \boxed{}^{k'} \parallel \langle l' \parallel vs \parallel \mu' \rangle \rangle :: \langle \boxed{rest}^{k \to \atop \to} {b_1 \atop \vdots \atop b_m} \parallel \langle l \parallel s \parallel \mu \rangle \rangle :: a}_{a_n}\end{aligned}$$

where $1 \le i \le x$, $\sigma_i = \langle l \parallel pars :: s \parallel \mu \rangle$, $\sigma_i' = (makescope\ \kappa.m(\vec{\tau}) : t)(\sigma_i)$ and b is the starting block of $\kappa.m(\vec{\tau}) : t$. Moreover, $\langle b \parallel \sigma_i' \rangle \Rightarrow^* \langle \boxed{}^{k'} \parallel \langle l' \parallel vs \parallel \mu' \rangle \rangle$. By the first case of Definition 1, we have

$$P(mk) = magic(P(k)) = \boxed{\begin{array}{l}\texttt{blockcall}\ mp_1 \cdots mp_x \\ \texttt{call}\ \ldots\kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{mk}.$$

Then,

$$\langle P(mk) \parallel \sigma_{in}\rangle \underset{(6)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in}\rangle :: \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{mk} \parallel \sigma_{in}\rangle$$

and, by inductive hypothesis,

$$\langle P(mp_i) \parallel \sigma_{in}\rangle \Rightarrow^* \langle \square^{mp_i} \parallel \sigma_i\rangle.$$

So we have

$$\begin{aligned}\langle P(mk) \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \square^{mp_i} \parallel \sigma_i\rangle :: \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{mk} \parallel \sigma_{in}\rangle\\ &\underset{(7)}{\Rightarrow} \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{mk} \parallel \sigma_i\rangle\\ &\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_i'\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle.\end{aligned}$$

Since we have observed that $\langle b \parallel \sigma_i'\rangle \Rightarrow^* \langle \square^{k'} \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle$, we conclude that

$$\begin{aligned}\langle P(mk) \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \square^{k'} \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle\\ &\underset{(3)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel \langle l \parallel vs :: s \parallel \mu'\rangle\rangle\end{aligned}$$

i.e., $\langle P(mk) \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma\rangle$.

4. If rule (4) is used then a_n has the form

$$\langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle :: a$$

and $\sigma = \underline{\langle l \parallel e \parallel \mu'\rangle}$. Notice that rule (4) corresponds to a situation when control returns to the caller $\overline{P(k)}$, since the only rule that can create a new top configuration with an original block is rule 2. As a `call` instruction is always located at the beginning of a block, we have

$$P(k) = \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\ b_m\end{array}.$$

Hence $P(k)$ has some predecessors (because we have assumed that only the starting blocks of the methods have no predecessor and that such blocks do not start with a `call`), say $p_1, \ldots, p_x$. So, the derivation from $\langle b_{in} \parallel \sigma_{in}\rangle$ to a_n has the form

$$\begin{aligned}\langle b_{in} \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \square^{p_i \rightarrow} \cdots \parallel \sigma_i\rangle :: a\\ &\underset{(5)}{\Rightarrow} \langle P(k) \parallel \sigma_i\rangle :: a\\ &\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_i'\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle :: a\\ &\Rightarrow^* \underbrace{\langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle :: a}_{a_n}\end{aligned}$$

where $1 \leq i \leq x$, $\sigma_i = \langle l \parallel pars :: s \parallel \mu \rangle$, $\sigma_i' = (makescope\ \kappa.m(\vec{\tau}) : t)(\sigma_i)$ and b is the starting block of $\kappa.m(\vec{\tau}) : t$. Moreover, $\langle b \parallel \sigma_i' \rangle \Rightarrow^* \langle \boxed{\ }^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle$. By the first case of Definition 1, we have

$$P(mk) = magic(P(k)) = \boxed{\begin{array}{l}\texttt{blockcall}\ mp_1 \cdots mp_l \\ \texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{mk}.$$

Then,

$$\langle P(mk) \parallel \sigma_{in} \rangle \underset{(6)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{\begin{array}{l}\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle$$

and, by inductive hypothesis,

$$\langle P(mp_i) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\ }^{mp_i} \parallel \sigma_i \rangle.$$

So we have

$$\begin{aligned} \langle P(mk) \parallel \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{\ }^{mp_i} \parallel \sigma_i \rangle :: \langle \boxed{\begin{array}{l}\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle \\ &\underset{(7)}{\Rightarrow} \langle \boxed{\begin{array}{l}\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{mk} \parallel \sigma_i \rangle \\ &\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_i' \rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu \rangle \rangle. \end{aligned}$$

Since we have observed that $\langle b \parallel \sigma_i' \rangle \Rightarrow^* \langle \boxed{\ }^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle$, we conclude that

$$\begin{aligned} \langle P(mk) \parallel \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{\ }^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu \rangle \rangle \\ &\underset{(4)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel \underline{\langle l \parallel e \parallel \mu' \rangle} \rangle \end{aligned}$$

i.e., $\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma \rangle$.

5. If rule (5) is used then a_n has the form $\langle \boxed{\ }^{k' \rightarrow\ b_1' \ldots b_{m'}'} \parallel \sigma \rangle :: a$ and $\boxed{rest}^{k \rightarrow\ b_1 \ldots b_m}$ is a b_i'. Then, $\boxed{rest}^{k \rightarrow\ b_1 \ldots b_m} = P(k)$. By the first case of Definition 1,

$$P(mk) = magic(P(k)) = \boxed{\begin{array}{l}\texttt{blockcall} \cdots mk' \cdots \\ rest\end{array}}^{mk}.$$

Hence,

$$\langle P(mk) \parallel \sigma_{in} \rangle \underset{(6)}{\Rightarrow} \langle P(mk') \parallel \sigma_{in} \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle.$$

As $\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^n a_n$, by inductive hypothesis we have

$$\langle P(mk') \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\ }^{mk'} \parallel \sigma \rangle.$$

Consequently,

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{}^{mk'} \parallel \sigma \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle$$
$$\underset{(7)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel \sigma \rangle.$$

6. Rule (6) cannot be used. Indeed, b_{in} is an original block and an original block does not call any magic block. Hence, the block in a_n is not a magic block.
7. Rule (7) cannot be used for the same reason as above. □

Proposition 3 *Let P be a program and $k \in \mathbb{N}$. If*

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{mk} \parallel \sigma \rangle$$

where rest does not contain any `blockcall` *nor* `makescope` *then*

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k \rightarrow\ b_1 \ldots \rightarrow\ b_m} \parallel \sigma \rangle :: a$$

for some a (*the hypothesis on rest is not restrictive, since in Theorem 1 rest is empty*).

Proof For any $n \in \mathbb{N}$, we let $Prop_{\supseteq}(n)$ denote the property:

for any $k \in \mathbb{N}$, if

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^n \langle \boxed{rest}^{mk} \parallel \sigma \rangle$$

where *rest* does not contain any `blockcall` nor `makescope` then

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k \rightarrow\ b_1 \ldots \rightarrow\ b_m} \parallel \sigma \rangle :: a$$

for some a.

We prove by induction on n that $Prop_{\supseteq}(n)$ holds for any $n \in \mathbb{N}$.

– (Basis) We prove that $Prop_{\supseteq}(0)$ holds. Let $k \in \mathbb{N}$. Suppose that

$$\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^0 \langle \boxed{rest}^{mk} \parallel \sigma \rangle$$

where *rest* does not contain any `blockcall` nor `makescope`. Then, $\sigma_{in} = \sigma$ and $P(mk) = \boxed{rest}^{mk}$, so $P(mk)$ does not contain any `blockcall` nor `makescope`. So, as $P(mk) = magic(P(k))$, $P(mk)$ is obtained from the third case of Definition 1. Consequently:

- $P(k) = \boxed{rest}^{k \rightarrow\ b_1 \ldots \rightarrow\ b_m}$,
- $P(k)$ has no predecessor, so $P(k)$ is the starting block of a method $\kappa.m(\vec{\tau}) : t$,
- each block of P starting with `call` $\ldots \kappa.m(\vec{\tau}) : t \ldots$ has no predecessor, hence it is the starting block of a method; as the starting block of any method does not start with a `call`, no block of P starts with `call` $\ldots \kappa.m(\vec{\tau}) : t \ldots$. Then $\kappa.m(\vec{\tau}) : t$ is the method `main` and $P(k) = b_{in}$.

Therefore, as $\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle b_{in} \parallel \sigma_{in}\rangle$ with $\sigma_{in} = \sigma$, we have

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{rest}^{k \to\ \vdots\ \to} {}^{b_1}_{b_m} \parallel \sigma\rangle.$$

– (Induction) Suppose that for each $i \leq n$, $Prop_{\supseteq}(i)$ holds. We prove that $Prop_{\supseteq}(n+1)$ also holds. Suppose that

$$\langle P(mk) \parallel \sigma_{in}\rangle \Rightarrow^{n+1} \langle \boxed{rest}^{mk} \parallel \sigma\rangle$$

where *rest* does not contain any `blockcall` nor `makescope`. Then, we have $\langle P(mk) \mid \mid \sigma_{in}\rangle \Rightarrow^n a_n \Rightarrow \langle \boxed{rest}^{mk} \parallel \sigma\rangle$. Let us consider the rule of Definition 9 that is used in the derivation from a_n.

1. If rule (1) is used then a_n has the form $\langle \boxed{\begin{matrix}\texttt{ins}\\ rest\end{matrix}}^{mk} \parallel \sigma'\rangle$ and $\sigma = ins(\sigma')$. If `ins` is not a `makescope` then, by inductive hypothesis,

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\begin{matrix}\texttt{ins}\\ rest\end{matrix}}^{k \to\ \vdots\ \to} {}^{b_1}_{b_m} \parallel \sigma'\rangle :: a.$$

Moreover,

$$\langle \boxed{\begin{matrix}\texttt{ins}\\ rest\end{matrix}}^{k \to\ \vdots\ \to} {}^{b_1}_{b_m} \parallel \sigma'\rangle :: a \underset{(1)}{\Rightarrow} \langle \boxed{rest}^{k \to\ \vdots\ \to} {}^{b_1}_{b_m} \parallel ins(\sigma')\rangle :: a.$$

Consequently, as $ins(\sigma') = \sigma$,

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{rest}^{k \to\ \vdots\ \to} {}^{b_1}_{b_m} \parallel \sigma\rangle :: a.$$

If $\texttt{ins} = \texttt{makescope}\ \kappa.m(\vec{\tau}) : t$ then, as $P(mk) = magic(P(k))$, by the second case of Definition 1, that is the only case that introduces a `makescope` instruction in the code, we have:

- $P(k) = \boxed{rest}^{k \to\ \vdots\ \to} {}^{b_1}_{b_m}$,
- $P(k)$ is the starting block of method $\kappa.m(\vec{\tau}) : t$,
- $P(mk) = \boxed{\begin{matrix}\texttt{blockcall}\ \cdots mk' \cdots\\ \texttt{makescope}\ \kappa.m(\vec{\tau}) : t\\ rest\end{matrix}}^{mk}$,
- $P(k')$ is a predecessor of a block of P, say $P(k'')$, that begins with $\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots$.

Moreover, the derivation from $\langle P(mk) \parallel \sigma_{in}\rangle$ to a_n has the form

$$\begin{aligned}\langle P(mk) \parallel \sigma_{in}\rangle &\underset{(6)}{\Rightarrow} \langle P(mk') \parallel \sigma_{in}\rangle :: \langle \boxed{\begin{matrix}\texttt{ins}\\ rest\end{matrix}}^{mk} \parallel \sigma_{in}\rangle\\ &\Rightarrow^* \langle \square^{mk'} \parallel \sigma'\rangle :: \langle \boxed{\begin{matrix}\texttt{ins}\\ rest\end{matrix}}^{mk} \parallel \sigma_{in}\rangle\\ &\underset{(7)}{\Rightarrow} \underbrace{\langle \boxed{\begin{matrix}\texttt{ins}\\ rest\end{matrix}}^{mk} \parallel \sigma'\rangle}_{a_n}\end{aligned}$$

where $\langle P(mk') \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\ }^{mk'} \parallel \sigma'\rangle$ in less than n steps. So, by inductive hypothesis, we have

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\ }^{k'} \overrightarrow{\ } \cdots \parallel \sigma'\rangle :: a'.$$

As $P(k'')$ is a successor of $P(k')$ and $P(k'')$ begins with

$$\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots$$

we have

$$\begin{aligned}
\langle b_{in} \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \boxed{\ }^{k'} \overrightarrow{\ } \cdots \parallel \sigma'\rangle :: a' \\
&\underset{(5)}{\Rightarrow} \langle \underbrace{\boxed{\begin{array}{c}\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{k''} \overrightarrow{\ } \cdots}_{P(k'')} \parallel \sigma'\rangle :: a' \\
&\underset{(2)}{\Rightarrow} \langle P(k) \parallel ins(\sigma')\rangle :: \langle \boxed{rest}^{k''} \overrightarrow{\ } \cdots \parallel \sigma''\rangle :: a'
\end{aligned}$$

since $\texttt{ins} = \texttt{makescope}\ \kappa.m(\vec{\tau}) : t$. Hence, since we have $P(k) = \boxed{rest}^{k} \overrightarrow{\ } \begin{matrix} b_1 \\ \vdots \\ b_m \end{matrix}$ and $\sigma = ins(\sigma')$:

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{rest}^{k} \overrightarrow{\ } \begin{matrix} b_1 \\ \vdots \\ b_m \end{matrix} \parallel \sigma\rangle :: a.$$

2. Rule (2) cannot be used because otherwise the length of the resulting activation stack would be at least equal to 2. Here, the resulting activation stack is $\langle \boxed{rest}^{mk} \parallel \sigma\rangle$, whose length is equal to 1.
3. If rule (3) is used then a_n has the form

$$\langle \boxed{\ }^{k'} \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

and $\sigma = \langle l \parallel vs :: s \parallel \mu'\rangle$. Notice that rule (3) corresponds to a situation when control returns to the caller $P(mk)$. So, the derivation from $\langle P(mk) \parallel \sigma_{in}\rangle$ to a_n has the form

$$\begin{aligned}
\langle P(mk) \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots \\ rest\end{array}}^{mk} \parallel \sigma_1\rangle \\
&\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_1'\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle \\
&\Rightarrow^* \underbrace{\langle \boxed{\ }^{k'} \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle}_{a_n}
\end{aligned}$$

where $\sigma_1 = \langle l \parallel pars :: s \parallel \mu\rangle$, $\sigma_1' = (makescope\ \kappa.m(\vec{\tau}) : t)(\sigma_1)$, b is the starting block of $\kappa.m(\vec{\tau}) : t$ and

$$\langle b \parallel \sigma_1'\rangle \Rightarrow^* \langle \boxed{\ }^{k'} \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle.$$

