Magic-sets transformation for the analysis of Java bytecode

Etienne Payet, Fausto Spoto

Magic-Sets Transformation for the Analysis of Java Bytecode

Étienne Payet[1] and Fausto Spoto[2]

[1] IREMIA, Université de la Réunion, France
[2] Dipartimento di Informatica, Università di Verona, Italy

Abstract. Denotational static analysis of Java bytecode has a nice and clean compositional definition and an efficient implementation with binary decision diagrams. But it models only 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. We overcome this limitation with a technique used up to now for logic programs only. It adds new *magic* blocks of code to P, whose functional behaviours are the internal behaviours of P. We prove this transformation correct with an operational semantics. We define an equivalent denotational semantics, 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 and *non-cyclicity* of variables. We get a static analyser for full Java bytecode that is faster and scales better than another operational pair-sharing analyser and a constraint-based pointer analyser.

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, verifying program annotations or security constraints [13]. This is very important for low-level languages such as Java bytecode, downloaded from insecure networks in a machine-independent, non-optimised format. Since its source code is not available, its direct analysis is desirable.

Correctness is usually mandatory for static analysis and proved *w.r.t.* a *reference semantics* of the analysed language. Abstract interpretation [7] 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 and can only model program properties that can be formalised in terms of the reference semantics.

There are three main ways of giving semantics to a piece of code c [18]: *operational semantics* models c's execution as a transition relation over *configurations*, which include implementational details such as activation stacks and return points from calls; *denotational semantics* provides instead a *denotation i.e.,* a function from the input state provided to c (the values of the variables before c is executed) to the resulting output state (the same values after c has been

executed); *axiomatic semantics* derives the *weakest precondition* which must hold before the execution of c from a given postcondition which holds after it.

A major drawback of denotational semantics is that denotations model only the *functional i.e.,* input/output behaviour of the code: they do not express its *internal i.e.,* input/internal program points behaviours. The derived static analyses inherit this drawback, which makes them almost useless in practice. Consider the Java code in Fig. 1, which implements a list of `C`'s with two cloning methods: `clone` returns a shallow copy of a list and `deepClone` a deep copy, where also the `C`'s have been cloned. Hence, in `main`:

1. the return value of `clone` *shares* data structures with the list `v1`, namely, its `C`'s objects. Moreover, it is a *non-cyclical* list, since `v1` is a non-cyclical list;
2. the return value of `deepClone` does not share with `v1`, since it is a deep copy, and is also non-cyclical.

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

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

 private List() {
  List cursor = null;
  for (int i = 5; i > 0; i--)
   cursor = new List(new C(i),cursor);
  head = new C(0);  tail = cursor;
 }

 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.clone(),null);
  else
   return new List(head.clone(),tail.deepClone());
 }