Note that the block $\boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{mk}$ does not contain any `blockcall` nor `makescope`, since *rest* does not contain them. Hence, by inductive hypothesis:

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma_1\rangle.$$

Consequently,

$$\begin{aligned}\langle b_{in} \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma_1\rangle\\ &\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_1'\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle\\ &\Rightarrow^* \langle \square^{k'} \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle\\ &\underset{(3)}{\Rightarrow} \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \langle l \parallel vs :: s \parallel \mu'\rangle\rangle\end{aligned}$$

i.e.,

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma\rangle.$$

4. If rule (4) is used then a_n has the form

$$\langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

and $\sigma = \underline{\langle l \parallel e \parallel \mu'\rangle}$. Notice that rule (4) corresponds to a situation when control returns to the caller $P(mk)$. So, the derivation from $\langle P(mk) \parallel \sigma_{in}\rangle$ to a_n has the form

$$\begin{aligned}\langle P(mk) \parallel \sigma_{in}\rangle &\Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{mk} \parallel \sigma_1\rangle\\ &\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_1'\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle\\ &\Rightarrow^* \underbrace{\langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle :: \langle \boxed{rest}^{mk} \parallel \langle l \parallel s \parallel \mu\rangle\rangle}_{a_n}\end{aligned}$$

where $\sigma_1 = \langle l \parallel pars :: s \parallel \mu\rangle$, $\sigma_1' = (makescope\ \kappa.m(\vec{\tau}):t)(\sigma_1)$, b is the starting block of $\kappa.m(\vec{\tau}):t$ and

$$\langle b \parallel \sigma_1'\rangle \Rightarrow^* \langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle.$$

Note that the block $\boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{mk}$ does not contain any `blockcall` nor `makescope`, since *rest* does not contain them. Hence, by inductive hypothesis:

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma_1\rangle.$$

Consequently,

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{call}\ \ldots\kappa.m(\vec{\tau}):t\ldots\\ rest\end{array}}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma_1\rangle$$

$$\underset{(2)}{\Rightarrow} \langle b \parallel \sigma_1'\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

$$\Rightarrow^* \langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle :: \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

$$\underset{(4)}{\Rightarrow} \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \underline{\langle l \parallel e \parallel \mu'\rangle}\rangle$$

i.e.,

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma\rangle.$$

5. Rule (5) cannot be used. Indeed, in this rule the top of the resulting activation stack is $\langle b_i \parallel \sigma\rangle$ where b_i is not a magic block, while here $\boxed{rest}^{mk}$ is a magic block.
6. Rule (6) cannot be used because otherwise the length of the resulting activation stack would be at least equal to 2. Here, the resulting activation stack is $\langle \boxed{rest}^{mk} \parallel \sigma\rangle$, whose length is equal to 1.
7. If rule (7) is used then a_n has the form

$$\langle \square^{mk'} \parallel \sigma\rangle :: \langle \boxed{rest}^{mk} \parallel \sigma'\rangle.$$

Since only rule 6 pushes a magic block on top of the stack, block $P(mk)$ has the form $\boxed{\begin{array}{c}\texttt{blockcall}\ \cdots mk' \cdots\\ rest\end{array}}^{mk}$ and the derivation

$$\langle P(mk) \parallel \sigma_{in}\rangle \Rightarrow^* a_n$$

has the form

$$\langle P(mk) \parallel \sigma_{in}\rangle \underset{(6)}{\Rightarrow} \langle P(mk') \parallel \sigma_{in}\rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in}\rangle$$

$$\Rightarrow^{n-1} \underbrace{\langle \square^{mk'} \parallel \sigma\rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in}\rangle}_{a_n}$$

where $\langle P(mk') \parallel \sigma_{in}\rangle \Rightarrow^{n-1} \langle \square^{mk'} \parallel \sigma\rangle$. Moreover, as *rest* does not contain any `makescope`, $P(mk)$ is obtained from $P(k)$ using the first case of Definition 1. Consequently, $P(k)$ has the form $\boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array}$ and $P(k')$ is a predecessor of $P(k)$. By inductive hypothesis, $\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \square^{k'} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \cdots \parallel \sigma\rangle :: a$. Hence

$$\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \square^{k'} \begin{array}{l}\rightarrow\\ \rightarrow\end{array} \cdots \parallel \sigma\rangle :: a$$

$$\underset{(5)}{\Rightarrow} \langle \boxed{rest}^{k} \begin{array}{l}\rightarrow\ b_1\\ \rightarrow\ \vdots\\ \rightarrow\ b_m\end{array} \parallel \sigma\rangle :: a$$

i.e., $\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k \rightarrow \; \rightarrow b_m}_{\rightarrow \; \vdots \; \rightarrow b_1} \parallel \sigma \rangle :: a$. □

In Sect. 6 we define a denotational semantics for Java bytecode and prove it *equivalent* to our operational semantics of Sect. 4 *w.r.t.* functional behaviours. By Theorem 1, we will conclude that the denotational semantics of *mk* is the internal behaviour at the end of block k.

6 Denotational semantics of Java bytecode

A denotational semantics for Java bytecode maps each block of code b in a *denotation* $[\![b]\!]$ (Definition 6) *i.e.,* in a partial function from an *initial* state at the beginning of b to an *output* or *final* state at the end of the execution of the code starting at b and continuing with the following blocks until no successor blocks are found anymore. Hence, if b_{in} is the initial block of method `main`, then $[\![b_{in}]\!]$ is the functional behaviour of the whole program. The use of *partial* functions allows one to model divergence.

The semantics *ins* of a bytecode `ins` (Sect. 4) is a denotation for `ins`. However, we never gave any definition for the semantics of `call`, that must reflect the execution of the callee method(s). A key feature of denotational semantics is that the semantics of a method (function, procedure...) is computed once and then *extended* or *plugged* at each calling point to the method. This is possible since the denotation of a method, as every denotation, is a function from the initial state to the final state of the method. Hence, it can be instantiated in the context of every calling point by providing a possibly different initial state to the method. This feature is called *context-sensitivity* of the semantics or *compositionality w.r.t.* method calls.

Let hence $\delta \in \Delta$ be the functional behaviour of a method $\kappa.m(\vec{\tau}) : t$. As the Java Virtual Machine specification requires, at its beginning the operand stack is empty and local variables hold the arguments of the call. At its end the operand stack holds the return value of $\kappa.m(\vec{\tau}) : t$ only, if any. From the point of view of a caller executing a call to method $\kappa.m(\vec{\tau}) : t$, its local variables and its operand stack do not change, except for the arguments of the call which get popped from the stack and replaced by the return value of $\kappa.m(\vec{\tau}) : t$, if any. The final memory is that reached at the end of the execution of $\kappa.m(\vec{\tau}) : t$. These considerations let us *extend* δ into the denotation of the `call` instruction.

Definition 11 (*extend*) Let $\delta \in \Delta$ and $\kappa.m(\vec{\tau}) : t$ be a method. We define the operator *extend* $\kappa.m(\vec{\tau}) : t \in \Delta \mapsto \Delta$ as

$$(extend\ \kappa.m(\vec{\tau}) : t)(\delta)(\langle l \parallel pars :: s \parallel \mu \rangle) = \langle l \parallel vs :: s \parallel \mu' \rangle$$

if $\delta((makescope\ \kappa.m(\vec{\tau}) : t)(\langle l \parallel pars :: s \parallel \mu \rangle)) = \langle l' \parallel vs \parallel \mu' \rangle$, where *pars* are the parameters passed to $\kappa.m(\vec{\tau}) : t$ and *vs* its return value, if any. If instead $\delta((makescope\ \kappa.m(\vec{\tau}) : t)(\langle l \parallel pars :: s \parallel \mu \rangle)) = \underline{\langle l' \parallel e \parallel \mu' \rangle}$, then we define

$$(extend\ \kappa.m(\vec{\tau}) : t)(\delta)(\langle l \parallel pars :: s \parallel \mu \rangle) = \underline{\langle l \parallel e \parallel \mu' \rangle}.$$

Definition 11 considers both the case when the method returns normally and the case when it throws an exception, that is propagated back to the caller.

Traditionally, the denotational semantics of a program is an *interpretation* that specifies the behaviour of each function of the program. Since Java bytecode is made of blocks of

instructions, our interpretations actually specify the behaviour of each *block* in the program by providing a set of denotations for each block. *Sets* can express non-deterministic behaviours, which means for us that we can observe more intermediate states between blocks. The operations *extend* and ; over denotations are consequently extended to sets of denotations.

Definition 12 (Interpretation) An *interpretation* for P is a map from P's blocks into sets of denotations. The set of interpretations $\mathbb{I}$ is ordered by pointwise set-inclusion. The least upper bound operator $\sqcup$ over $\mathbb{I}$ is pointwise $\cup$.

Given an interpretation ι providing the functional behaviour of the blocks of P, we can determine the functional behaviour $[\![b]\!]^\iota$ of the code starting at a given block b, not necessarily in P, that can call methods and blocks of P, and continues with its successor blocks until a block with no successors is reached. Namely, the denotation of a block composes, sequentially, the denotations of the instructions inside the block and then continues by composing the result with the denotations of its successors $b_1, \ldots, b_n$. The denotations of the instructions inside the block are those given in Sect. 4, except for `call`, that *extend*s the denotation of the first block of the called method(s), as provided by ι; and of `blockcall`, that joins, non-deterministically, the denotations of the called blocks, as provided by the interpretation ι.

Definition 13 (Denotations of instructions and blocks) Let $\iota \in \mathbb{I}$. The *denotations in ι of an instruction* are

$$[\![\texttt{ins}]\!]^\iota = \{ins\} \quad \text{if ins is not a } \texttt{call} \text{ nor a } \texttt{blockcall}$$

$$[\![\texttt{blockcall}\ mp_1 \cdots mp_l]\!]^\iota = \iota(P(mp_1)) \cup \cdots \cup \iota(P(mp_l))$$

$$[\![\texttt{call}\ \kappa_1.m(\vec{\tau}) : t, \ldots, \kappa_n.m(\vec{\tau}) : t]\!]^\iota = \bigcup_{1 \le i \le n} (extend\ \kappa_i.m(\vec{\tau}) : t)(\iota(b_{\kappa_i.m(\vec{\tau}):t}))$$

where $b_{\kappa_i.m(\vec{\tau}):t}$ is the block where method or constructor $\kappa_i.m(\vec{\tau}) : t$ starts. The function $[\![\_]\!]^\iota$ is extended to blocks as

$$\left[\!\!\left[\boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_n\end{array}}^{\ell} \right]\!\!\right]^\iota = [\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_n]\!]^\iota$$

$$\left[\!\!\left[\boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_n\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \cdots\\ \rightarrow b_m\end{array} \right]\!\!\right]^\iota = [\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_n]\!]^\iota ; (\iota(b_1) \cup \cdots \cup \iota(b_m))$$

with the assumption that if $n = 0$ then $[\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_n]\!]^\iota = \{id\}$, where the identity denotation id is such that $id = \lambda\sigma.\sigma$.

The blocks of P are in general interdependent, because of loops and recursion, and a denotational semantics must be built through a fixpoint computation. Given an empty approximation $\iota \in \mathbb{I}$ of the denotational semantics, such that $\iota(b) = \emptyset$ for every block b of P, one improves it into $T_P(\iota) \in \mathbb{I}$ and iterates the application of T_P until a fixpoint *i.e.,* the denotational semantics of P (our implementation JULIA performs smaller fixpoints on each strongly-connected component of blocks rather than a huge fixpoint over all blocks. This is important for efficiency reasons but irrelevant here for our theoretical results).

Definition 14 (Transformer on interpretations) The *transformer* $T_P : \mathbb{I} \mapsto \mathbb{I}$ for P is defined as $T_P(\iota)(b) = [\![b]\!]^\iota$ for every $\iota \in \mathbb{I}$ and block b of P.

We want to prove that the operator T_P is additive (Proposition 4), which entails that its least fixpoint exists and can be built through an iterative computation (Definition 15). To that purpose, we first state that the interpretation of a single bytecode is an isomorphism *w.r.t.* possibly infinite union of interpretations.

Lemma 2 *Let* `ins` *be a bytecode instruction and* $\{\iota_j\}_{j\in J} \subseteq \mathbb{I}$ *with* $J \subseteq \mathbb{N}$. *Then*

$$[\![\texttt{ins}]\!]^{\bigsqcup_{j\in J}\iota_j} = \bigcup_{j\in J} [\![\texttt{ins}]\!]^{\iota_j}.$$

Proof If `ins` is not a `call` nor a `blockcall`, then

$$[\![\texttt{ins}]\!]^{\bigsqcup_{j\in J}\iota_j} = \{ins\} = \bigcup_{j\in J} [\![\texttt{ins}]\!]^{\iota_j}.$$

If `ins` is a `call` $\kappa_1.m(\vec{\tau}) : t, \ldots, \kappa_n.m(\vec{\tau}) : t$ and $b_{\kappa_i.m(\vec{\tau}):t}$ is the block where $\kappa_i.m(\vec{\tau}) : t$ starts then, since *extend* has been extended to sets of denotations:

$$\begin{aligned}
[\![\texttt{ins}]\!]^{\bigsqcup_{j\in J}\iota_j} &= \bigcup_{1\le i\le n} (extend\ \kappa_i.m(\vec{\tau}) : t)\left(\left(\bigsqcup_{j\in J}\iota_j\right)(b_{\kappa_i.m(\vec{\tau}):t})\right)\\
&= \bigcup_{1\le i\le n} (extend\ \kappa_i.m(\vec{\tau}) : t)\left(\bigcup_{j\in J}\iota_j(b_{\kappa_i.m(\vec{\tau}):t})\right)\\
&= \bigcup_{1\le i\le n}\bigcup_{j\in J}(extend\ \kappa_i.m(\vec{\tau}) : t)(\iota_j(b_{\kappa_i.m(\vec{\tau}):t}))\\
&= \bigcup_{j\in J}\bigcup_{1\le i\le n} (extend\ \kappa_i.m(\vec{\tau}) : t)(\iota_j(b_{\kappa_i.m(\vec{\tau}):t}))\\
&= \bigcup_{j\in J} [\![\texttt{ins}]\!]^{\iota_j}.
\end{aligned}$$

If `ins` is a `blockcall` $mp_1 \cdots mp_l$ then

$$\begin{aligned}
[\![\texttt{ins}]\!]^{\bigsqcup_{j\in J}\iota_j} &= \left(\bigsqcup_{j\in J}\iota_j\right)(P(mp_1)) \cup \cdots \cup \left(\bigsqcup_{j\in J}\iota_j\right)(P(mp_l))\\
&= \left(\bigcup_{j\in J}\iota_j(P(mp_1))\right) \cup \cdots \cup \left(\bigcup_{j\in J}\iota_j(P(mp_l))\right)\\
&= \bigcup_{j\in J}\left(\iota_j(P(mp_1)) \cup \cdots \cup \iota_j(P(mp_l))\right)\\
&= \bigcup_{j\in J} [\![\texttt{ins}]\!]^{\iota_j}.
\end{aligned}$$

□

We can now prove that T_P is additive.

Proposition 4 *The operator T_P is additive, so its least fixpoint exists and is $\bigsqcup_{i\geq 0} T_P^i$, where $T_P^0(b) = \varnothing$ for every block b of P and $T_P^{i+1} = T_P(T_P^i)$ for every $i \geq 0$* [46].

Proof Let $\{\iota_j\}_{j\in J} \subseteq \mathbb{I}$ with $J \subseteq \mathbb{N}$. We have to prove additivity, that is:

$$T_P\left(\bigsqcup_{j\in J} \iota_j\right)(b) = \left(\bigsqcup_{j\in J} T_P(\iota_j)\right)(b)$$

for all blocks b.

Let first b be $\boxed{\begin{array}{c}\mathtt{ins_1}\\ \cdots\\ \mathtt{ins_k}\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \\ \rightarrow b_m\end{array}$ with $k > 0$ and $m > 0$ (the cases when $k = 0$ or $m = 0$ are considered later). We have:

$$\begin{aligned} T_P\left(\bigsqcup_{j\in J} \iota_j\right)(b) &= [\![b]\!]^{\bigsqcup_{j\in J}\iota_j} \\ &= [\![\mathtt{ins_1}]\!]^{\bigsqcup_{j\in J}\iota_j} ; \cdots ; [\![\mathtt{ins_n}]\!]^{\bigsqcup_{j\in J}\iota_j} ; \left(\left(\bigsqcup_{j\in J} \iota_j\right)(b_1) \cup \cdots \cup \left(\bigsqcup_{j\in J} \iota_j\right)(b_m)\right) \end{aligned}$$

which by Lemma 2 is equal to

$$\bigcup_{j\in J} [\![\mathtt{ins_1}]\!]^{\iota_j} ; \cdots ; \bigcup_{j\in J} [\![\mathtt{ins_n}]\!]^{\iota_j} ; \left(\bigcup_{j\in J} (\iota_j(b_1)) \cup \cdots \cup \bigcup_{j\in J} (\iota_j(b_m))\right). \tag{8}$$

Since ; is the extention of ; over sets of denotations, it is by definition additive; the same holds for $\cup$. Since the composition of additive functions is additive, (8) can be rewritten into

$$\begin{aligned} &\bigcup_{j\in J} \left([\![\mathtt{ins_1}]\!]^{\iota_j} ; \cdots ; [\![\mathtt{ins_n}]\!]^{\iota_j} ; \left(\iota_j(b_1) \cup \cdots \cup \iota_j(b_m)\right)\right) \\ &\quad = \bigcup_{j\in J} [\![b]\!]^{\iota_j} = \bigcup_{j\in J} (T_P(\iota_j)(b)) \\ &\quad = \left(\bigsqcup_{j\in J} T_P(\iota_j)\right)(b). \end{aligned}$$

The cases when $k = 0$ or $m = 0$ follow similarly: when $k = 0$ we remove the interpretations of the instructions $\mathtt{ins_1}, \ldots, \mathtt{ins_m}$; when $m = 0$ we remove the interpretations of the blocks $b_1, \ldots, b_m$. □

Definition 15 (Denotational semantics) Let P be a Java bytecode program (possibly enriched with its magic blocks). Its *denotational semantics* $\mathcal{D}_P$ is the least fixpoint $\bigsqcup_{i\geq 0} T_P^i$ of T_P.

In general, Definition 15 does not provide an effective way for computing the least fixpoint of T_P, since the number of iterations required to reach the fixpoint might be infinite. However, abstract interpretation [15] allows one to replace sets of denotations with abstract domain elements, expressing some *property* of the denotations. If the number of abstract

domain elements is finite or if the abstract domain satisfies at least the *ascending chain condition*, then the computation of the resulting *abstract* semantics is feasible in a finite number of fixpoint iterations. If the abstract domain does not satisfy the ascending chain condition, termination can still be enforced by fixpoint acceleration techniques such as *widening* [15].

6.1 Handling of subroutines

Our operational and denotational semantics have been defined for a restricted, still representative subset of the actual Java bytecode. The way other bytecodes are accommodated inside our framework should be relatively clear from the examples of bytecodes considered in Sects. 4 and 6. We just describe here the most complex scenario of the `jsr` and `ret` bytecodes, that are used to implement *subroutines* in Java bytecode [25]. Subroutines are portions of code inside a method or constructor that can be called from different points of the same method or constructor and hence *shared* across different execution paths. They were used almost exclusively to compile the `finally` blocks of Java, that are always executed at the end of a `try` block. It seemed sensible, to the developers of Java, to share the code inside the `finally` block by letting it be called at the end of all normal as well as at the end of all exceptional executions of the code inside the `try` block or inside its exception handlers. Note, however, that `jsr` and `ret` might also be used for other purposes and a static analyser for Java bytecode must be able to deal with those instructions in a general way.

The static requirements of the Java Virtual Machine [25] prescribe that the height of the stack and the type of its elements at the beginning of a subroutine are statically known, as well as the number and type of the local variables. It is actually possible to call a subroutine with different number and type of local variables, but then the local variables that do not always exist or have conflicting type, depending on the calling point, cannot be accessed inside the subroutine and are typed with the special type *unused*.

We show here an example of code using the `jsr` and `ret` instructions, taken from [25]. Consider a Java class `Example` defining a method

```
void tryCatchFinally() {
  try {
    tryItOut();
  } catch (TestExc e) {
    handleExc(e);
  } finally {
    wrapItOut();
  }
}
```

The `wrapItOut()` method is *always* called at the end of the execution of `tryCatchFinally()`: this happens when `tryItOut()` does not throw any exception, as well as when the exception is a `TestExc` and `handleExc(e)` does not throw any exception, as well as when the exception is a `TestExc` and `handleExc(e)` throws an exception, as well as when the exception is not a `TestExc`. In the last two cases, the `finally` block is executed and then the exception is thrown back to the caller of `tryCatchFinally()`. Old Java compilers would compile `tryCatchFinally()` by using a subroutine (modern Java compilers do not generate subroutines anymore). The resulting bytecode is represented in our framework as in Fig. 7. The subroutine is the block of code starting with the

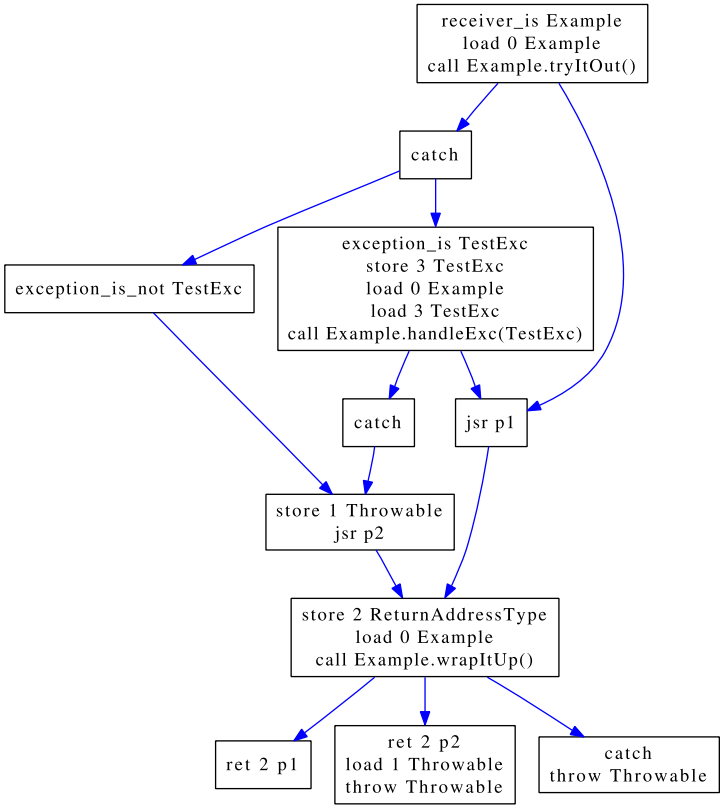

Fig. 7 The Java bytecode of the method `tryCatchFinally`

`store 2 ReturnAddressType` bytecode. Its goal is to save into local variable 2 the return address pushed on top of the stack by the `jsr p1` or by the `jsr p2` bytecodes, where `p1` and `p2` are return addresses: they can be represented in many ways in our framework, the only important requirement being that they are distinct. At the end of the subroutine, if no exception is thrown inside the subroutine, then the `ret` bytecodes select the right return point, on the basis of the fact that local variable 2 contains `p1` or `p2`. If an exception is thrown inside the `try` or `catch` blocks, then that exception is temporarily *parked* in local variable 1 and finally recovered after the subroutine terminates. Note that a unique `ret` bytecode is used in Java bytecode, while we split it in two in our representation in order to get a definition of `ret` as a state transformer. Namely, the semantics of the `jsr` and `ret` bytecodes is defined in our framework as the state transformers:

$$jsr\ p = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel p{::}s \parallel \mu\rangle$$

$$ret\ i\ p = \lambda\langle l \parallel s \parallel \mu\rangle.\begin{cases}\langle l \parallel s \parallel \mu\rangle & \text{if } l(i) = p\\ \textit{undefined} & \text{otherwise}\end{cases}$$

where, in the case of *ret*, local variable i must exist in l and have type `ReturnAddress Type`. Note that these definitions require us to modify the set of values (Definition 4) so that return addresses are values. These definitions let us apply the magic-sets transformation of

Sect. 3 to programs containing subroutines. The `jsr` and `ret` bytecodes are not special in any way. Also the operational and denotational semantics of Sects. 4 and 6 can be applied to those programs, with no change at all. The correctness result of Sect. 5 and the equivalence result of next section also hold for those programs. The proofs do not need any modification nor expansion *w.r.t.* those presented in this paper.

7 Equivalence of operational and denotational semantics

We show here that the operational semantics of Sect. 4 and the denotational semantics of Sect. 6 coincide, so that (Theorem 1) the denotation of a magic block mk is the internal behaviour at the end of block k. Namely, the main result of this section is Theorem 2.

Theorem 2 (Equivalence of the semantics) *Let b be a block (not necessarily of P) and σ_{in} an initial state for b. The functional behaviour of b, as modelled by the operational semantics of Sect. 4, coincides with its denotational semantics*:

$$\{\sigma_{out} \mid \langle b \parallel \sigma_{in}\rangle \Rightarrow_P^* \langle b' \parallel \sigma_{out}\rangle \not\Rightarrow_P\} = \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

We prove Theorem 2 by a double inclusion of its left-hand side in its right-hand side (Proposition 5) and vice versa (Proposition 6). To that purpose, we first need a lemma that states that if a block cannot be rewritten by our operational semantics then it is empty and has no successors.

Lemma 3 *Let $\langle b \parallel \sigma\rangle$ be a state such that $\langle b \parallel \sigma\rangle \not\Rightarrow$. Then b has the form* $\square^{\ell}$.

Proof The proof follows from these remarks:

- b cannot have the form $\boxed{\begin{array}{c}\mathtt{ins}_1\\ \cdots\\ \mathtt{ins}_n\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{array}$ with $n \neq 0$, otherwise one of the rules (1), (2) and (6) of Definition 9 would be applicable to $\langle b \parallel \sigma\rangle$. Note that when $\mathtt{ins}_1$ is a `call` then rule 2 is applicable since the Java bytecode is verifiable [25].
- b cannot have the form $\square^{k\rightarrow} \begin{array}{l} b_1\\ \vdots\\ b_m\end{array}$ with $m \neq 0$ and $k \in \mathbb{N}$, otherwise rule (5) of Definition 9 would be applicable to $\langle b \parallel \sigma\rangle$.
- b cannot have the form $\square^{mk\rightarrow} \begin{array}{l} b_1\\ \vdots\\ b_m\end{array}$ with $m \neq 0$ and $k \in \mathbb{N}$ since magic blocks have no successors, accordingly with Definition 1. □

Proposition 5 *Let b a block (not necessarily of P) and σ_{in} an initial state for b. Then*,

$$\{\sigma_{out} \mid \langle b \parallel \sigma_{in}\rangle \Rightarrow^* \langle b' \parallel \sigma_{out}\rangle \not\Rightarrow\} \subseteq \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

Proof For any $n \in \mathbb{N}$, block b and state σ_{in}, we let $Prop_{\subseteq}(n)$ denote the property:

if

$$\langle b \parallel \sigma_{in}\rangle \Rightarrow^n \langle b' \parallel \sigma_{out}\rangle \not\Rightarrow$$

then

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

We prove by induction on n that $Prop_{\subseteq}(n)$ holds for any $n \in \mathbb{N}$.

– (Basis) We prove that $Prop_{\subseteq}(0)$ holds. Suppose that

$$\langle b \parallel \sigma_{in} \rangle \Rightarrow^0 \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow .$$

Then, $b' = b$ and $\sigma_{out} = \sigma_{in}$. So, by Lemma 3, b has the form $\boxed{\ }^{\ell}$. Consequently, $[\![b]\!]^{\mathcal{D}_P} = \{id\}$. Hence, as $id(\sigma_{in}) = \sigma_{in}$, we have

$$\sigma_{in} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}$$

so $\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}$.