 public static void main(String[] args) {
  List v1 = new List();
  List v2 = v1.clone();
  v2 = v1.deepClone();
 }
}
```

Fig. 1. Our running example.

Sharing analysis of pairs of variables and *non-cyclicity* analysis of variables, based on denotational semantics and implemented with a *pair-sharing domain* [14] and a *non-cyclicity domain* [12], can only prove 2, since 1 needs information at the internal program point just after the call to `clone`. If we add a command at the end of `main`, they cannot even prove 2 anymore.

Years ago abstract interpretation was largely secluded in the nowadays crystallised realm of logic (sometimes functional) 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, specific to logic languages, which adds extra *magic* clauses whose functional behaviours are the internal behaviours of P [2,3,6]. Codish [5] 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, even low-level ones such as Java bytecode. Suddenly, operational semantics became *the* reference semantics. This was a consequence of the lack of a magic-sets transformation for imperative languages and of the intuitive definition of operational semantics, very close to an actual implementation of the run-time engine of the language.

Our contributions here are the definition of a magic-sets transformation for Java bytecode, its proof of correctness, its implementation inside our JULIA de-

notational analyser [15], its instantiation with two domains for pair-sharing [14] and non-cyclicity [12] and its evaluation and comparison with an operational analyser for pair-sharing [10] and the points-to analyser SPARK [8]. JULIA is one or two orders of magnitude faster. It scales to programs of up to 19000 methods, for which the other two analysers are not always applicable.

To understand why we want to *rediscover* denotational static analysis and why its implementation JULIA is so efficient, consider the following:

- if method m (constructor, function, procedure...) is called in 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. Operational analyses 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 [4]. Hence there is a potentially very efficient implementation of denotational static analyses, which is not always the case for operational static 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 activation stacks nor return points from calls. Hence it is simpler to abstract than an operational semantics;
- denotational semantics models naturally properties of the functional behaviour of the code, such as information flows [13]. Operational semantics is very awkward here.

These are not *theoretical* insights, as our experiments show in Section 7.

2 Our Magic-Sets Transformation for Java Bytecode

The left of Fig. 2 reports the Java bytecode for the `main` method in Fig. 1, after a light preprocessing performed by our JULIA analyser. It has a simple sequential control, being a single block of code. This is because we do not consider exceptions for simplicity, which are implicitly raised by some instructions and break the sequential structure of the code without changing the sense of our magic-sets transformation (our actual implementation in JULIA considers exceptions). The code in Figure 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. For type inference and resolution we used the official algorithms [9]. A method or constructor implementation in class κ, named m, expecting parameters of types $\boldsymbol{\tau}$ and returning a value of type t is written as

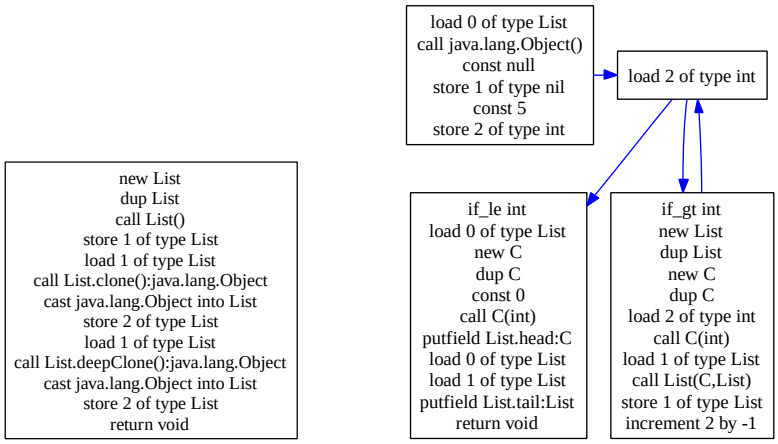

Fig. 2. The Java bytecode of `main` and of the empty constructor of `List` in Fig. 1.

$\kappa.m(\boldsymbol{\tau}) : t$. The `call` instruction implements the four `invoke`'s available in Java bytecode. It reports the explicit list of method or constructor implementations that it might call at run-time, accordingly with the semantics of the specific `invoke` that it implements. We allow more than one implementation for late-binding, but we use only one in our examples, for simplicity. Dynamic lookup of the correct implementation of a method is performed by *filter* instructions at the beginning of each method, which we do not show for simplicity.

Local variable 1 on the left of Fig. 2 implements variable `v1` in Fig. 1. Hence, just after the call to `clone`, it *shares* with the return value of `clone`, left on top of the stack, and is non-cyclical; after the call to `deepClone`, it *does not share* with the return value of `deepClone`, left on top of the stack, and is non-cyclical. To prove these results with a denotational analysis, our magic-sets transformation builds new *magic* blocks of code whose functional behaviours are the internal behaviours just after the calls to `clone` and `deepClone` on the left of Fig. 2.

Let us describe this transformation. It starts by splitting the code after the two calls to `clone` and `deepClone`, since we want to observe the intermediate states there. For reasons that will be clear soon, it also splits the code before each `call`. The result is in Fig. 3. The original code is split into blocks $0, \ldots, 5$ now. These have outgoing dashed arrows leading to new grey *magic* blocks $m0, \ldots, m5$. Block mk contains the same bytecode as block k plus a leading `blockcall` mp, where p is the predecessor of block k, if any.

The functional behaviour of magic block mk coincides with the internal behaviour at the end of block k. For instance, the functional behaviours of $m2$ and $m4$ are maps from the input state provided to the program to the intermediate states just after the calls to `clone` and `deepClone`, respectively. To understand why, let us start from $m0$. It is a clone of block 0 so that, at its end, the computation reaches the intermediate state at the internal program point between 0

and 1. Block $m1$ executes $m0$ (because of the `blockcall m0` instruction), then the same instructions as block 1. At its end, the computation reaches hence the intermediate state at the internal program point between 1 and 2. The same reasoning applies to the other magic blocks.

Consider the Java bytecode of the empty constructor of `List` in Fig. 1 now, called by `main` and shown on the right of Fig. 2. It is not sequential since it contains a loop. Its magic-sets transformation is in Fig. 4. As for `main`, we split the original code before each `call`; each magic block mk contains the code of k plus a leading `blockcall` to the predecessor(s) of k, if any. Since 8 has two predecessors 7 and 11, block $m8$ starts with `blockcall m7 m11` *i.e.,* there are two ways of reaching 8 and the states observable at its end are obtained by executing `load 2 of type int` from a state reachable at the end of 7 or 11. Something new happens for $m6$. It starts with a call to $m0$ in Fig. 3, which provides the intermediate states just before the only call in the program to this empty constructor of `List`. Block $m6$ continues with a `makescope List()` instruction which builds the scope of the constructor: in Java bytecode the caller stores the actual arguments on the operand stack and the callee retrieves them from the local variables [9]. Hence `makescope List()` copies the `List` object, left on top of the stack by $m0$ (*i.e.,* the implicit `this` parameter), into local variable 0 and clears the stack. At the end of $m6$ we observe hence the states reachable at the internal program point between blocks 6 and 7. 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.

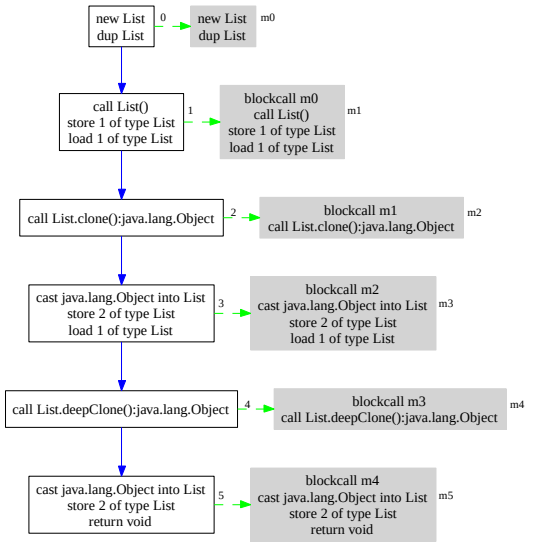

Fig. 3. The magic-sets transformation of the Java bytecode on the left of Fig. 2.

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. 3 and Fig. 4. We assume that the starting block of a method has no predecessors and does not start with a `call`, without loss of generality since it is always possible to add an extra initial block containing `nop`; we assume that the other blocks have at least a predecessor, since otherwise they would be dead-code and eliminated; we assume that each

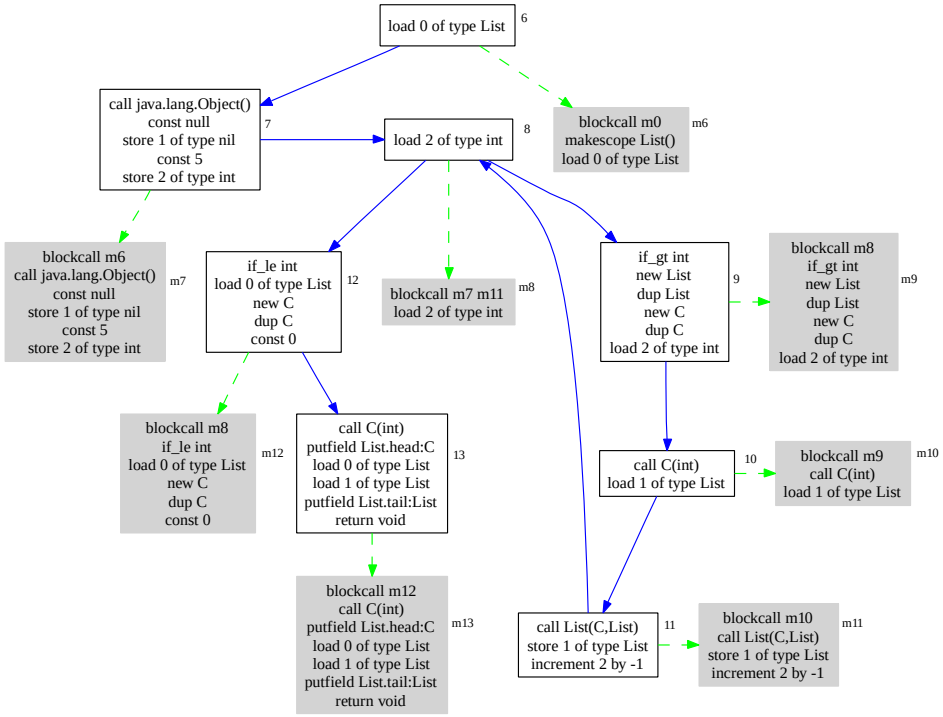

Fig. 4. The magic-sets transformation of the constructor on the right of Fig. 2.

`call` starts a block and that each `return` ends a block with no successors; we assume that the `main` method is not called from inside the program, without loss of generality since we can always rename `main` into `main'` wherever in the program and add a new `main` which 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

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

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*

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

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

Definition 1 has three cases. In the first case block k does not start a method (or constructor). Hence it has $l > 0$ predecessors and magic block mk begins with a `blockcall` to their magic blocks, as block *m8* in Fig. 4. In the second and third case block k starts a method or constructor $\kappa.m(\boldsymbol{\tau}) : t$, so that it has no predecessors. If the program P calls $\kappa.m(\boldsymbol{\tau}) : t$ (second case) there are $u > 0$ predecessors of those `call`s, 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 $\kappa.m(\boldsymbol{\tau}) : t$, as for block *m6* in Fig. 4. Otherwise (third case), P never calls $\kappa.m(\boldsymbol{\tau}) : t$ and mk is a clone of k, as for block *m0* in Fig. 3.

4 Operational Semantics of the Java Bytecode

In this section we describe an operational semantics of the Java bytecode, which we use in Section 5 to prove our magic-sets transformation correct.

Definition 2. *A* state *of the Java Virtual Machine is a triple* $\langle l \parallel s \parallel \mu \rangle$ *where* l *maps local variables to* values*,* s *is a stack of values (the* operand stack*), which grows leftwards, and* μ *is a* memory*, or* heap*, which maps* locations *into* objects*. We do not formalise further what values, memories and objects are, since this is irrelevant here. The set of states is* Σ*.* □

The semantics of a bytecode instruction `ins` different from `call` and `blockcall` is a partial map *ins* from states to states. For instance, the semantics of `dup` t is

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

where $s = top :: s'$ and *top* has type t. This is always true since legal Java bytecode is verifiable [9]. The semantics of `load` i `of type` t is

$$load\ i\ of\ type\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel l(i) :: s \parallel \mu\rangle$$

where $l(i)$ exists and has type t since legal Java bytecode is verifiable.

Also the semantics of a `return` t bytecode is a map over states, which leaves on the operand stack only those elements which hold the return value of type t:

$$return\ t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle l \parallel vs \parallel \mu\rangle$$

where $s = vs :: s'$ and vs are the stack elements which hold the return value. If $t = \texttt{void}$ then $vs = \varepsilon$. We formalise later in Definition 5 how control returns to the caller. Also the semantics of a conditional bytecode is a map over states, undefined when its condition is false. For instance, the semantics of $\texttt{if\_le}\ t$ is

$$\mathit{if\_le}\ t = \lambda\langle l \parallel s \parallel \mu\rangle. \begin{cases} \langle l \parallel s' \parallel \mu\rangle & \text{if } \mathit{top} \leq 0 \\ \mathit{undefined} & \text{otherwise,} \end{cases}$$

where $s = \mathit{top} :: s'$ and top has numerical type t.

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

Definition 3. *Let $\kappa.m(\boldsymbol{\tau}) : t$ 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 $\kappa.m(\boldsymbol{\tau}) : t) : \Sigma \to \Sigma$ as*

$$\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t = \lambda\langle l \parallel s \parallel \mu\rangle.\langle [i \mapsto v_i \mid 0 \leq i < p] \parallel \varepsilon \parallel \mu\rangle,$$

where $s = v_{p-1} :: \cdots :: v_0 :: s'$ since legal Java bytecode is verifiable. □

Definition 3 formalises the fact that the ith local variable of the callee is a copy of the element $p - 1 - i$ positions down the top of the stack of the caller.

Definition 4. *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.

Definition 5. *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:*

$$\frac{\texttt{ins}\ \textit{is not a}\ \texttt{call}\ \textit{nor a}\ \texttt{blockcall}}{\left\langle \boxed{\begin{matrix}\texttt{ins}\\ \mathit{rest}\end{matrix}} \overset{\ell}{\rightrightarrows} \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \parallel \sigma\right\rangle :: a \Rightarrow \left\langle \boxed{\mathit{rest}} \overset{\ell}{\rightrightarrows} \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \parallel \mathit{ins}(\sigma)\right\rangle :: a} \tag{1}$$

$$\frac{\begin{matrix}b\ \textit{is the block where method}\ \kappa.m(\boldsymbol{\tau}) : t\ \textit{starts}\\ \sigma = \langle l \parallel \mathit{pars} :: s \parallel \mu\rangle, \quad \mathit{pars}\ \textit{are the actual parameters of the call}\\ \sigma' = (\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t)(\sigma)\end{matrix}}{\left\langle \boxed{\begin{matrix}\texttt{call}\ \kappa.m(\boldsymbol{\tau}){:}t\\ \mathit{rest}\end{matrix}} \overset{\ell}{\rightrightarrows} \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \parallel \sigma\right\rangle :: a \Rightarrow \langle b \parallel \sigma'\rangle :: \left\langle \boxed{\mathit{rest}} \overset{\ell}{\rightrightarrows} \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \parallel \langle l \parallel s \parallel \mu\rangle\right\rangle :: a} \tag{2}$$

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

$$\frac{1 \leq i \leq m}{\left\langle \boxed{\ }^{k} \overset{}{\rightrightarrows} \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \parallel \sigma\right\rangle :: a \Rightarrow \langle b_i \parallel \sigma\rangle :: a} \tag{4}$$

$$\frac{1 \leq i \leq l}{\left\langle \boxed{\begin{array}{l}\texttt{blockcall}\ mp_1 \cdots mp_l \\ rest\end{array}}^{mk} \parallel \sigma \right\rangle :: a \Rightarrow \langle P(mp_i) \parallel \sigma \rangle :: \left\langle \boxed{rest}^{mk} \parallel \sigma \right\rangle :: a} \quad (5)$$

$$\overline{\left\langle \square^{mk} \parallel \sigma \right\rangle :: \langle b \parallel \sigma' \rangle :: a \Rightarrow \langle\, b \parallel \sigma \rangle :: a} \quad (6)$$

□

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 3. 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. Control returns to the caller by rule (3), which 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. Rule (4) 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 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 (5) 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 (6) 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) and (4) only for the original blocks; rules (5) and (6) only for the magic ones.

Our small step operational semantics allows us to define the set of intermediate states at a given, internal program point $*$, provided $*$ ends a block. This can always be obtained by splitting after $*$ the block where $*$ occurs.

Definition 6. *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*

$$\Sigma_k = \{\sigma \mid \langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \square^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a\}$$

□

Note that Σ_k is in general 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 Section 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 with the internal behaviour at the end of k.

Theorem 1. *Let σ_{in} be the initial state provided to the* `main` *method of P and $k \in \mathbb{N}$ a block of P. We have $\Sigma_k = \{\sigma \mid \langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \square^{mk} \parallel \sigma \rangle\}$.* □

In Section 6 we define a denotational semantics for the Java bytecode and prove it *equivalent* to our operational semantics of Section 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 the Java Bytecode

A denotational semantics for the Java bytecode maps each block of code b in a *denotation* $[\![b]\!]$ *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. Hence, if b_{in} is the initial block of method `main`, then $[\![b_{in}]\!]$ is the functional behaviour of the whole program.

Definition 7. *A denotation is a partial function from an* input *state to an* output *or* final *state. The set of denotations is written as Δ. Let $\delta_1, \delta_2 \in \Delta$. Their* sequential composition *is $\delta_1; \delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$, which is undefined when $\delta_1(\sigma)$ is undefined or when $\delta_2(\delta_1(\sigma))$ is undefined.* □

It follows that the semantics *ins* of a bytecode `ins` is a denotation.

Let $\delta \in \Delta$ be the functional behaviour of a method $\kappa.m(\boldsymbol{\tau}) : t$. At its beginning the operand stack is empty and the local variables hold the actual arguments of the call. At its end the operand stack holds the return value of $\kappa.m(\boldsymbol{\tau}) : t$ only, if any (the semantics of `return` drops all stack elements but the return value. See Section 4). From the point of view of a caller executing a `call` $\kappa.m(\boldsymbol{\tau}) : t$, the local variables and the operand stack do not change, except for the actual arguments which get popped from the stack and substituted with the return value, if any. The final memory is that reached at the end of $\kappa.m(\boldsymbol{\tau}) : t$. These considerations let us *extend* δ into the denotation of the `call` instruction.