– (Induction) Suppose that for each $i \leq n$, $Prop_{\subseteq}(i)$ holds. We prove that $Prop_{\subseteq}(n+1)$ also holds. Assume that

$$\langle b \parallel \sigma_{in} \rangle \Rightarrow^{n+1} \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow .$$

Then, $\langle b \parallel \sigma_{in} \rangle \Rightarrow a \Rightarrow^n \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow$. Let us consider the rule of Definition 9 that is used in the first derivation step.

1. If rule (1) is used then

$$b = \boxed{\begin{array}{c}\texttt{ins}\\ rest\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{array} \quad \text{and} \quad a = \langle \boxed{rest}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{array} \parallel ins(\sigma_{in}) \rangle.$$

Let $b_a = \boxed{rest}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{array}$. By Definition 13,

$$[\![b]\!]^{\mathcal{D}_P} = [\![\texttt{ins}]\!]^{\mathcal{D}_P} ; [\![b_a]\!]^{\mathcal{D}_P}$$

and, as `ins` is not a `call` nor a `blockcall`, $[\![\texttt{ins}]\!]^{\mathcal{D}_P} = \{ins\}$. Therefore,

$$[\![b]\!]^{\mathcal{D}_P} = \{ins\}; [\![b_a]\!]^{\mathcal{D}_P} .$$

By inductive hypothesis,

$$\sigma_{out} \in \{\delta(ins(\sigma_{in})) \mid \delta \in [\![b_a]\!]^{\mathcal{D}_P},\ \delta(ins(\sigma_{in})) \text{ is defined}\}.$$

Then, there exists $\delta \in [\![b_a]\!]^{\mathcal{D}_P}$ such that $\delta(ins(\sigma_{in}))$ is defined and $\sigma_{out} = \delta(ins(\sigma_{in}))$. Then, $\sigma_{out} = (ins; \delta)(\sigma_{in})$ with $ins; \delta \in \{ins\}; [\![b_a]\!]^{\mathcal{D}_P}$ *i.e.*, $ins; \delta \in [\![b]\!]^{\mathcal{D}_P}$. Consequently,

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

2. If rule (2) is used then

$$b = \boxed{\begin{array}{c}\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots\\ rest\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{array}$$

and

$$a = \langle b_1 \parallel \sigma_1 \rangle :: \langle \boxed{rest}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{array} \parallel \langle l \parallel s \parallel \mu \rangle \rangle$$

where b_1 is the starting block of method $\kappa.m(\vec{\tau}) : t$, $\sigma_{in} = \langle l \parallel pars :: s \parallel \mu \rangle$ and $\sigma_1 = (makescope\ \kappa.m(\vec{\tau}) : t)(\sigma_{in})$. We consider the two possible forms of the derivation from a to $\langle b' \parallel \sigma_{out} \rangle$.

- Suppose that this derivation has the form:

$$\begin{aligned}
&\langle b_1 \parallel \sigma_1\rangle :: \langle \boxed{rest}^{\ell} \begin{matrix}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu\rangle\rangle \\
\Rightarrow^* &\langle \square^k \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle :: \langle \boxed{rest}^{\ell} \begin{matrix}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu\rangle\rangle \\
\underset{(3)}{\Rightarrow} &\langle \boxed{rest}^{\ell} \begin{matrix}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel vs :: s \parallel \mu'\rangle\rangle \\
\Rightarrow^* &\langle b' \parallel \sigma_{out}\rangle
\end{aligned}$$

where $\langle b_1 \parallel \sigma_1\rangle \Rightarrow^* \langle \square^k \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle$ in less than n steps. Notice that $\langle \square^k \parallel \langle l' \parallel vs \parallel \mu'\rangle\rangle \not\Rightarrow$. So, by inductive hypothesis,

$$\langle l' \parallel vs \parallel \mu'\rangle \in \{\delta(\sigma_1) \mid \delta \in [\![b_1]\!]^{\mathcal{D}_P},\ \delta(\sigma_1) \text{ is defined}\}.$$

Then, there exists $\delta_1 \in [\![b_1]\!]^{\mathcal{D}_P}$ such that $\delta_1(\sigma_1)$ is defined and

$$\langle l' \parallel vs \parallel \mu'\rangle = \delta_1(\sigma_1) = \delta_1((makescope\ \kappa.m(\vec{\tau}) : t)(\sigma_{in})).$$

Consequently, by Definition 11,

$$\langle l \parallel vs :: s \parallel \mu'\rangle = (extend\ \kappa.m(\vec{\tau}) : t)(\delta_1)(\sigma_{in}). \tag{9}$$

Let $b_a = \boxed{rest}^{\ell} \begin{matrix}\rightarrow b_1\\ \rightarrow \vdots\\ \rightarrow b_m\end{matrix}$. By Definition 13,

$$\begin{aligned}
[\![b]\!]^{\mathcal{D}_P} &= \left[\!\left[\texttt{call}\ \ldots \kappa.m(\vec{\tau}) : t \ldots\right]\!\right]^{\mathcal{D}_P} ; [\![b_a]\!]^{\mathcal{D}_P} \\
&\supseteq (extend\ \kappa.m(\vec{\tau}) : t)(\mathcal{D}_P(b_1)); [\![b_a]\!]^{\mathcal{D}_P}.
\end{aligned}$$

As $\langle b_a \parallel \langle l \parallel vs :: s \parallel \mu'\rangle\rangle \Rightarrow^* \langle b' \parallel \sigma_{out}\rangle$ in less than n steps, by inductive hypothesis

$$\sigma_{out} \in \left\{ \delta(\langle l \parallel vs :: s \parallel \mu'\rangle) \,\middle|\, \begin{matrix}\delta \in [\![b_a]\!]^{\mathcal{D}_P},\\ \delta(\langle l \parallel vs :: s \parallel \mu'\rangle) \text{ is defined}\end{matrix} \right\}.$$

Hence, there exists $\delta_2 \in [\![b_a]\!]^{\mathcal{D}_P}$ such that $\delta_2(\langle l \parallel vs :: s \parallel \mu'\rangle)$ is defined and $\sigma_{out} = \delta_2(\langle l \parallel vs :: s \parallel \mu'\rangle)$. So, by (9), we have

$$\begin{aligned}
\sigma_{out} &= \delta_2((extend\ \kappa.m(\vec{\tau}) : t)(\delta_1)(\sigma_{in})) \\
&= ((extend\ \kappa.m(\vec{\tau}) : t)(\delta_1); \delta_2)(\sigma_{in}).
\end{aligned}$$

Notice that, by Definition 14, $[\![b_1]\!]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(b_1)$. As $\mathcal{D}_P$ is the least fixpoint of T_P, we have $T_P(\mathcal{D}_P)(b_1) = \mathcal{D}_P(b_1)$. Hence, $[\![b_1]\!]^{\mathcal{D}_P} = \mathcal{D}_P(b_1)$ which implies, as $\delta_1 \in [\![b_1]\!]^{\mathcal{D}_P}$, that $\delta_1 \in \mathcal{D}_P(b_1)$ *i.e.*, that $(extend\ \kappa.m(\vec{\tau}) : t)(\delta_1) \in (extend\ \kappa.m(\vec{\tau}) : t)(\mathcal{D}_P(b_1))$. Therefore,

$$(extend\ \kappa.m(\vec{\tau}) : t)(\delta_1); \delta_2 \in (extend\ \kappa.m(\vec{\tau}) : t)(\mathcal{D}_P(b_1)); [\![b_a]\!]^{\mathcal{D}_P}$$

i.e., $(extend\ \kappa.m(\vec{\tau}) : t)(\delta_1); \delta_2 \in [\![b]\!]^{\mathcal{D}_P}$. Consequently,

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

- Suppose that the derivation from a to $\langle b' \parallel \sigma_{out} \rangle$ has the form:

$$\begin{aligned}
&\langle b_1 \parallel \sigma_1 \rangle :: \langle \boxed{rest}^{\ell} \begin{matrix} \rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m \end{matrix} \parallel \langle l \parallel s \parallel \mu \rangle \rangle \\
\Rightarrow^* &\langle \square^k \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle :: \langle \boxed{rest}^{\ell} \begin{matrix} \rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m \end{matrix} \parallel \langle l \parallel s \parallel \mu \rangle \rangle \\
\underset{(4)}{\Rightarrow} &\langle \boxed{rest}^{\ell} \begin{matrix} \rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m \end{matrix} \parallel \underline{\langle l \parallel e \parallel \mu' \rangle} \rangle \\
\Rightarrow^* &\langle b' \parallel \sigma_{out} \rangle
\end{aligned}$$

where $\langle b_1 \parallel \sigma_1 \rangle \Rightarrow^* \langle \square^k \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle$ in less than n steps. Notice that $\langle \square^k \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle \not\Rightarrow$. So, by inductive hypothesis,

$$\underline{\langle l' \parallel e \parallel \mu' \rangle} \in \{\delta(\sigma_1) \mid \delta \in [\![b_1]\!]^{\mathcal{D}_P},\ \delta(\sigma_1) \text{ is defined}\}.$$

Then, there exists $\delta_1 \in [\![b_1]\!]^{\mathcal{D}_P}$ such that $\delta_1(\sigma_1)$ is defined and

$$\underline{\langle l' \parallel e \parallel \mu' \rangle} = \delta_1(\sigma_1) = \delta_1((makescope\ \kappa.m(\vec{\tau}) : t)(\sigma_{in})).$$

Consequently, by Definition 11,

$$\underline{\langle l \parallel e \parallel \mu' \rangle} = (extend\ \kappa.m(\vec{\tau}) : t)(\delta_1)(\sigma_{in}). \tag{10}$$

Let $b_a = \boxed{rest}^{\ell} \begin{matrix} \rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m \end{matrix}$. By Definition 13,

$$\begin{aligned}
[\![b]\!]^{\mathcal{D}_P} &= \left[\!\!\left[\texttt{call } \ldots \kappa.m(\vec{\tau}) : t \ldots\right]\!\!\right]^{\mathcal{D}_P} ; [\![b_a]\!]^{\mathcal{D}_P} \\
&\supseteq (extend\ \kappa.m(\vec{\tau}) : t)(\mathcal{D}_P(b_1)); [\![b_a]\!]^{\mathcal{D}_P}.
\end{aligned}$$

As $\langle b_a \parallel \underline{\langle l \parallel e \parallel \mu' \rangle} \rangle \Rightarrow^* \langle b' \parallel \sigma_{out} \rangle$ in less than n steps, by inductive hypothesis

$$\sigma_{out} \in \left\{ \delta(\underline{\langle l \parallel e \parallel \mu' \rangle}) \;\middle|\; \begin{matrix} \delta \in [\![b_a]\!]^{\mathcal{D}_P}, \\ \delta(\underline{\langle l \parallel e \parallel \mu' \rangle}) \text{ is defined} \end{matrix} \right\}.$$

Hence, there exists $\delta_2 \in [\![b_a]\!]^{\mathcal{D}_P}$ such that $\delta_2(\underline{\langle l \parallel e \parallel \mu' \rangle})$ is defined and $\sigma_{out} = \delta_2(\underline{\langle l \parallel e \parallel \mu' \rangle})$. So, by (10), we have

$$\begin{aligned}
\sigma_{out} &= \delta_2((extend\ \kappa.m(\vec{\tau}) : t)(\delta_1)(\sigma_{in})) \\
&= ((extend\ \kappa.m(\vec{\tau}) : t)(\delta_1); \delta_2)(\sigma_{in}).
\end{aligned}$$

Notice that, by Definition 14, $[\![b_1]\!]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(b_1)$. As $\mathcal{D}_P$ is the least fixpoint of T_P, we have $T_P(\mathcal{D}_P)(b_1) = \mathcal{D}_P(b_1)$. Hence, $[\![b_1]\!]^{\mathcal{D}_P} = \mathcal{D}_P(b_1)$ which implies, as $\delta_1 \in [\![b_1]\!]^{\mathcal{D}_P}$, that $\delta_1 \in \mathcal{D}_P(b_1)$ *i.e.*, that $(extend\ \kappa.m(\vec{\tau}) : t)(\delta_1) \in (extend\ \kappa.m(\vec{\tau}) : t)(\mathcal{D}_P(b_1))$. Therefore,

$$(extend\ \kappa.m(\vec{\tau}) : t)(\delta_1); \delta_2 \in (extend\ \kappa.m(\vec{\tau}) : t)(\mathcal{D}_P(b_1)); [\![b_a]\!]^{\mathcal{D}_P}$$

i.e., $(extend\ \kappa.m(\vec{\tau}) : t)(\delta_1); \delta_2 \in [\![b]\!]^{\mathcal{D}_P}$. Consequently,

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

3. Rule (3) cannot be used because it requires a starting activation stack whose length is at least equal to 2. Here, the starting activation stack is $\langle b \parallel \sigma_{in} \rangle$, whose length is equal to 1.
4. Rule (4) cannot be used for the same reasons as above.
5. If rule (5) is used then

$$b = \boxed{\ }^{k \rightarrow} \begin{matrix} b_1 \\ \vdots \\ b_m \end{matrix} \quad \text{and} \quad a = \langle b_i \parallel \sigma_{in} \rangle$$

where $i \in \{1, \ldots, m\}$. Notice that, by Definition 13,

$$[\![b]\!]^{\mathcal{D}_P} = \mathcal{D}_P(b_1) \cup \cdots \cup \mathcal{D}_P(b_m).$$

Moreover, for each $j \in \{1, \ldots, m\}$, $[\![b_j]\!]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(b_j)$ by Definition 14. As $\mathcal{D}_P$ is the least fixpoint of T_P, then we have $T_P(\mathcal{D}_P)(b_j) = \mathcal{D}_P(b_j)$. Therefore, $[\![b_j]\!]^{\mathcal{D}_P} = \mathcal{D}_P(b_j)$. Consequently,

$$[\![b]\!]^{\mathcal{D}_P} = [\![b_1]\!]^{\mathcal{D}_P} \cup \cdots \cup [\![b_m]\!]^{\mathcal{D}_P}.$$