Definition 8. *Let $\delta \in \Delta$ and $\kappa.m(\boldsymbol{\tau}) : t$ be a method. We define the operator extend $\kappa.m(\boldsymbol{\tau}) : t \in \Delta \mapsto \Delta$ as*

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

where $\langle l' \parallel \mathit{vs} \parallel \mu'\rangle = \delta((\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t)(\langle l \parallel \mathit{pars} :: s \parallel \mu\rangle))$, pars are the actual parameters passed to $\kappa.m(\boldsymbol{\tau}) : t$ and vs its return value, if any. □

An *interpretation* is a set of denotations for each block of P. *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 9. *An* interpretation *for P is a map from P's blocks into sets of denotations. The set of interpretations I is ordered by pointwise set-inclusion.* □

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, which can call methods and blocks of P.

Definition 10. *Let $\iota \in I$. The* denotations in ι of an instruction *are*

$$[\![\texttt{ins}]\!]^\iota = \{ins\} \quad \textit{if } \texttt{ins} \textit{ is not a } \texttt{call} \textit{ nor a } \texttt{blockcall}$$
$$[\![\texttt{blockcall } mp_1 \cdots mp_l]\!]^\iota = \iota(P(mp_1)) \cup \cdots \cup \iota(P(mp_l))$$
$$[\![\texttt{call } \kappa.m(\boldsymbol{\tau}) : t]\!]^\iota = (\textit{extend } \kappa.m(\boldsymbol{\tau}) : t)(\iota(b_{\kappa.m(\boldsymbol{\tau}):t}))$$

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

$$\left[\!\!\left[\boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_\texttt{n}\end{array}}^{\ell} \rightrightarrows \begin{array}{c} b_1\\ \cdots\\ b_m\end{array} \right]\!\!\right]^\iota = \begin{cases} [\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_\texttt{n}]\!]^\iota & \textit{if } m = 0 \\ [\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_\texttt{n}]\!]^\iota ; (\iota(b_1) \cup \cdots \cup \iota(b_m)) & \textit{if } m > 0. \end{cases}$$

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

The blocks of P are in general interdependent, because of loops and method calls, and a denotational semantics must be built through a fixpoint computation. Given an empty approximation $\iota \in I$ of the denotational semantics, one improves it into $T_P(\iota) \in I$ and iterates the application of T_P until a fixpoint[3].

Definition 11. *The* transformer $T_P : I \mapsto I$ *for P is defined as $T_P(\iota)(b) = [\![b]\!]^\iota$ for every $\iota \in I$ and block b of P.* □

Proposition 1. *The operator T_P is additive, so its least fixpoint exists [17].* □

Definition 12. *Let P be a Java bytecode program (possibly enriched with its magic blocks). Its* denotational semantics *$\mathcal{D}_P$ is the least fixpoint $\sqcup_{i\geq 0} T_P^i$ of T_P, where $T_P^0(b) = \varnothing$ for every block b of P and $T_P^{i+1} = T_P(T_P^i)$ for every $i \geq 0$.* □

We show now that the operational semantics of Section 4 and the denotational semantics of this section coincide, so that (Theorem 1) the denotation of a magic block mk is the internal behaviour at the end of block k.

Theorem 2. *Let b 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 Section 4, coincides with its denotational semantics:*

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

[3] 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.

7 Experiments

We have implemented our magic-sets transformation inside the generic analyser JULIA for Java bytecode [15] and used it with two abstract domains. The first [14] overapproximates the set of pairs of program variables, which for the Java bytecode means local variables or stack elements, which *share i.e.,* reach the same memory location; it is used for automatic program parallelisation and to support other analyses. The second [12] overapproximates the set of *cyclical* program variables, those which reach a loop of memory locations; it needs a preliminary pair-sharing analysis. We used Boolean formulas to abstract sets of denotations by relating properties of their input to properties of their output. 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 [14]). We have implemented Boolean formulas through binary decision diagrams [4].

Let us consider pair-sharing. JULIA computes the formula $(l1, s0)$ as abstract denotation for block *m2* in Fig. 3. It states that $(l1, s0)$ is true for *m2 i.e.,* at its end, only local variable 1, which holds the list `v1` of Fig. 1, might share with stack element 0 (the base of the stack), which holds the return value of `clone`. Hence JULIA proves that all other pairs of local variables and stack elements definitely do not share. This is an optimal approximation of the behaviour of the program between blocks 2 and 3. JULIA computes $(l1, l2)$ as abstract denotation for block *m4*. Hence it proves that local variables *l1* (`v1` in Fig. 1) and *l2* (`v2` in Fig. 1) *might* share there, while all other local variables and stack elements definitely do not share; in particular, the return value of `deepClone` (the stack element 0) does not share with `v1`. Note that `v1` and `v2` actually share after the call to `deepClone`, whose return value has not been stored into `v2` yet. This is an optimal approximation of the behaviour of the program between blocks 4 and 5. Let us consider cyclicity analysis. JULIA computes *false* as abstract denotation of both *m2* and *m4* in Fig. 3 *i.e.,* it proves that no local variable and stack element might be cyclical there, which is an optimal approximation of the behaviour of the program between blocks 2 and 3 and between blocks 4 and 5. In conclusion, JULIA proves both points 1 and 2 of Section 1.

Fig. 1 shows a simple program. More complex benchmarks such as those in Fig. 5 challenge the scalability, the efficiency and the precision of the analyses. The first and smaller 4 have been also analysed with the pair-sharing analyser in [10] so we can build a comparison. The others are progressively larger to check the scalability of the analyses. Fig. 5 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 its percentage due to the magic-sets transformation, which is never more than 31%. We consider two scenarios: whether the Java libraries are not analysed (calls to the missing classes use a worst-case assumption) or they are analysed, for more precise but more costly analyses. We used an Intel Xeon machine running at 2.8GHz, with 2.5 gigabytes of RAM, Linux 2.6.15 and Sun jdk 1.5.

	libraries are not included			libraries are included		
	methods	preproc.	magic-sets	methods	preproc.	magic-sets
Qsort	8	369	2.14%	72	767	3.65%
IntegerQsort	9	369	2.14%	72	765	3.66%
Passau	10	351	1.51%	13	388	1.19%
ZipVector	13	395	2.48%	76	778	4.08%
JLex	130	1292	14.53%	744	2160	10.97%
JavaCup	293	1502	9.8%	1136	2657	21.88%
julia	1441	3351	13.56%	4809	8552	14.9%
jess	1506	3344	25.1%	6046	9911	28.88%
jEdit	2473	6887	22.74%	7943	15156	30.77%
soot	15617	75925	10.49%	19032	84709	14.54%

Fig. 5. Size and preprocessing times (in milliseconds) for our benchmarks.

	sharing analysis				**cyclicity analysis**			
	libr. not included		libr. included		libr. not included		libr. included	
	time	precision	time	precision	time	precision	time	precision
Qsort	127	35.09%	267	71.79%	20	100.00%	43	100.00%
IntegerQsort	208	36.17%	295	53.46%	23	100.00%	38	100.00%
Passau	152	36.88%	118	43.03%	14	100.00%	6	100.00%
ZipVector	251	21.15%	395	40.47%	34	98.38%	50	100.00%
JLex	1438	30.34%	2312	33.86%	269	80.69%	877	83.22%
JavaCup	2418	16.26%	4996	22.96%	474	87.25%	1836	93.82%
julia	10829	11.03%	33589	12.22%	2852	78.48%	5245	83.59%
jess	24526	12.79%	66163	15.96%	4136	73.00%	8293	79.64%
jEdit	42332	16.34%	135208	19.92%	6654	76.08%	15926	81.50%
soot	125819	6.26%	282923	7.69%	113884	73.70%	196456	80.42%

Fig. 6. Time (in milliseconds) and precision of our sharing and cyclicity analyses.