As $\langle b_i \parallel \sigma_{in} \rangle \Rightarrow^n \langle b' \parallel \sigma_{out} \rangle$, by inductive hypothesis

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b_i]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

Hence, there exists $\delta \in [\![b_i]\!]^{\mathcal{D}_P}$ such that $\delta(\sigma_{in})$ is defined and $\sigma_{out} = \delta(\sigma_{in})$. As $[\![b_i]\!]^{\mathcal{D}_P} \subseteq [\![b]\!]^{\mathcal{D}_P}$, we have $\delta \in [\![b]\!]^{\mathcal{D}_P}$. So,

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

6. If rule (6) is used then

$$b = \boxed{\begin{matrix} \texttt{blockcall}\ mp_1 \cdots mp_l \\ rest \end{matrix}}^{mk}$$

and

$$a = \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle$$

where $i \in \{1, \ldots, l\}$. The derivation from a to $\langle b' \parallel \sigma_{out} \rangle$ has the form

$$\begin{aligned} & \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle \\ \Rightarrow^* & \langle \boxed{\ }^{mp_i} \parallel \sigma \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle \\ \underset{(7)}{\Rightarrow} & \langle \boxed{rest}^{mk} \parallel \sigma \rangle \\ \Rightarrow^* & \langle b' \parallel \sigma_{out} \rangle \end{aligned}$$

where $\langle P(mp_i) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\ }^{mp_i} \parallel \sigma \rangle$ in less than n steps. Notice that $\langle \boxed{\ }^{mp_i} \parallel \sigma \rangle \not\Rightarrow$. So, by inductive hypothesis,

$$\sigma \in \{\delta(\sigma_{in}) \mid \delta \in [\![P(mp_i)]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

Then, there exists $\delta_i \in [\![P(mp_i)]\!]^{\mathcal{D}_P}$ such that $\delta_i(\sigma_{in})$ is defined and $\sigma = \delta_i(\sigma_{in})$. Let $b_a = \boxed{\mathit{rest}}^{mk}$. By Definition 13,

$$\begin{aligned}[\![b]\!]^{\mathcal{D}_P} &= \left[\!\left[\texttt{blockcall}\ mp_1 \cdots mp_l\right]\!\right]^{\mathcal{D}_P}; [\![b_a]\!]^{\mathcal{D}_P} \\ &= (\mathcal{D}_P(P(mp_1)) \cup \cdots \cup \mathcal{D}_P(P(mp_l)))\,; [\![b_a]\!]^{\mathcal{D}_P}.\end{aligned}$$

Moreover, for each $j \in \{1, \ldots, l\}$,

$$\left[\!\left[P(mp_j)\right]\!\right]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(P(mp_j))$$

by Definition 14. As $\mathcal{D}_P$ is the least fixpoint of T_P,

$$T_P(\mathcal{D}_P)(P(mp_j)) = \mathcal{D}_P(P(mp_j)).$$

Therefore, $[\![P(mp_j)]\!]^{\mathcal{D}_P} = \mathcal{D}_P(P(mp_j))$. Consequently,

$$[\![b]\!]^{\mathcal{D}_P} = ([\![P(mp_1)]\!]^{\mathcal{D}_P} \cup \cdots \cup [\![P(mp_l)]\!]^{\mathcal{D}_P}); [\![b_a]\!]^{\mathcal{D}_P}.$$

As $\langle b_a \parallel \sigma \rangle \Rightarrow^* \langle b' \parallel \sigma_{out} \rangle$ in less than n steps, by inductive hypothesis

$$\sigma_{out} \in \{\delta(\sigma) \mid \delta \in [\![b_a]\!]^{\mathcal{D}_P},\ \delta(\sigma) \text{ is defined}\}.$$

Hence, there exists $\delta' \in [\![b_a]\!]^{\mathcal{D}_P}$ such that $\delta'(\sigma)$ is defined and $\sigma_{out} = \delta'(\sigma)$ *i.e.,* $\sigma_{out} = \delta'(\delta_i(\sigma_{in})) = (\delta_i; \delta')(\sigma_{in})$. As $\delta_i \in [\![P(mp_i)]\!]^{\mathcal{D}_P}$, we have $\delta_i \in [\![P(mp_1)]\!]^{\mathcal{D}_P} \cup \cdots \cup [\![P(mp_l)]\!]^{\mathcal{D}_P}$. Moreover, $\delta' \in [\![b_a]\!]^{\mathcal{D}_P}$. Hence,

$$\delta_i; \delta' \in ([\![P(mp_1)]\!]^{\mathcal{D}_P} \cup \cdots \cup [\![P(mp_l)]\!]^{\mathcal{D}_P}); [\![b_a]\!]^{\mathcal{D}_P}$$

i.e., $\delta_i; \delta' \in [\![b]\!]^{\mathcal{D}_P}$. Therefore,

$$\sigma_{out} \in \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \text{ is defined}\}.$$

7. Rule (7) cannot be used because it requires a starting activation stack whose length is at least equal to 2. Here, the starting activation stack is $\langle b \parallel \sigma_{in} \rangle$, whose length is equal to 1. □

Proposition 6 *Let b a block (not necessarily of P) and σ_{in} an initial state for b. Then,*

$$\{\sigma_{out} \mid \langle b \parallel \sigma_{in} \rangle \Rightarrow^* \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow\} \supseteq \{\delta(\sigma_{in}) \mid \delta \in [\![b]\!]^{\mathcal{D}_P},\ \delta(\sigma_{in}) \textit{ is defined}\}.$$

Proof Notice that, by Definition 15, $\mathcal{D}_P = \bigsqcup_{i \geq 0} T_P^i$. Hence, for any $n \in \mathbb{N}$, we let $Prop_{\supseteq}(n)$ denote the property:

for every $\delta \in [\![b]\!]^{T_P^n}$ such that $\delta(\sigma_{in})$ is defined we have

$$\delta(\sigma_{in}) \in \{\sigma_{out} \mid \langle b \parallel \sigma_{in} \rangle \Rightarrow^* \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow\}.$$

We prove by induction on n that $Prop_{\supseteq}(n)$ holds for any $n \in \mathbb{N}$. Without loss of generality, suppose that b has the form $\boxed{\begin{array}{c}\texttt{ins}_1 \\ \cdots \\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1 \\ \cdots \\ \rightarrow b_m\end{array}$ with $k \geq 0$ and $m \geq 0$.

– (Basis) We prove that $Prop_{\supseteq}(0)$ holds.
  - If $m \neq 0$ or if there is an $i \in \{1, \ldots, k\}$ such that $\texttt{ins}_\texttt{i}$ is a `call` or a `blockcall`, then, as T_P^0 maps every block to $\varnothing$, we have $[\![b]\!]^{T_P^0} = \varnothing$ by Definition 13. So, $Prop_{\supseteq}(0)$ holds.
  - If $m = 0$ and, for each $i \in \{1, \ldots, k\}$, $\texttt{ins}_\texttt{i}$ is not a `call` nor a `blockcall` then, by Definition 13, we have

$$[\![b]\!]^{T_P^0} = \{ins_1\}; \cdots ; \{ins_k\} = \{ins_1; \cdots ; ins_k\}.$$

Moreover, by Definition 9,

$$\langle \underbrace{\boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_\texttt{k}\end{array}}^{\ell}}_{b} \parallel \sigma_{in}\rangle \underset{(1)}{\Rightarrow} \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_\texttt{k}\end{array}}^{\ell} \parallel ins_1(\sigma_{in})\rangle$$

$$\underset{(1)}{\Rightarrow} \langle \boxed{\begin{array}{c}\texttt{ins}_3\\ \cdots\\ \texttt{ins}_\texttt{k}\end{array}}^{\ell} \parallel ins_2(ins_1(\sigma_{in}))\rangle$$

$$\vdots$$

$$\underset{(1)}{\Rightarrow} \langle \square^{\ell} \parallel ins_k(\cdots ins_1(\sigma_{in}) \cdots)\rangle$$

$$\not\Rightarrow$$

with $ins_k(\cdots ins_1(\sigma_{in}) \cdots) = (ins_1; \cdots ; ins_k)(\sigma_{in})$. Hence, for every $\delta \in [\![b]\!]^{T_P^0}$ such that $\delta(\sigma_{in})$ is defined we have

$$\delta(\sigma_{in}) \in \{\sigma_{out} \mid \langle b \parallel \sigma_{in}\rangle \Rightarrow^* \langle b' \parallel \sigma_{out}\rangle \not\Rightarrow\}$$

i.e., $Prop_{\supseteq}(0)$ holds.

– (Induction) Suppose that $Prop_{\supseteq}(n)$ holds. We prove that then $Prop_{\supseteq}(n+1)$ also holds.

If $[\![b]\!]^{T_P^{n+1}} = \varnothing$ then $Prop_{\supseteq}(n+1)$ holds. Suppose that $[\![b]\!]^{T_P^{n+1}} \neq \varnothing$. Let $\delta \in [\![b]\!]^{T_P^{n+1}}$ such that $\delta(\sigma_{in})$ is defined. By Definition 13, we have

$$[\![b]\!]^{T_P^{n+1}} = [\![\texttt{ins}_1]\!]^{T_P^{n+1}}; \cdots ; [\![\texttt{ins}_\texttt{k}]\!]^{T_P^{n+1}}; (T_P^{n+1}(b_1) \cup \cdots \cup T_P^{n+1}(b_m)).$$

Notice that for all $i \in \{1, \ldots, m\}$, $T_P^{n+1}(b_i) = T_P(T_P^n)(b_i)$ with $T_P(T_P^n)(b_i) = [\![b_i]\!]^{T_P^n}$ by Definition 14. Hence,

$$[\![b]\!]^{T_P^{n+1}} = [\![\texttt{ins}_1]\!]^{T_P^{n+1}}; \cdots ; [\![\texttt{ins}_\texttt{k}]\!]^{T_P^{n+1}}; ([\![b_1]\!]^{T_P^n} \cup \cdots \cup [\![b_m]\!]^{T_P^n}).$$

Hence there exist $\delta_1 \in [\![\texttt{ins}_1]\!]^{T_P^{n+1}}, \ldots, \delta_k \in [\![\texttt{ins}_\texttt{k}]\!]^{T_P^{n+1}}$, $i \in \{1, \ldots, m\}$ and $\delta' \in [\![b_i]\!]^{T_P^n}$ such that

$$\delta = \delta_1; \cdots ; \delta_k; \delta'.$$

If b is not a magic block, then b does not contain any `blockcall` and either b does not contain any `call` or only the first instruction of b is a `call`. If b is a magic block, derived from a non-magic block b' accordingly to Definition 1, then we assumed that b'

can only start with a `call` when it is not the first block of a method. Hence either b does not contain any `blockcall` nor `call` (third case of Definition 1), or b starts with a `blockcall` and then consists of instructions that are not a `call` nor a `blockcall` (first and second case of Definition 1), or b starts with a `blockcall` then with a `call` and then consists of instructions that are not a `call` nor a `blockcall` (first case of Definition 1). Let us consider each of these cases.

1. Suppose that b does not contain any `blockcall` nor `call`. Then, by Definition 13,

$$\delta = ins_1; \cdots; ins_k; \delta'.$$

Moreover, by Definition 9,

$$\langle \underbrace{\boxed{\begin{array}{c}\mathtt{ins}_1\\ \cdots\\ \mathtt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \vdots\\ \rightarrow b_m\end{array}}_{b} \parallel \sigma_{in}\rangle \underset{(1)}{\Rightarrow} \langle \boxed{\begin{array}{c}\mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \vdots\\ \rightarrow b_m\end{array} \parallel ins_1(\sigma_{in})\rangle$$

$$\underset{(1)}{\Rightarrow} \langle \boxed{\begin{array}{c}\mathtt{ins}_3\\ \cdots\\ \mathtt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \vdots\\ \rightarrow b_m\end{array} \parallel ins_2(ins_1(\sigma_{in}))\rangle$$

$$\vdots$$

$$\underset{(1)}{\Rightarrow} \langle \square^{\ell} \begin{array}{l}\rightarrow b_1\\ \vdots\\ \rightarrow b_m\end{array} \parallel ins_k(\cdots ins_1(\sigma_{in})\cdots)\rangle$$

$$\underset{(5)}{\Rightarrow} \langle b_i \parallel ins_k(\cdots ins_1(\sigma_{in})\cdots)\rangle.$$

As $\delta(\sigma_{in})$ is defined,

$$(ins_1; \cdots; ins_k; \delta')(\sigma_{in}) = \delta'(ins_k(\cdots ins_1(\sigma_{in})\cdots))$$

is defined. Consequently, as $\delta' \in [\![b_i]\!]^{T_P^n}$, by induction hypothesis we have

$$\langle b_i \parallel ins_k(\cdots ins_1(\sigma_{in})\cdots)\rangle \Rightarrow^* \langle b' \parallel \delta'(ins_k(\cdots ins_1(\sigma_{in})\cdots))\rangle$$
$$\not\Rightarrow$$

So, $\langle b \parallel \sigma_{in}\rangle \Rightarrow^* \langle b' \parallel \delta(\sigma_{in})\rangle \not\Rightarrow$ *i.e.*,

$$\delta(\sigma_{in}) \in \{\sigma_{out} \mid \langle b \parallel \sigma_{in}\rangle \Rightarrow^* \langle b' \parallel \sigma_{out}\rangle \not\Rightarrow\}.$$

2. Suppose that b starts with a `call` and then consists of instructions that are not a `call` nor a `blockcall`. Then, $\mathtt{ins}_1$ has the form `call` $\kappa_1.m(\vec{\tau}) : t, \ldots, \kappa_x.m(\vec{\tau}) : t$ and, by Definition 13,

$$[\![\mathtt{ins}_1]\!]^{T_P^{n+1}} = \bigcup_{1 \le i \le x} (extend\ \kappa_i.m(\vec{\tau}) : t)(T_P^{n+1}(b_{\kappa_i.m(\vec{\tau}):t}))$$

(where $b_{\kappa_i.m(\vec{\tau}):t}$ is the block where method $\kappa_i.m(\vec{\tau}) : t$ starts) *i.e.*,

$$[\![\mathtt{ins}_1]\!]^{T_P^{n+1}} = \bigcup_{1 \le i \le x} (extend\ \kappa_i.m(\vec{\tau}) : t)(T_P(T_P^n)(b_{\kappa_i.m(\vec{\tau}):t}))$$

$$= \bigcup_{1 \le i \le x} (extend\ \kappa_i.m(\vec{\tau}) : t)([\![b_{\kappa_i.m(\vec{\tau}):t}]\!]^{T_P^n}).$$