Fig. 6 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 which are proved not to share, definitely, before a `putfield`, an `arraystore` or a `call`. We consider only variables and stack elements of reference type since primitive types cannot share in Java (bytecode); only `putfield`'s, `arraystore`'s and `call`'s since it is there that sharing analysis helps other analyses (see for instance [12] for its help to cyclicity analysis). Precision, for cyclicity analysis, is the percentage of local variables or stack elements which are proved to be non-cyclical, definitely, before a `getfield` bytecode. We consider only variables and stack elements of reference type since primitive values are never cyclical; only `getfield`'s since cyclicity information is typically used there, for instance to prove termination of iterations over dynamic data-structures [16]. For better efficiency, we *cache* the analysis of each bytecode so that, if it is needed twice, we only compute it once. This happens frequently with our magic-sets transformation, which introduces code duplication. For instance, block *m1* in Fig. 3 shares three bytecodes which block 1. This technique has

been inspired by a similar optimisation of the analysis of *magic logic programs*, defined in [5]. Since it caches the *functional* behaviour of the code, it is different from *memoisation*, which only caches its behaviour for *each given* input state.

We are not aware of any other cyclicity analysis for Java bytecode. An operational pair-sharing analyser was instead applied [10] to the smallest 4 benchmarks in Fig. 5, without including the library classes. We asked its authors to try it on our machine and benchmarks, without any answer. They take time $P + T + A$: P is the preprocessing time, which they do not report. Since they use the generic analyser SOOT, we could compute P with SOOT version 2.2.2; T is the time to transform the output of SOOT into the format required in [10]. We cannot estimate T without the analyser; A is the *preliminary running time* reported in [10], normalised *w.r.t.* the relative speeds of our machine and theirs. Fig. 7 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 T. Exceptions, subroutines, static initialisers and native methods are not tested by such small benchmarks. So it is not clear if the analyser in [10] is ready for *real* analyses. Precision is expressed in [10] as a *level of multivariance* which we could not translate into our (more natural) notion. Another analysis for (definite) sharing is implemented in [11] for a *restricted* subset of Java. Times and precision are not reported. The code is not publicly available.

	JULIA	[10]
Qsort	496	≥1625
IntegerQsort	577	≥1335
Passau	502	≥1595
ZipVector	646	≥2780

Fig. 7. Times (in milliseconds) of our pair-sharing analysis and of that in [10].

We compared our pair-sharing analysis with the SPARK [8] points-to analysis, also based on SOOT. Points-to and sharing information are somehow similar. SPARK includes all Java libraries in the analysis as also JULIA can do. SOOT, SPARK and JULIA are all written in Java. Hence a comparison is relatively fair. Fig. 8 compares the overall running times, including preprocessing. JULIA is always faster, up to two orders of magnitude as for Passau; it completes all analyses while SPARK stops in three cases with hard-to-understand run-time errors (for jEdit and jess: *This operation requires resolving level hierarchy but* `someclass` *is at resolving level dangling*; for julia: *couldn't find class* `jdd.bdd.BDD`, which does not exist and is not used by JULIA). We stress that sharing and points-to analyses are anyway different analyses and neither of them is an abstraction of the other. Hence this comparison only indicates that JULIA compares well *w.r.t.* an existing tool.

	JULIA	[8]
Qsort	1.0	93
IntegerQsort	1.1	92
Passau	0.5	89
ZipVector	1.2	90
JLex	4.5	95
JavaCup	7.7	99
julia	42.1	fails
jess	76.1	fails
jEdit	150.4	fails
soot	367.7	452

Fig. 8. Times (in seconds) of our pair-sharing analysis and of the points-to analysis in [8].

8 Conclusion

Our experiments show that denotational analyses of Java bytecode, with a preliminary magic-sets transformation, are feasible, fast and compare well with other analyses. We will soon use widenings [7] to still improve their efficiency.

Our magic-sets transformation is completely independent from the abstract domains, which 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, which depends on the specific goal for which he develops the abstract domain.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.
2. F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of the 5th ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.
3. C. Beeri and R. Ramakrishnan. On the Power of Magic. *The Journal of Logic Programming*, 10(3 & 4):255–300, 1991.
4. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
5. M. Codish. Efficient Goal Directed Bottom-Up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
6. M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.
7. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.
8. Lhoták H. and L. Hendren. Scaling Java Points-to Analysis using Spark. In G. Hedin, editor, *Proc. of Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169, Warsaw, Poland, April 2003.
9. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
10. M. Méndez, J. Navas, and M. V. Hermenegildo. An Efficient, Parametric Fixpoint Algorithm for Incremental Analysis of Java Bytecode. In *Proc. of the second workshop on Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes on Theoretical Computer Science, Braga, Portugal, March 2007. To appear.
11. I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 77–98, Budapest, Hungary, June 2001.
12. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006.

13. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
14. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. of Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005.
15. F. Spoto. The JULIA Static Analyser. `profs.sci.univr.it/~spoto/julia`, 2007.
16. F. Spoto, P. M. Hill, and E. Payet. 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`.
17. A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5:285–309, 1955.
18. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

9 Proofs (not meant for publication)

We use the notation $\Rightarrow^n$, standing for $n \geq 0$ steps of derivation, and the notation $\underset{(r)}{\Rightarrow}$ standing for a single derivation step through rule r.

9.1 Proof of Theorem 1

The proof follows from the next 2 propositions.

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

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a$$

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

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^n \langle \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a$$

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 $\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^0 \langle \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a$. Then, $b_{in} = \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$, $\sigma_{in} = \sigma$ 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

  $$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^{n+1} \langle \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a \ .$$

  Then $\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^n a_n \Rightarrow \langle \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a$. Let us consider the rule of Definition 5 that is used in the last derivation step, from a_n.
  1. If rule (1) is used then $a_n = \langle \boxed{\begin{smallmatrix} \texttt{ins} \\ rest \end{smallmatrix}}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma' \rangle :: a$ and $\sigma = ins(\sigma')$. By inductive hypothesis,

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

Moreover,

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

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

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

where $\sigma' = \langle l \parallel pars :: s \parallel \mu \rangle$ and $\sigma = (makescope\ \kappa.m(\boldsymbol{\tau}) : t)(\sigma')$. Moreover, a has the form $\langle \boxed{rest}^{k'} \rightrightarrows \cdots \parallel \langle l \parallel s \parallel \mu \rangle \rangle :: a'$.
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, $\boxed{\begin{array}{c}\texttt{call } \kappa.m(\boldsymbol{\tau}){:}t\\ rest\end{array}}^{k'} \rightrightarrows \cdots$ 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

$$\begin{array}{l} \langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\ }^{p_i} \rightrightarrows \cdots \parallel \sigma' \rangle :: a' \\ \qquad \underset{(4)}{\Rightarrow} \underbrace{\langle \boxed{\begin{array}{c}\texttt{call } \kappa.m(\boldsymbol{\tau}){:}t\\ rest\end{array}}^{k'} \rightrightarrows \cdots \parallel \sigma' \rangle :: a'}_{a_n} \end{array}$$

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

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

So, we have:

$$\langle P(mk) \parallel \sigma_{in} \rangle \underset{(5)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{\begin{array}{c}\texttt{makescope } \kappa.m(\boldsymbol{\tau}){:}t\\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle$$

and, by inductive hypothesis,

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

Consequently,

$$\begin{array}{l} \langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\ }^{mp_i} \parallel \sigma' \rangle :: \langle \boxed{\begin{array}{c}\texttt{makescope } \kappa.m(\boldsymbol{\tau}){:}t\\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle \\ \qquad \underset{(6)}{\Rightarrow} \langle \boxed{\begin{array}{c}\texttt{makescope } \kappa.m(\boldsymbol{\tau}){:}t\\ rest\end{array}}^{mk} \parallel \sigma' \rangle \\ \qquad \underset{(1)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel \underbrace{(makescope\ \kappa.m(\boldsymbol{\tau}) : t)(\sigma')}_{\sigma} \rangle \ . \end{array}$$

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

$$\langle \boxed{}^{k'} \| \langle l' \| vs \| \mu' \rangle \rangle :: \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \langle l \| s \| \mu \rangle \rangle :: a$$

and $\sigma = \langle l \| vs :: s \| \mu' \rangle$. Notice that rule (3) corresponds to the situation where control returns to the caller $P(k)$, since the only rule which can create a new top configuration with a normal block is rule (2). As a `call` instruction is always located at the beginning of a block, we have

$$P(k) = \boxed{\begin{smallmatrix} \texttt{call}\ \kappa.m(\boldsymbol{\tau}){:}t \\ rest \end{smallmatrix}}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$$

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 $p_1, \ldots, p_l$. So, the derivation from $\langle b_{in} \| \sigma_{in} \rangle$ to a_n has the form

$$\begin{aligned}
\langle b_{in} \| \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{}^{p_i} \rightrightarrows \cdots \| \sigma_i \rangle :: a \\
&\underset{(4)}{\Rightarrow} \langle P(k) \| \sigma_i \rangle :: a \\
&\underset{(2)}{\Rightarrow} \langle b \| \sigma_i' \rangle :: \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \langle l \| s \| \mu \rangle \rangle :: a \\
&\Rightarrow^* \underbrace{\langle \boxed{}^{k'} \| \langle l' \| vs \| \mu' \rangle \rangle :: \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \langle l \| s \| \mu \rangle \rangle :: a}_{a_n}
\end{aligned}$$

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

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

Then,

$$\langle P(mk) \| \sigma_{in} \rangle \underset{(5)}{\Rightarrow} \langle P(mp_i) \| \sigma_{in} \rangle :: \langle \boxed{\begin{smallmatrix} \texttt{call}\ \kappa.m(\boldsymbol{\tau}){:}t \\ rest \end{smallmatrix}}^{mk} \| \sigma_{in} \rangle$$

and, by inductive hypothesis,

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

So we have

$$\begin{aligned}
\langle P(mk) \| \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{}^{mp_i} \| \sigma_i \rangle :: \langle \boxed{\begin{smallmatrix} \texttt{call}\ \kappa.m(\boldsymbol{\tau}){:}t \\ rest \end{smallmatrix}}^{mk} \| \sigma_{in} \rangle \\
&\underset{(6)}{\Rightarrow} \langle \boxed{\begin{smallmatrix} \texttt{call}\ \kappa.m(\boldsymbol{\tau}){:}t \\ rest \end{smallmatrix}}^{mk} \| \sigma_i \rangle \\
&\underset{(2)}{\Rightarrow} \langle b \| \sigma_i' \rangle :: \langle \boxed{rest}^{mk} \| \langle l \| s \| \mu \rangle \rangle \ .
\end{aligned}$$

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

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

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 $\langle \boxed{}^{k'} \rightrightarrows \begin{smallmatrix} b_1' \\ \cdots \\ b_{m'}' \end{smallmatrix} \parallel \sigma \rangle :: a$ and $\boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$ is a b_i'. Then, $\boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} = P(k)$. By the first case of Definition 1,

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

Hence,

$$\langle P(mk) \parallel \sigma_{in} \rangle \underset{(5)}{\Rightarrow} \langle P(mk') \parallel \sigma_{in} \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle \ .$$

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,

$$\begin{aligned}\langle P(mk) \parallel \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{}^{mk'} \parallel \sigma \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle \\ &\underset{(6)}{\Rightarrow} \langle \boxed{rest}^{mk} \parallel \sigma \rangle \ .\end{aligned}$$

5. Rule (5) cannot be used. Indeed, b_{in} is a "normal" (*i.e.,* non-magic) block and a normal block does not call any magic block. Hence, the block in a_n is not a magic block.
6. Rule (6) cannot be used for the same reasons 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*

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a$$

for some a.

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

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle :: a$$

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`. Hence, as $P(mk) = magic(P(k))$, $P(mk)$ is obtained from the third case of Definition 1. Consequently:

- $P(k) = \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$,
- $P(k)$ has no predecessor, so $P(k)$ is the starting block of a method $\kappa.m(\boldsymbol{\tau}) : t$,
- each block of P starting with `call` $\kappa.m(\boldsymbol{\tau}) : t$ 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` $\kappa.m(\boldsymbol{\tau}) : t$. Then $\kappa.m(\boldsymbol{\tau}) : t$ 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

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma \rangle \ .$$

– (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 $\langle P(mk) \parallel \sigma_{in} \rangle \Rightarrow^n a_n \Rightarrow \langle \boxed{rest}^{mk} \parallel \sigma \rangle$. Let us consider the rule of Definition 5 that is used in the derivation from a_n.

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

$$\langle b_{in} \parallel \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\begin{smallmatrix} \texttt{ins} \\ rest \end{smallmatrix}}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma' \rangle :: a \ .$$

Moreover,

$$\langle \boxed{\begin{smallmatrix} \texttt{ins} \\ rest \end{smallmatrix}}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel \sigma' \rangle :: a \underset{(1)}{\Rightarrow} \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \parallel ins(\sigma') \rangle :: a \ .$$

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

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

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

- $P(k) = \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$,
- $P(k)$ is the starting block of method $\kappa.m(\boldsymbol{\tau}) : t$,
- $P(mk) = \boxed{\begin{array}{c}\texttt{blockcall}\ \cdots mk' \cdots \\ \texttt{makescope}\ \kappa.m(\boldsymbol{\tau}):t \\ rest\end{array}}^{mk}$,
- $P(k')$ is a predecessor of a block of P, say $P(k'')$, that begins with $\texttt{call}\ \kappa.m(\boldsymbol{\tau}) : t$.

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

$$\begin{aligned}
\langle P(mk) \parallel \sigma_{in} \rangle &\underset{(5)}{\Rightarrow} \langle P(mk') \parallel \sigma_{in} \rangle :: \langle \boxed{\begin{array}{c}\texttt{ins} \\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle \\
&\Rightarrow^* \langle \square^{mk'} \parallel \sigma' \rangle :: \langle \boxed{\begin{array}{c}\texttt{ins} \\ rest\end{array}}^{mk} \parallel \sigma_{in} \rangle \\
&\underset{(6)}{\Rightarrow} \underbrace{\langle \boxed{\begin{array}{c}\texttt{ins} \\ rest\end{array}}^{mk} \parallel \sigma' \rangle}_{a_n}
\end{aligned}$$

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

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

As $P(k'')$ is a successor of $P(k')$ and $P(k'')$ begins with $\texttt{call}\ \kappa.m(\boldsymbol{\tau}) : t$, we have

$$\begin{aligned}
\langle b_{in} \parallel \sigma_{in} \rangle &\Rightarrow^* \langle \square^{k'} \rightrightarrows \cdots \parallel \sigma' \rangle :: a' \\
&\underset{(4)}{\Rightarrow} \langle \underbrace{\boxed{\begin{array}{c}\texttt{call}\ \kappa.m(\boldsymbol{\tau}):t \\ rest\end{array}}^{k''} \rightrightarrows \cdots}_{P(k'')} \parallel \sigma' \rangle :: a' \\
&\underset{(2)}{\Rightarrow} \langle P(k) \parallel ins(\sigma') \rangle :: \langle \boxed{rest}^{k''} \rightrightarrows \cdots \parallel \sigma'' \rangle :: a'
\end{aligned}$$

since $\texttt{ins} = \texttt{makescope}\ \kappa.m(\boldsymbol{\tau}) : t$. Hence, since $P(k) = \boxed{rest}^k \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$ and $\sigma = ins(\sigma')$:

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

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

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