Moreover, by Definition 9, $\sigma_{in} = \langle l \parallel pars :: s \parallel \mu \rangle$ and

$$\langle \underbrace{\boxed{\begin{matrix}\mathtt{ins}_1 \\ \cdots \\ \mathtt{ins}_k\end{matrix}}^{\ell} \begin{matrix}\rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m\end{matrix}}_{b} \parallel \sigma_{in}\rangle \underset{(2)}{\Rightarrow} \langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma' \rangle :: \langle \boxed{\begin{matrix}\mathtt{ins}_2 \\ \cdots \\ \mathtt{ins}_k\end{matrix}}^{\ell} \begin{matrix}\rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu \rangle \rangle \tag{11}$$

for some $1 \le i \le x$, where $\sigma' = (makescope\ \kappa_i.m(\vec{\tau}) : t)(\sigma_{in})$. As $\delta_1 \in [\![\mathtt{ins}_1]\!]^{T_P^{n+1}}$, there exists $\delta_1' \in [\![b_{\kappa_i.m(\vec{\tau}):t}]\!]^{T_P^n}$ such that $\delta_1 = (extend\ \kappa_i.m(\vec{\tau}) : t)(\delta_1')$. As $\delta_1(\sigma_{in})$ is defined (because $\delta(\sigma_{in})$ is defined) by Definition 11 we have two possible situations:

- In the first situation,

$$\delta_1(\sigma_{in}) = \langle l \parallel vs :: s \parallel \mu' \rangle$$

where $\langle l' \parallel vs \parallel \mu' \rangle = \delta_1'((makescope\ \kappa_i.m(\vec{\tau}) : t)(\sigma_{in})) = \delta_1'(\sigma')$. Hence, $\delta_1'(\sigma')$ is defined and, as $\delta_1' \in [\![b_{\kappa_i.m(\vec{\tau}):t}]\!]^{T_P^n}$, by inductive hypothesis $\langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma' \rangle \Rightarrow^* \langle b' \parallel \delta_1'(\sigma') \rangle \not\Rightarrow$ *i.e.*,

$$\langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma' \rangle \Rightarrow^* \langle b' \parallel \langle l' \parallel vs \parallel \mu' \rangle \rangle \not\Rightarrow .$$

Note that b' is not a magic block (because no magic block is reachable from a non-magic block of a method, like $b_{\kappa_i.m(\vec{\tau}):t}$) and that, by Lemma 3, b' has the form $\square^{k'}$. Therefore, by (11) and Definition 9, we have

$$\langle b \parallel \sigma_{in} \rangle \underset{(2)}{\Rightarrow} \langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma' \rangle :: \langle \boxed{\begin{matrix}\mathtt{ins}_2 \\ \cdots \\ \mathtt{ins}_k\end{matrix}}^{\ell} \begin{matrix}\rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu \rangle \rangle$$

$$\Rightarrow^* \langle \square^{k'} \parallel \langle l' \parallel vs \parallel \mu' \rangle \rangle :: \langle \boxed{\begin{matrix}\mathtt{ins}_2 \\ \cdots \\ \mathtt{ins}_k\end{matrix}}^{\ell} \begin{matrix}\rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu \rangle \rangle$$

$$\underset{(3)}{\Rightarrow} \langle \boxed{\begin{matrix}\mathtt{ins}_2 \\ \cdots \\ \mathtt{ins}_k\end{matrix}}^{\ell} \begin{matrix}\rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m\end{matrix} \parallel \langle l \parallel vs :: s \parallel \mu' \rangle \rangle$$

i.e., $\langle b \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\begin{matrix}\mathtt{ins}_2 \\ \cdots \\ \mathtt{ins}_k\end{matrix}}^{\ell} \begin{matrix}\rightarrow b_1 \\ \rightarrow \vdots \\ \rightarrow b_m\end{matrix} \parallel \delta_1(\sigma_{in}) \rangle$. Then, proceeding as in case 1 above, we prove that

$$\delta(\sigma_{in}) \in \{\sigma_{out} \mid \langle b \parallel \sigma_{in} \rangle \Rightarrow^* \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow\}.$$

- In the second situation,

$$\delta_1(\sigma_{in}) = \underline{\langle l \parallel e \parallel \mu' \rangle}$$

where $\underline{\langle l' \parallel e \parallel \mu' \rangle} = \delta_1'((makescope\ \kappa_i.m(\vec{\tau}) : t)(\sigma_{in})) = \delta_1'(\sigma')$. Hence, $\delta_1'(\sigma')$ is defined and, as $\delta_1' \in [\![b_{\kappa_i.m(\vec{\tau}):t}]\!]^{T_P^n}$, by inductive hypothesis $\langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma' \rangle \Rightarrow^* \langle b' \parallel \delta_1'(\sigma') \rangle \not\Rightarrow$ *i.e.*,

$$\langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma' \rangle \Rightarrow^* \langle b' \parallel \underline{\langle l' \parallel e \parallel \mu' \rangle} \rangle \not\Rightarrow .$$

Note that b' is not a magic block (because no magic block is reachable from a non-magic block of a method, like $b_{\kappa_i.m(\vec{\tau}):t}$) and that, by Lemma 3, b' has the form $\square^{k'}$. Therefore, by (11) and Definition 9, we have

$$\langle b \parallel \sigma_{in}\rangle \underset{(2)}{\Rightarrow} \langle b_{\kappa_i.m(\vec{\tau}):t} \parallel \sigma'\rangle :: \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

$$\Rightarrow^* \langle \square^{k'} \parallel \underline{\langle l' \parallel e \parallel \mu'\rangle}\rangle :: \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

$$\underset{(4)}{\Rightarrow} \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array} \parallel \underline{\langle l \parallel e \parallel \mu'\rangle}\rangle$$

i.e., $\langle b \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array} \parallel \delta_1(\sigma_{in})\rangle$. Then, proceeding as in case 1 above, we prove that

$$\delta(\sigma_{in}) \in \{\sigma_{out} \mid \langle b \parallel \sigma_{in}\rangle \Rightarrow^* \langle b' \parallel \sigma_{out}\rangle \not\Rightarrow\}.$$

3. Suppose that b starts with a `blockcall` and then consists of instructions that are not a `call` nor a `blockcall`. Then, $\texttt{ins}_1$ has the form `blockcall` $mp_1 \cdots mp_l$ and, by Definition 13,

$$\begin{aligned}[[\texttt{ins}_1]]^{T_P^{n+1}} &= T_P^{n+1}(P(mp_1)) \cup \cdots \cup T_P^{n+1}(P(mp_l))\\ &= T_P(T_P^n)(P(mp_1)) \cup \cdots \cup T_P(T_P^n)(P(mp_l))\\ &= [[P(mp_1)]]^{T_P^n} \cup \cdots \cup [[P(mp_l)]]^{T_P^n}.\end{aligned}$$

As $\delta_1 \in [[\texttt{ins}_1]]^{T_P^{n+1}}$, there is an $i \in \{1, \ldots, l\}$ such that $\delta_1 \in [[P(mp_i)]]^{T_P^n}$. By Definition 9, we have

$$\langle \underbrace{\boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array}}_{b} \parallel \sigma_{in}\rangle \underset{(6)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in}\rangle :: \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array} \parallel \sigma_{in}\rangle. \tag{12}$$

As $\delta_1(\sigma_{in})$ is defined (because $\delta(\sigma_{in})$ is defined) and $\delta_1 \in [[P(mp_i)]]^{T_P^n}$, by inductive hypothesis we have

$$\langle P(mp_i) \parallel \sigma_{in}\rangle \Rightarrow^* \langle b' \parallel \delta_1(\sigma_{in})\rangle \not\Rightarrow .$$

By the rules of Definition 9, b' is a magic block labelled with mp_i and, by Lemma 3, b' has the form $\square^{mp_i}$. Therefore, by (12) and Definition 9, we have

$$\langle b \parallel \sigma_{in}\rangle \underset{(6)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in}\rangle :: \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \cdots\\ \rightarrow b_m\end{array} \parallel \sigma_{in}\rangle$$

$$\Rightarrow^* \langle \square^{mp_i} \parallel \delta_1(\sigma_{in}) \rangle :: \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \dot{b}_m\end{array} \parallel \sigma_{in} \rangle$$

$$\underset{(7)}{\Rightarrow} \langle \boxed{\begin{array}{c}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{array}}^{\ell} \begin{array}{l}\rightarrow b_1\\ \rightarrow \dot{b}_m\end{array} \parallel \delta_1(\sigma_{in}) \rangle .$$

Then, proceeding as in case 1 above, we prove that

$$\delta(\sigma_{in}) \in \{\sigma_{out} \mid \langle b \parallel \sigma_{in} \rangle \Rightarrow^* \langle b' \parallel \sigma_{out} \rangle \not\Rightarrow\}.$$

4. Suppose that b starts with a `blockcall` then with a `call` and then consists of instructions that are not a `call` nor a `blockcall`. This case is a combination of the three above. In order to conclude, one has first to reason as in case 3, then as in case 2 and finally as in case 1. □

The conclusion of this section is that we can apply the magic-set transformation to a program, compute the denotational semantics of the transformed program and get information at internal program points by looking at the denotations of the magic blocks.

8 Abstraction of the denotational semantics

This paper provides a semantical basis for the analysis of program properties at internal program points through denotational semantics. The abstraction of this denotational semantics is completely identical to well-known abstractions of denotational semantics, such as that described in [17]. Nevertheless, in this section we show an example of abstract denotational semantics, that should help understanding our denotational analyses. It is meant to track the program variables that hold `null`. We do not provide formal proofs of correctness here, in terms of abstraction and concretisation maps, that are outside the scope of this paper. Moreover, we do not discuss how exceptions can be taken into account in this abstraction and how fields can receive a non-trivial approximation. All this material, including proofs, can be found in [41] and [39].

The first step for defining an abstract semantics through abstract interpretation is the definition of the abstract domain. Since the concrete denotational semantics of Sect. 6 works over denotations, the abstract domain abstracts denotations. Since such denotations are maps from input to output states and we want to track the nullness of reference variables, also the abstract domain will naturally distinguish between nullness of variables in the input and in the normal (non-exceptional) output. Thus we write $\check{v}$ to express the fact that variable v holds `null` in the input and we write $\hat{v}$ to express the fact that variable v holds `null` in the normal output. We discuss Java bytecode and hence by *variable* we mean both a local variable l_i and a stack element s_i, where i is the index of the local variable or the height of the stack element from the bottom s_0. The abstract domain $\mathbb{N}$ for nullness analysis [39, 41] is made of Boolean formulas over $\check{v}$ and $\hat{v}$, for every variable v which is in scope at a given program point.

Consider for instance the denotations of every single bytecode, given in Sect. 4. The denotation *const c* pushes a constant c on top of the stack; all local variables $l_0, \ldots, l_k$ and all stack elements $s_h :: \cdots :: s_0$ of the input state keep their value unchanged. Note that k and h are fixed and statically known for each instance of a bytecode in a program, as the Java

Virtual Machine specification requires. As a consequence, the following Boolean formula is true *w.r.t.* nullness:

$$(\check{l}_0 \iff \hat{l}_0) \wedge \cdots \wedge (\check{l}_k \iff \hat{l}_k) \wedge (\check{s}_0 \iff \hat{s}_0) \wedge \cdots \wedge (\check{s}_h \iff \hat{s}_h). \tag{13}$$

This formula says that each local variable or stack element in the input state holds `null` if and only if the same local variable or stack element holds `null` in the normal output state. The formula (13) is known as *frame condition*. From now on, we will write U (as *unchanged*) for (13) and assume that U predicates over the variables of the input which are never modified by the bytecode nor disappear after its execution. By using this notation, we can provide an approximation of the *const c* denotation:

$$const^{\mathbb{N}}\ c = \begin{cases} U \wedge \hat{s}_{h+1} & \text{if } c = \texttt{null} \\ U \wedge \neg\hat{s}_{h+1} & \text{if } c \neq \texttt{null}. \end{cases}$$

In words, the original variables do not change their nullness, and a new variable s_{h+1} is pushed on top of the output stack, holding `null` if and only if $c = \texttt{null}$.

We can continue this way and define a Boolean formula for each denotation of a single bytecode. For instance, the *dup t* denotation duplicates the topmost value s_h of the input stack. No local variable changes and no original stack variable changes nor disappear. Hence we define

$$dup^{\mathbb{N}}\ t = U \wedge (\check{s}_h \iff \hat{s}_{h+1}).$$

This formula means that no local variable and no stack element changes and the new top of the stack s_{h+1} holds `null` in the output if and only if the old top of the stack s_h holds `null` in the input. This is true since *dup t* copies the old top of the stack over itself, thus duplicating it.

Consider the denotation *load i t*. It pushes the value of the *i*th local variable on top of the output stack. No input local variable nor stack element changes or disappears. Hence we define

$$load^{\mathbb{N}}\ i\ t = U \wedge (\check{l}_i \iff \hat{s}_{h+1}).$$

The conditional bytecode *if_ne t* asserts that the value on top of the input stack was non-`null`, since otherwise the concrete bytecode would be undefined (Sect. 4) and there would not be any normal output:

$$if\_ne^{\mathbb{N}}\ t = U \wedge \neg\check{s}_h.$$

Moreover, the top of the input stack is lost, which means that variable s_h is not considered in the formula U above.

The denotation *new* κ pushes on top of the stack a reference to a newly created object. That reference is by definition non-`null`. Hence we define

$$new^{\mathbb{N}}\ \kappa = U \wedge \neg\hat{s}_{h+1}.$$

Note the importance of considering only normal output states in the abstraction: if the `new` bytecode threw an exception, there would not be any s_{h+1} element in the exceptional output state. See [39, 41] for an abstraction that considers exceptional output states also.