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

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

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

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

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

$$\langle b_{in} \| \sigma_{in} \rangle \Rightarrow^* \langle \boxed{\begin{array}{l}\texttt{call } \kappa.m(\boldsymbol{\tau}){:}t \\ rest\end{array}}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \sigma_1 \rangle$$

Consequently,

$$\begin{aligned}
\langle b_{in} \| \sigma_{in} \rangle &\Rightarrow^* \langle \boxed{\begin{array}{l}\texttt{call } \kappa.m(\boldsymbol{\tau}){:}t \\ rest\end{array}}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \sigma_1 \rangle \\
&\underset{(2)}{\Rightarrow} \langle b \| \sigma_1' \rangle :: \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \langle l \| s \| \mu \rangle \rangle \\
&\Rightarrow^* \langle \boxed{}^{k'} \| \langle l' \| vs \| \mu' \rangle \rangle :: \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \langle l \| s \| \mu \rangle \rangle \\
&\underset{(3)}{\Rightarrow} \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \langle l \| vs :: s \| \mu' \rangle \rangle
\end{aligned}$$

i.e.,

$$\langle b_{in} \| \sigma_{in} \rangle \Rightarrow^* \langle \boxed{rest}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \| \sigma \rangle \ .$$

4. Rule (4) cannot be used. Indeed, in this rule the top of the resulting activation stack is $\langle b_i \| \sigma \rangle$ where b_i is not a magic block, while here $\boxed{rest}^{mk}$ is a magic block.
5. Rule (5) 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 $\langle \boxed{rest}^{mk} \| \sigma \rangle$, whose length is equal to 1.

6. If rule (6) is used then a_n has the form $\langle \boxed{}^{mk'} \parallel \sigma\rangle :: \langle \boxed{\mathit{rest}}^{mk} \parallel \sigma'\rangle$. Since only rule (5) pushes a magic block on top of the stack, block $P(mk)$ has the form $\boxed{\begin{array}{c}\texttt{blockcall}\ \cdots mk' \cdots \\ \mathit{rest}\end{array}}^{mk}$ and the derivation

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

has the form

$$\begin{array}{rl} \langle P(mk) \parallel \sigma_{in}\rangle & \underset{(5)}{\Rightarrow} \langle P(mk') \parallel \sigma_{in}\rangle :: \langle \boxed{\mathit{rest}}^{mk} \parallel \sigma_{in}\rangle \\ & \Rightarrow^{n-1} \underbrace{\langle \boxed{}^{mk'} \parallel \sigma\rangle :: \langle \boxed{\mathit{rest}}^{mk} \parallel \sigma_{in}\rangle}_{a_n} \end{array}$$

where $\langle P(mk') \parallel \sigma_{in}\rangle \Rightarrow^{n-1} \langle \boxed{}^{mk'} \parallel \sigma\rangle$. 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 $\boxed{\mathit{rest}}^{k} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array}$ and $P(k')$ is a predecessor of $P(k)$. By inductive hypothesis, $\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{}^{k'} \rightrightarrows \cdots \parallel \sigma\rangle :: a$. Hence

$$\begin{array}{rl} \langle b_{in} \parallel \sigma_{in}\rangle & \Rightarrow^* \langle \boxed{}^{k'} \rightrightarrows \cdots \parallel \sigma\rangle :: a \\ & \underset{(4)}{\Rightarrow} \langle \boxed{\mathit{rest}}^{k} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \parallel \sigma\rangle :: a \end{array}$$

i.e., $\langle b_{in} \parallel \sigma_{in}\rangle \Rightarrow^* \langle \boxed{\mathit{rest}}^{k} \rightrightarrows \begin{array}{c} b_1 \\ \cdots \\ b_m \end{array} \parallel \sigma\rangle :: a$.

□

9.2 Proof of Proposition 1

We first need a lemma.

Lemma 1. *Let* `ins` *be a bytecode instruction and* $\{\iota_j\}_{j\in J} \subseteq I$ *with* $J \subseteq \mathbb{N}$. *Then*

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

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

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

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

$$\begin{array}{rl} [\![\texttt{ins}]\!]^{\sqcup_{j\in J}\iota_j} & = (extend\ \kappa.m(\boldsymbol{\tau}) : t)((\sqcup_{j\in J}\iota_j)(b_{\kappa.m(\boldsymbol{\tau}):t})) \\ & = (extend\ \kappa.m(\boldsymbol{\tau}) : t)(\cup_{j\in J}\iota_j(b_{\kappa.m(\boldsymbol{\tau}):t})) \\ & = \cup_{j\in J}(extend\ \kappa.m(\boldsymbol{\tau}) : t)(\iota_j(b_{\kappa.m(\boldsymbol{\tau}):t})) \\ & = \cup_{j\in J} [\![\texttt{ins}]\!]^{\iota_j} . \end{array}$$

If $\texttt{ins}$ is a $\texttt{blockcall}\ mp_1 \cdots mp_l$ then

$$\begin{aligned}
[\![\texttt{ins}]\!]^{\sqcup_{j\in J}\iota_j} &= (\sqcup_{j\in J}\iota_j)(P(mp_1)) \cup \cdots \cup (\sqcup_{j\in J}\iota_j)(P(mp_l)) \\
&= \Big(\cup_{j\in J}\, \iota_j(P(mp_1))\Big) \cup \cdots \cup \Big(\cup_{j\in J}\, \iota_j(P(mp_l))\Big) \\
&= \cup_{j\in J}\Big(\iota_j(P(mp_1)) \cup \cdots \cup \iota_j(P(mp_l))\Big) \\
&= \cup_{j\in J}\, [\![\texttt{ins}]\!]^{\iota_j}\,.
\end{aligned}$$

□

We can now prove Proposition 1:

Proof. Let $\{\iota_j\}_{j\in J} \subseteq I$ with $J \subseteq \mathbb{N}$. We prove that

$$T_P(\sqcup_{j\in J}\iota_j)(b) = (\sqcup_{j\in J} T_P(\iota_j))(b)$$

for all blocks b.

Let first b be $\boxed{\begin{smallmatrix}\texttt{ins}_1\\ \cdots \\ \texttt{ins}_k\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m\end{smallmatrix}$ with $k > 0$ and $m > 0$ (the cases when $k = 0$ or $m = 0$ are considered later). We have:

$$\begin{aligned}
&T_P(\sqcup_{j\in J}\iota_j)(b) = [\![b]\!]^{\sqcup_{j\in J}\iota_j} \\
&= [\![\texttt{ins}_1]\!]^{\sqcup_{j\in J}\iota_j}\,;\cdots;\,[\![\texttt{ins}_n]\!]^{\sqcup_{j\in J}\iota_j}\,;((\sqcup_{j\in J}\iota_j)(b_1) \cup \cdots \cup (\sqcup_{j\in J}\iota_j)(b_m))
\end{aligned}$$

which by Lemma 1 is equal to

$$\bigcup_{j\in J} [\![\texttt{ins}_1]\!]^{\iota_j}\,;\cdots;\,\bigcup_{j\in J} [\![\texttt{ins}_n]\!]^{\iota_j}\,;\left(\bigcup_{j\in J}(\iota_j(b_1)) \cup \cdots \cup \bigcup_{j\in J}(\iota_j(b_m))\right) \tag{7}$$

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, Equation (7) can be rewritten into

$$\begin{aligned}
&\bigcup_{j\in J}([\![\texttt{ins}_1]\!]^{\iota_j}\,;\cdots;\,[\![\texttt{ins}_n]\!]^{\iota_j}\,;(\iota_j(b_1) \cup \cdots \cup \iota_j(b_m))) \\
&= \bigcup_{j\in J} [\![b]\!]^{\iota_j} = \bigcup_{j\in J}(T_P(\iota_j)(b)) \\
&= (\sqcup_{j\in J} T_P(\iota_j))\,(b)\,.
\end{aligned}$$

The cases when $k = 0$ or $m = 0$ follow similarly: when $k = 0$ we remove the interpretations of the instructions $\texttt{ins}_1, \ldots, \texttt{ins}_m$; when $m = 0$ we remove the interpretations of the blocks $b_1, \ldots, b_m$. □

9.3 Proof of Theorem 2

Lemma 2. *Let $\langle b \,\|\, \sigma\rangle$ be a state such that $\langle b \,\|\, \sigma\rangle \not\Rightarrow$. Then b has the form $\boxed{\ }^{\ell}$.*

Proof. The proof follows from these remarks:

- b cannot have the form $\boxed{\begin{smallmatrix}\mathtt{ins_1}\\ \cdots \\ \mathtt{ins_n}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$ with $n \neq 0$, otherwise one of the rules (1), (2) and (5) of Definition 5 would be applicable to $\langle b \,\|\, \sigma\rangle$. Note that when $\mathtt{ins_1}$ is a `call` then rule (2) is applicable since the Java bytecode is verifiable [9].
- b cannot have the form $\boxed{\ }^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$ with $m \neq 0$ and $k \in \mathbb{N}$, otherwise rule (4) of Definition 5 would be applicable to $\langle b \,\|\, \sigma\rangle$.
- b cannot have the form $\boxed{\ }^{mk} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$ with $m \neq 0$ and $k \in \mathbb{N}$ since magic blocks have no successors, accordingly with Definition 1.

□

Proposition 4. *Let b a block (not necessarily of P) and σ_{in} an initial state for b. Then,*

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

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

if
$$\langle b \,\|\, \sigma_{in}\rangle \Rightarrow^{n} \langle b' \,\|\, \sigma_{out}\rangle \not\Rightarrow$$
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
$$\langle b \,\|\, \sigma_{in}\rangle \Rightarrow^{0} \langle b' \,\|\, \sigma_{out}\rangle \not\Rightarrow\ .$$
Then, $b' = b$ and $\sigma_{out} = \sigma_{in}$. So, by Lemma 2, 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
$$\langle b \,\|\, \sigma_{in}\rangle \Rightarrow^{n+1} \langle b' \,\|\, \sigma_{out}\rangle \not\Rightarrow\ .$$
Then, $\langle b \,\|\, \sigma_{in}\rangle \Rightarrow a \Rightarrow^{n} \langle b' \,\|\, \sigma_{out}\rangle \not\Rightarrow$. Let us consider the rule of Definition 5 that is used in the first derivation step.

1. If rule (1) is used then

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