The denotation *getfield* $\kappa.f{:}t$ replaces the top of the input stack s_h, that must be `null` or a reference to an object o, with the value of the field of o named f, having type t and defined

in class κ. Since we are using an abstract domain that only tracks nullness of variables, we do not know anything about the value of field f (fields are not variables in the frame of the Java Virtual Machine, they are instance variables inside each given object). The only conservative assumption is to leave the nullness of the new top of the stack undefined (see [41] and [39] for a better solution):

$$\mathit{getfield}\ \kappa.f{:}t^{\mathbb{N}} = U \wedge \neg\check{s}_h.$$

In the formula above, U does not consider the top of the input stack, since it is modified by the bytecode. Nothing is said about the top of the output stack. However, the formula says that the top of the input stack is definitely non-`null` *in order for an output state to exist*. This is because our formulas express properties of input/output pairs assuming that the output is a normal state, so that an output state is required to exist and hence we only consider pairs where the top of the input stack is non-`null`.

The denotational semantics of Sect. 6 uses a $\cup$ operation over sets of denotations and an *extend* operation for plugging the denotations of a callee at the calling place(s). In order to define an abstract semantics, we need their abstract counterparts. For $\cup$, it is always safe to use $\vee$ *i.e.,* logical disjunction. Assume now that the formula ϕ abstracts the denotations of a callee $\kappa.m(\vec{\tau}) : t$ and let p be the number of stack elements required to hold the parameters of that method, including the implicit `this` parameter. Then a safe abstraction, over $\mathbb{N}$, of *extend* $\kappa.m(\vec{\tau}) : t$ is

$$U \wedge \exists_{\{\hat{l}_0,\ldots,\hat{l}_r\}}(\phi[\check{l}_{p-1} \mapsto \check{s}_h, \ldots, \check{l}_0 \mapsto \check{s}_{h-p+1}, \hat{s}_0 \mapsto \hat{s}_{h-p+1}]).$$

That is, the input local variables of the callee are matched to the highest input stack elements of the caller and the return value of the callee, left as the only element on the stack of the output, is mapped to the top of the output stack of the caller. The output local variables of the callee are removed since they disappear at the end of the execution of the callee. The $\exists$ operation removes them and is implemented as *Schröder elimination* [3]. A similar renaming allows one to define a correct abstraction for *makescope* also.

The last operation used in Sect. 6 is the sequential composition of denotations ;. In the abstract semantics, the sequential composition of two formulas ϕ_1 and ϕ_2 is defined by renaming the output variables of ϕ_1 and the input variables of ϕ_2 into the same new set of fresh variables; the renamed formulas are then conjuncted (through $\wedge$) and the new fresh variables are existentially quantified through Schröder elimination (hence removed from the resulting Boolean formula). See [41] and [39] for details.

We have now abstractions for each single bytecode and for the semantical operators used in Sect. 6. Hence the construction of the abstract semantics proceeds by Definitions 12, 13 and 14, except that we use abstractions now instead of concrete bytecodes and of concrete operators. The abstract semantics is computed as a fixpoint, by following Definition 15. For better efficiency, that fixpoint is computed by following, backwards, the strongly-connected components of blocks. No widening is used for this computation, but other abstract domains will also require the definition of a widening operator [15].

Note that this semantics is only defined in terms of a single abstract domain of Boolean formulas. All the required operations over Boolean formulas can be efficiently implemented in terms of binary decision diagrams [10].

9 Experiments

We have implemented our magic-sets transformation inside the generic analyser JULIA for Java bytecode [47] and used it with some abstract domains. JULIA is a bottom-up analyser

that follows the strongly-connected components of blocks backwards and computes local fixpoints for each component. For better efficiency, it caches the analysis of each bytecode so that, if it is needed twice, it is computed only once. This happens frequently with our magic-sets transformation, that introduces code duplication. For instance, block $m16$ in Fig. 6 shares three bytecodes which block 16. This technique has been inspired by a similar optimisation of the analysis of *magic logic programs*, defined in [12]. Since it caches the *functional* behaviour of the code, it is different from *memoisation*, that only caches its behaviour for *each given* input state. The development of an abstract domain for JULIA requires only to provide the abstraction of each single bytecode, abstract sequential composition, abstract union and abstract *extend* and *makescope* (as those that we have defined in Sect. 8 for the case of nullness analysis). The definition of a widening operator [15] is also mandatory if the abstract domain does not satisfy the ascending chain condition (this is the case of size analysis, for instance).

The abstract domain that we consider in these experiments are the following:

1. The first domain [38] over-approximates the set of pairs of program variables, which for the Java bytecode means local variables or stack elements, that *share i.e.,* reach the same memory location; it is used for automatic program parallelisation and to support other analyses;
2. The second [35] over-approximates the set of *cyclical* program variables, those that reach a loop of memory locations; it needs a preliminary pair-sharing analysis;
3. The third, which comes in two versions of different precision [39, 41], over-approximates the set of program variables that can actually hold `null` at run-time. The other variables are, hence, definitely non-`null`;
4. The fourth under-approximates the set of classes that are definitely initialised at a given program point. As a consequence, subsequent initialisation tests over those classes are useless and static reference to those classes will not call their static initialiser [25]. This analysis is important to simplify the control-flow of the program before other analyses are applied;
5. The fifth is a kind of *size* analysis, called *path-length*, that provides a linear approximation of the size of the variables at a given program point. It is defined in [42, 43] and is inspired by the domain in [18]. Compared to that work, our domain also approximates the values of variables of reference type with the length of the longest path of pointers that can be followed from them. It also provides approximations for a few fields, deemed relevant for the analysis. It is used for proving termination of programs.

The first three domains are implemented with Boolean formulas to abstract sets of denotations by relating properties of their input to properties of their output, as shown in Sect. 8. For instance, $(l1, s1) \Rightarrow (l1, l2)$ abstracts those denotations δ such that for every state σ, where only local variable 1 and stack element 1 might share (the base of the stack is $s0$), we have that in $\delta(\sigma)$ only local variables 1 and 2 might share (for simplicity, we do not report variables sharing with themselves [38]). We have implemented Boolean formulas through binary decision diagrams [10]. The fourth domain is implemented through bitsets that specify which classes are definitely initialised during the execution of a piece of code. The fifth domain is implemented with linear numerical constraints, represented through polyhedra of the native C++ Parma Polyhedra Library [4]. Since polyhedra have expensive operations, we keep them in a simpler form, as zones [30] represented in Java, whenever this is possible without any loss of precision. This *hybrid* approach (polyhedra and zones, together) yields a static analysis that is as precise as a purely polyhedral one but still relatively fast. We have also another implementation of path-length, where only zones are used, that is much faster and only slightly less precise.

Our experiments have been performed on a quad-core Intel Xeon machine running at 2.66 GHz, with 4 gigabytes of RAM, Linux 2.6.27 and Sun jdk 1.6.

Let us consider pair-sharing. JULIA computes a formula containing the conjunct $(l1, s0)$ as abstract denotation for the magic block corresponding to the internal program point just after the `l.clone()` call in method `main` of Fig. 1. This means that $(l1, s0)$ is true just after that call, that is, local variable 1, that holds the list `l` of Fig. 1, might share with stack element 0 (the base of the stack), that holds the return value of `clone`. JULIA computes a formula where $s0$ only occurs in pair with itself as abstract denotation for the magic block corresponding to the internal program point just after the `l.deepClone()` call. Hence it proves that no local variable can share there with the return value of that call, held in the stack before being written into `l2`. Let us consider cyclicity analysis. JULIA computes *false* as abstract denotation for the same magic blocks as above *i.e.,* it proves that no local variable and stack element might be cyclical after the calls `l.clone()` and `l.deepClone()`, which is an optimal approximation of the actual behaviour of the program. Let us consider nullness analysis now. JULIA computes a formula containing the conjunct $\neg s2$ as abstract denotation for the blocks that precede, immediately, the calls to `head.copy()` in Fig. 1. This means that the top value of the stack just before those calls (*i.e.,* their receiver) is never `null`. A recent evolution of the domain in [41], defined in [39], computes a formula containing also the conjunct $\neg s3$ as abstract denotation for the block that precedes, immediately, the recursive call to `tail.clone()` in the same figure. This means that the receiver of that call is never `null`. In conclusion, JULIA proves points 1, 2 and 3 of Sect. 1.

Figure 1 shows a simple program. More complex benchmarks such as those in Fig. 8 challenge the scalability, the efficiency and the precision of the analyses. The first 4 benchmarks, which are the smallest, have been also analysed with the pair-sharing analyser in [29] so we can build a comparison. The others are progressively larger to check the scalability of the analyses. Figure 8 reports their size (number of methods), their preprocessing time with JULIA (extraction and parsing of the `.class` files, building a high-level representation of the bytecode and the magic-sets) and the percentage of the latter time due to the magic-sets transformation, which is never more than 6.63%. Only `java.lang.` and `java.util.` library classes are included in the analysis: calls to the missing classes use a worst-case assumption. We have computed the average number of dependencies between magic-blocks, that is, how many blocks are `call`ed or `blockcall`ed by a given block, on the average. This information is important since the analysis of a block depends on that of its dependencies (Definition 13), so that a high level of dependencies might make the cost of the static analyses explode. The result is an average of just 2.2 dependencies per block.

Figure 9 reports the results of pair-sharing and cyclicity analyses with JULIA. Precision, for sharing analysis, is the percentage of pairs of distinct local variables or stack elements that are proved not to share, definitely, before a `putfield`, an `arraystore` or a `call`. Only `putfield`'s, `arraystore`'s and `call`'s are considered, since it is there where sharing analysis helps other analyses (see for instance [35] for its help to cyclicity analysis). Precision, for cyclicity analysis, is the percentage of local variables or stack elements that are proved to be non-cyclical, definitely, before a `getfield` bytecode. This information is only computed at `getfield`'s that access a field or reference type, since cyclicity information is typically used there, for instance to prove termination of iterations over dynamic data-structures [42, 43].

We are not aware of any other cyclicity analysis for Java bytecode. An operational pair-sharing analyser was instead applied in [29] to the smallest 4 benchmarks in Fig. 8, without including the library classes, but we could not use their analyser. It takes time $P + T + A$: P is the preprocessing time, that they do not report. Since they use the generic analyser

	methods	preproc.	magic-sets	dependencies
Qsort	193	0.15	7.14%	2.05
IntegerQsort	194	0.19	6.84%	2.06
Passau	192	0.19	5.05%	2.04
ZipVector	198	0.22	4.54%	2.07
JLex	502	1.39	5.96%	2.22
JavaCup	1029	4.62	6.44%	2.28
Jess	3432	12.19	6.63%	2.47
jEdit	4710	20.39	4.67%	2.32
Julia	6671	21.93	4.40%	2.36

Fig. 8 Size and preprocessing times (in seconds) for our benchmarks. Only `java.lang.*` and `java.util.*` library classes are included. *Dependencies* is the average number of blocks that each block `calls` or `blockcalls`

	sharing analysis		**cyclicity analysis**	
	time	precision	time	precision
Qsort	0.24	70.24%	0.08	10.71%
IntegerQsort	0.23	73.87%	0.17	17.18%
Passau	0.19	42.13%	0.08	100.00%
ZipVector	0.32	54.67%	0.09	7.69%
JLex	1.006	30.34%	1.23	33.46%
JavaCup	5.78	53.70%	1.23	33.46%
Jess	72.99	38.60%	4.11	30.42%
jEdit	108.83	40.49%	7.99	20.22%
Julia	104.45	23.86%	7.08	14.31%

Fig. 9 Time (in seconds) and precision of our sharing and cyclicity analyses. Only `java.lang.*` and `java.util.*` library classes are included

Fig. 10 Time (in seconds) of our pair-sharing analysis and of that in [29]

	JULIA	[29]
Qsort	0.33	≥1.02
IntegerQsort	0.40	≥0.82
Passau	0.33	≥1.00
ZipVector	0.54	≥1.73

SOOT, we could compute P with SOOT version 2.3.0; T is the time to transform the output of SOOT into the format required in [29]. We cannot estimate T without the analyser; A is the *preliminary running time* reported in [29], normalised *w.r.t.* the relative speeds of our machine and theirs. Figure 10 compares the running time of JULIA, including preprocessing and without analysing the libraries, with $P + A$, since T is unknown. JULIA is faster, even without knowing T. Exceptions, subroutines, static initialisers and native methods are not tested by such small benchmarks, so it is not clear if the analyser in [29] is ready for *real* analyses. Precision is expressed as a *level of multivariance*, that we cannot translate into our more natural notion. Another analysis for (definite) sharing is implemented in [34] for a *subset* of Java. Times and precision are not reported. The code is not publicly available.

The CIBAI tool [26] is able to derive class invariants from Java *source code* rather than from bytecode. It currently includes an abstract domain tracking *abstract locations*, that

	The **nullness analysis** in [20]				
program	time	fs	get's	put's	calls
JLex	3.52	44	71.69%	64.18%	48.32%
CaffeineMark	3.80	1	98.55%	100%	63.08%
JavaCup	5.52	31	47.44%	96.04%	86.17%
JavaCC	8.07	58	92.55%	95.80%	71.58%
Kitten	-	-	-	-	-
Jess	-	-	-	-	-
Julia	-	-	-	-	-
	Our nullness analysis with the domain in [41]				
program	time	fs	get's	put's	calls
JLex	2.22	50	71.07%	71.76%	54.01%
CaffeineMark	2.35	3	100%	100%	100%
JavaCup	4.43	45	47.78%	96.04%	93.92%
JavaCC	19.79	161	92.14%	96.16%	76.78%
Kitten	6.71	85	55.24%	98.52%	90.13%
Jess	24.70	97	97.60%	99.58%	76.38%
Julia	311.13	894	88.25%	97.94%	87.04%
	Our nullness analysis with the domain in [39]				
program	time	fs	get's	put's	calls
JLex	4.89	50	90.09%	88.47%	58.82%
CaffeineMark	3.33	3	100%	100%	100%
JavaCup	10.25	45	48.52%	98.30%	94.17%
JavaCC	31.09	162	94.51%	98.26%	77.68%
Kitten	18.46	87	55.53%	99.41%	91.64%
Jess	102.19	99	97.73%	100.00%	79.11%
Julia	-	-	-	-	-

Fig. 11 Number M of methods, time (in seconds), number fs of instance fields of reference type proved non-null and amount of getfields, putfields and calls proved safe. Only java.lang.* and java.util.* libraries are included. Dereferences and fields in the library code are not counted

should provide some sharing information, although this is not detailed in [26]. No precision about sharing analysis is reported by the tool. It has been applied to programs of a few hundreds methods only. The tool is not freely available on the net so we could not build a comparison.

We have compared our nullness analysis, using the abstract domain in [41], with the implementation NIT of [20].[1] The results are in Fig. 11. Times are global (preprocessing plus analysis). Our analysis, coded in Java, performs similarly to NIT, that is natively compiled OCaml code. The latter did not manage to analyse Kitten, Jess and Julia and signalled some error. NIT is definitely faster on one example, JavaCC, but the results of its analysis are unusual, since it reports a total number of fields (105) that is actually half of those defined in the program analysed. Our analysis always finds more non-null fields than NIT. Another interesting comparison is the amount of getfields, putfields and instance calls that are proved safe. Figure 11 shows that JULIA is, on the average, more precise than NIT, with

[1] We thank Laurent Hubert for his help to understand the results of NIT.

the only exception of `JLex` and `JavaCC` again, *w.r.t.* `getfield`'s. Figure 11 also shows more precise nullness analyses performed with JULIA and the abstract domain in [39], that is able to spot locally non-`null` fields. The results are more precise, but the analysis more expensive. This version of the analysis did not manage to analyse `Julia` since it exhausted the available memory.

Figure 12 shows the results of our experiments with the abstract domains for class initialisation and for path-length. Only the `java.lang.*` libraries are included. This is particularly critical for the path-length, that is based on polyhedra and has, hence, a very high computational cost. Nevertheless, we manage to analyse up to 1300 methods in a couple of minutes. For class initialition, precision is given as the number of class initialisation tests that are proved useless by the analysis. We do not count initialisation tests against the same class where the test occurs, or against one of its superclasses, since they are always useless [25], nor against `java.lang.String`, that is always initialised when a Java program starts. For path-length, precision is given as the *termination ratio i.e.,* the amount of loops proved to terminate over the total number of loops in the program. For information on how termination is proved from path-length approximations, see [42, 43]. Note that the time to prove the termination after the path-length analysis is completed (that is relatively high) is not reported in this figure, since it is not part of the path-length analysis. Some data are missing in Fig. 12: when the time for the path-length analysis with polyhedra is missing, this means that the analysis runs out of memory; when the precision is missing, it means that the termination prover (a separate component, that is not part of the path-length analysis) runs out of memory.

These last two examples show that we can deal with a case of analysis that cannot be done with static single assignment transformations (class initialisation) and another that is usually considered very expensive (polyhedral approximations), thus validating the strength of our framework of analysis. Finally, the last two columns in Fig. 12 reports results about the path-length analysis performed with *zones* [30] only instead of polyhedra and zones together. Zones are more efficient than polyhedra but relatively less precise. For Java bytecode, it might be the case that a complex Java instruction gets compiled into many bytecodes, so that one of them does not have a precise approximation through zones although the original instruction has one. This has been already observed in [28]. Consequently, Fig. 12 shows that this implementation of path-length through zones is sometimes slightly less precise than that using polyhedra, at least for termination analysis. The last columns of Fig. 12 can be compared with the results reported in [7] about the numerical analysis performed by the CLOUSOT analyser with pentagons over a similar hardware. The average time for the analysis of a method was 21 milliseconds in [7] (normalised *w.r.t.* the different hardware) while JULIA requires 23 milliseconds per method (with zones). Since the benchmarks and the language are different, this cannot be considered as a general rule. Namely, it must be observed that this comparison is not completely fair since CLOUSOT works over `.NET` code while JULIA works over Java bytecode, so we cannot analyse the same benchmarks with those two tools. Moreover, CLOUSOT is natively-compiled C# code while JULIA is slower Java; pentagons are less precise and probably more efficient than zones; the analysis in CLOUSOT approximates numerical variables only and is not always correct since it uses a possibly incorrect heap abstraction, while our path-length analysis approximates also variables of reference type, is always correct (as proved in [43]) and it also approximates a few fields that might be important for proving the termination of the program. The goals of those two analyses are also different (checking array bounds for CLOUSOT, proving termination for JULIA) so a comparison *w.r.t.* precision is impossible. Nevertheless, our experiments show that our analyser scales similarly to a state-of-art analyser for `.NET` code.

		class init. analysis		path-length analysis			
				polyhedra and zones		zones only	
program	*M*	time	precision	time	precision	time	precision
`JLex`	376	0.27	71.35%	11.37	49.38%	4.33	49.38%
`JavaCup`	534	0.99	71.40%	17.21	42.20%	5.24	42.20%
`Kitten`	1049	1.34	45.27%	24.62	61.40%	5.35	58.77%
`JavaCC`	1300	1.84	69.42%	103.40	-	26.65	-
`Jess`	2286	4.08	20.91%	-	-	81.31	-
`jEdit`	3282	4.42	36.85%	-	-	93.41	-
`Julia`	5409	8.58	46.97%	-	-	107.32	-

Fig. 12 Number *M* of methods, time (in seconds) and precision of our class initialisation and path-length analyses, performed with polyhedra and with zones. Only `java.lang.*` library classes are included

The analyses discussed in this section are part of the JULIA analyser, that can be used through the web interface at the address http://julia.scienze.univr.it. Sharing, cyclicity, path-length with polyhedra and class initialisation analyses are run by a *termination analysis with polyhedra* of a program. Path-length analysis with zones is run by a *termination analysis with zones only* of a program. Nullness analysis is run by a *simple nullness analysis* of a program. The more precise nullness analysis in [39] is run by a *medium nullness analysis*. Times reported by that web interface do not include network delays but might be affected by the load of the remote machine.

10 Conclusion

Our magic-sets transformation is completely independent from the abstract domains, that can be developed without even knowing its existence. Then all abstract domains defined so far for the analysis of Java bytecode can in principle be used in our framework. The domain developer must only specify the internal program points where he wants to observe the results of the analysis, depending on the specific goal for which he develops the abstract domain. Our experiments show that denotational analyses of Java bytecode, with a preliminary magic-sets transformation, are feasible, fast and compare well with other analyses.

Acknowledgements The authors are grateful to the anonymous reviewers for detailed and helpful comments on preliminary versions of this article, and to Andy King for his editorship.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles Techniques and Tools. Addison-Wesley, Reading (1986)
2. Albert, E., Arenas, P., Codish, C., Genaim, S., Puebla, G., Zanardini, D.: Termination analysis of Java bytecode. In: Barthe, G., de Boer, F.S. (eds.) Proc. of Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS'08, Oslo, Norway, June 2008. Lecture Notes in Computer Science, vol. 5051, pp. 2–18. Springer, Berlin (2008)
3. Armstrong, T., Marriott, K., Schachte, P., Søndergaard, H.: Two classes of Boolean functions for dependency analysis. Sci. Comput. Program. **31**(1), 3–45 (1998)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)

5. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic sets and other strange ways to implement logic programs. In: Proc. of the 5th ACM Symposium on Principles of Database Systems, pp. 1–15 (1986)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Boogie, K.R.M. Leino: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Proc. of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05), Amsterdam, The Netherlands, November 2005. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer, Berlin (2005)
7. Barnett, M., Fahndrich, M., Logozzo, F.: Foxtrot and Clousot: Language agnostic dynamic and static contract checking for .NET. Technical Report MSR-TR-2008-105, Microsoft Research (August 2008)
8. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program. **10**(3 & 4), 255–300 (1991)
9. Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for secrecy and non-interference in networks of processes. In: Proc. of PaCT'01, Lecture Notes in Computer Science, vol. 2127, pp. 27–41. Springer, Berlin (2001)
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
11. Clark, D., Hankin, C., Hunt, S.: Information flow for ALGOL-like languages. Comput. Lang. **28**(1), 3–28 (2002)
12. Codish, M.: Efficient goal directed bottom-up evaluation of logic programs. J. Log. Program. **38**(3), 355–370 (1999)
13. Codish, M., Dams, D., Yardeni, E.: Bottom-up abstract interpretation of logic programs. J. Theor. Comput. Sci. **124**, 93–125 (1994)
14. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Schwartzbach, M.I., Ball, T. (eds.) Proc. of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06), Ottawa, Ontario, Canada, June 2006, pp. 415–426. ACM, New York (2006)
15. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), pp. 238–252 (1977)
16. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of the 6th ACM Symposium on Principles of Programming Languages (POPL'79), pp. 269–282 (1979)
17. Cousot, P., Cousot, R.: Abstract interpretation and applications to logic programs. J. Log. Program. **13**(2 & 3), 103–179 (1992)
18. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. Fifth ACM Symp. Principles of Programming Languages, pp. 84–96 (1978)
19. Danvy, O., Filinski, A.: Representing control, a study of the CPS transformation. Math. Struct. Comput. Sci. **2**(4), 361–391 (1992)
20. Hubert, L., Jensen, T., Pichardie, D.: Semantic foundations and inference of non-null annotations. In: Barthe, G., de Boer, F.S. (eds.) Proc. of the 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08), Oslo, Norway, June 2008. Lecture Notes in Computer Science, pp. 142–149. Springer, Berlin (2008)
21. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. (TOPLAS) **28**(4), 619–695 (2006)
22. Laud, P.: Semantics and program analysis of computationally secure information flow. In: Proc. of the 10th European Symposium On Programming (ESOP'01). Lecture Notes in Computer Science, vol. 2028, pp. 77–91. Springer, Berlin (2001)
23. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) Proc. of the 18th European Conference on Object-Oriented Programming (ECOOP'04), Oslo, Norway, June 2004. Lecture Notes in Computer Science, vol. 3086, pp. 491–516. Springer, Berlin (2004)
24. Leino, K.R.M., Wallenburg, A.: Class-local object invariants. In: Proc. of the 1st India Software Engineering Conference (ISEC'08), Hyderabad, India, February 2008, pp. 57–66. ACM, New York (2008)
25. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
26. Logozzo, F.: Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In: Cook, B., Podelski, A. (eds.) 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'07), Nice, France, January 2007. Lecture Notes in Computer Science, vol. 4349, pp. 293–298. Springer, Berlin (2007)
27. Logozzo, F.: Class invariants as abstract interpretation of trace semantics. Comput. Lang. Syst. Struct. **35**(2), 100–142 (2009)
28. Logozzo, F., Fähndrich, M.: On the relative completeness of bytecode analysis versus source code analysis. In: Hendren, L.J. (ed.) Proc. of the 17th International Conference on Compiler Construction, (CC'08), Budapest, Hungary, 2008. Lecture Notes in Computer Science, vol. 4959, pp. 197–212. Springer, Berlin (2008)

29. Méndez, M., Navas, J., Hermenegildo, M.V.: An efficient, parametric fixpoint algorithm for incremental analysis of Java bytecode. In: Proc. of the Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Braga, Portugal, March 2007. Electronic Notes on Theoretical Computer Science, vol. 190(1), pp. 51–66
30. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Proc. of the 2nd Symposium on Programs as Data Objects (PADO II), Aarhus, Danemark, May 2001. Lecture Notes in Computer Science, vol. 2053, pp. 155–172. Springer, Berlin (2001)
31. Müller, P.: Reasoning about object structures using ownership. In: Meyer, B., Woodcock, J. (eds.) Proc. of the Workshop on Verified Software: Theories, Tools, Experiments (VSTTE'07). Lecture Notes in Computer Science, vol. 4171. Springer, Berlin (2007)
32. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Proc. of OOPSLA'91, ACM SIGPLAN Notices, vol. 26(11), pp. 146–161. ACM Press, New York (1991)
33. Payet, É., Spoto, F.: Magic-sets transformation for the analysis of Java bytecode. In: Nielson, H.R., Filé, G. (eds.) Proceedings of the 14th International Static Analysis Symposium (SAS'07), Kongens Lyngby, Denmark, August 2007. Lecture Notes in Computer Science, vol. 4634, pp. 452–467. Springer, Berlin (2007)
34. Pollet, I., Le Charlier, B., Cortesi, A.: Distinctness and sharing domains for static analysis of Java programs. In: 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, June 2001. Lecture Notes in Computer Science, vol. 2072, pp. 77–98. Springer, Berlin (2001)
35. Rossignoli, S., Spoto, F.: Detecting non-cyclicity by abstract compilation into boolean functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) Proc. of Verification, Model Checking and Abstract Interpretation, Charleston, SC, USA, January 2006. Lecture Notes in Computer Science, vol. 3855, pp. 95–110. Springer, Berlin (2006)
36. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
37. Schmidt, D.A.: Trace-based abstract interpretation of operational semantics. J. Lisp Symb. Comput. **10**(3), 237–271 (1998)
38. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Hankin, C. (ed.) Proc. of the 12th Static Analysis Symposium (SAS'05), London, UK, September 2005. Lecture Notes in Computer Science, vol. 3672, pp. 320–335. Springer, Berlin (2005)
39. Spoto, F.: Precise null-pointer analysis. J. Softw. Syst. Model. (to appear)
40. Spoto, F.: Watchpoint semantics: a tool for compositional and focussed static analyses. In: Cousot, P. (ed.) Proceedings of the 8th International Static Analysis Symposium (SAS'01), Paris, July 2001. Lecture Notes in Computer Science, vol. 2126, pp. 127–145. Springer, Berlin (2001)
41. Spoto, F.: Nullness analysis in Boolean form. In: Cerone, A., Gruner, S. (eds.) Proc. of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08), Cape Town, South Africa, November 2008, pp. 21–30. IEEE, New York (2008)
42. Spoto, F., Hill, P.M., Payet, É.: Path-length analysis for object-oriented programs. In: Proc. of Emerging Applications of Abstract Interpretation, Vienna, Austria March 2006. profs.sci.univr.it/~spoto/papers.html
43. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for Java bytecode based on path-length. ACM Trans. Program. Lang. Syst. **32**(3) (2010)
44. Sussman, G.J., Steele, G.L.: Scheme: An interpreter for extended lambda calculus. In: AI Memo, vol. 349. MIT Artificial Intelligence Laboratory (December 1975)
45. Sussman, G.J., Steele, G.L.: Scheme: An interpreter for extended lambda calculus. High.-Order Symb. Comput. **11**(4), 405–439 (1998)
46. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**, 285–309 (1955)
47. The JULIA Static Analyser. http://julia.scienze.univr.it
48. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2,3), 167–187 (1996)
49. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)