Let $b_a = \boxed{rest}^{\ell} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array}$. By Definition 10, $[\![b]\!]^{\mathcal{D}_P} = [\![\texttt{ins}]\!]^{\mathcal{D}_P}; [\![b_a]\!]^{\mathcal{D}_P}$ with, 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

$$b = \boxed{\begin{array}{c}\texttt{call } \kappa.m(\boldsymbol{\tau}){:}t\\ rest\end{array}}^{\ell} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \text{ and } a = \langle b_1 \parallel \sigma_1\rangle :: \langle \boxed{rest}^{\ell} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \parallel \langle l \parallel s \parallel \mu\rangle\rangle$$

where b_1 is the starting block of method $\kappa.m(\boldsymbol{\tau}) : t$, $\sigma_{in} = \langle l \parallel pars :: s \parallel \mu\rangle$ and $\sigma_1 = (makescope\ \kappa.m(\boldsymbol{\tau}) : t)(\sigma_{in})$. Notice that the derivation from a to $\langle b' \parallel \sigma_{out}\rangle$ has the form

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

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

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

Consequently, by Definition 8,

$$\langle l \parallel vs :: s \parallel \mu'\rangle = (extend\ \kappa.m(\boldsymbol{\tau}) : t)(\delta_1)(\sigma_{in}) \ . \tag{8}$$

Let $b_a = \boxed{\mathit{rest}}^{\ell} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix}$. By Definition 10,

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

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

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

Hence, there exists $\delta_2 \in [\![b_a]\!]^{\mathcal{D}_P}$ such that $\delta_2(\langle l \,\|\, vs :: s \,\|\, \mu' \rangle)$ is defined and $\sigma_{out} = \delta_2(\langle l \,\|\, vs :: s \,\|\, \mu' \rangle)$. So, by (8), we have

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

Notice that, by Definition 11, $[\![b_1]\!]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(b_1)$. 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 $(\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(\delta_1) \in (\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(\mathcal{D}_P(b_1))$. Therefore,

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

i.e., $(\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(\delta_1); \delta_2 \in [\![b]\!]^{\mathcal{D}_P}$. 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 $\langle b \,\|\, \sigma_{in} \rangle$, whose length is equal to 1.
4. If rule (4) is used then

$$b = \boxed{}^{k} \rightrightarrows \begin{smallmatrix} b_1 \\ \cdots \\ b_m \end{smallmatrix} \quad \text{and} \quad a = \langle b_i \,\|\, \sigma_{in} \rangle$$

where $i \in \{1, \ldots, m\}$. Notice that, by Definition 10,

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

Moreover, for each $j \in \{1, \ldots, m\}$, we have $[\![b_j]\!]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(b_j)$ by Definition 11. As $\mathcal{D}_P$ is the least fixpoint of T_p, 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 $\langle b_i \,\|\, \sigma_{in} \rangle \Rightarrow^n \langle b' \,\|\, \sigma_{out} \rangle$, 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,

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

5. If rule (5) is used then

$$b = \boxed{\begin{array}{l}\texttt{blockcall}\ mp_1 \cdots mp_l \\ rest\end{array}}^{mk} \quad \text{and} \quad a = \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle$$

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

$$\begin{aligned} & \langle P(mp_i) \parallel \sigma_{in} \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle \\ \Rightarrow^* & \langle \square^{mp_i} \parallel \sigma \rangle :: \langle \boxed{rest}^{mk} \parallel \sigma_{in} \rangle \\ \underset{(6)}{\Rightarrow} & \langle \boxed{rest}^{mk} \parallel \sigma \rangle \\ \Rightarrow^* & \langle b' \parallel \sigma_{out} \rangle \end{aligned}$$

where $\langle P(mp_i) \parallel \sigma_{in} \rangle \Rightarrow^* \langle \square^{mp_i} \parallel \sigma \rangle$ in less than n steps. Notice that $\langle \square^{mp_i} \parallel \sigma \rangle \not\Rightarrow$. 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 $b_a = \boxed{rest}^{mk}$. By Definition 10,

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

Moreover, for each $j \in \{1, \ldots, l\}$, $[\![P(mp_j)]\!]^{\mathcal{D}_P} = T_P(\mathcal{D}_P)(P(mp_j))$ by Definition 11. 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,

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

6. Rule (6) cannot be used because it requires a starting activation stack whose length is at least equal to 2. Here, the starting activation stack is $\langle b \,\|\, \sigma_{in}\rangle$, whose length is equal to 1.

□

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

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

Proof. Notice that, by Definition 12, $\mathcal{D}_P = \sqcup_{i\geq 0} T_P^i$. 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

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

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 $\boxed{\begin{smallmatrix}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_k\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix}$ with $k \geq 0$ and $m \geq 0$.

– (Basis) We prove that $Prop_{\supseteq}(0)$ holds.
  - If $m \neq 0$ or if there is a $i \in \{1, \ldots, k\}$ 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 10. 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 10, we have

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

Moreover, by Definition 5,

$$\begin{aligned}
\langle \underbrace{\boxed{\begin{smallmatrix}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_k\end{smallmatrix}}^{\ell}}_{b} \,\|\, \sigma_{in}\rangle &\underset{(1)}{\Rightarrow} \langle \boxed{\begin{smallmatrix}\texttt{ins}_2\\ \cdots\\ \texttt{ins}_k\end{smallmatrix}}^{\ell} \,\|\, ins_1(\sigma_{in})\rangle \\
&\underset{(1)}{\Rightarrow} \langle \boxed{\begin{smallmatrix}\texttt{ins}_3\\ \cdots\\ \texttt{ins}_k\end{smallmatrix}}^{\ell} \,\|\, ins_2(ins_1(\sigma_{in}))\rangle \\
&\ \ \vdots \\
&\underset{(1)}{\Rightarrow} \langle \square^{\ell} \,\|\, ins_k(\cdots ins_1(\sigma_{in})\cdots)\rangle \\
&\not\Rightarrow
\end{aligned}$$

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

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

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

– (Induction) Suppose that $Prop_{\supseteq}(n)$ holds. We prove that $Prop_{\supseteq}(n+1)$ 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 10, we have

$$[\![b]\!]^{T_P^{n+1}} = [\![\mathtt{ins_1}]\!]^{T_P^{n+1}} ; \cdots ; [\![\mathtt{ins_k}]\!]^{T_P^{n+1}} ; (T_P^{n+1}(b_1) \cup \cdots \cup T_P^{n+1}(b_m)) \ .$$

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 11. Hence,

$$[\![b]\!]^{T_P^{n+1}} = [\![\mathtt{ins_1}]\!]^{T_P^{n+1}} ; \cdots ; [\![\mathtt{ins_k}]\!]^{T_P^{n+1}} ; ([\![b_1]\!]^{T_P^n} \cup \cdots \cup [\![b_m]\!]^{T_P^n}) \ .$$

Hence there exist $\delta_1 \in [\![\mathtt{ins_1}]\!]^{T_P^{n+1}}, \ldots, \delta_k \in [\![\mathtt{ins_k}]\!]^{T_P^{n+1}}$, $i \in \{1, \ldots, m\}$ 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 10,

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

Moreover, by Definition 5,

$$\begin{aligned}
\langle \underbrace{\boxed{\begin{smallmatrix}\mathtt{ins_1}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix}}_{b} \parallel \sigma_{in} \rangle &\underset{(1)}{\Rightarrow} \langle \boxed{\begin{smallmatrix}\mathtt{ins_2}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel ins_1(\sigma_{in}) \rangle \\
&\underset{(1)}{\Rightarrow} \langle \boxed{\begin{smallmatrix}\mathtt{ins_3}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel ins_2(ins_1(\sigma_{in})) \rangle \\
&\ \ \vdots \\
&\underset{(1)}{\Rightarrow} \langle \square^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel ins_k(\cdots ins_1(\sigma_{in}) \cdots) \rangle \\
&\underset{(4)}{\Rightarrow} \langle \, b_i \parallel ins_k(\cdots ins_1(\sigma_{in}) \cdots) \rangle \ .
\end{aligned}$$

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

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

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 $\mathtt{call}\ \kappa.m(\boldsymbol{\tau}) : t$ and, by Definition 10, $[\![\mathtt{ins}_1]\!]^{T_P^{n+1}} = (\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(T_P^{n+1}(b_{\kappa.m(\boldsymbol{\tau}):t}))$ (where $b_{\kappa.m(\boldsymbol{\tau}):t}$ is the block where $\kappa.m(\boldsymbol{\tau}) : t$ starts) *i.e.,*

$$\begin{aligned}[\![\mathtt{ins}_1]\!]^{T_P^{n+1}} &= (\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(T_P(T_P^n)(b_{\kappa.m(\boldsymbol{\tau}):t})) \\ &= (\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)([\![b_{\kappa.m(\boldsymbol{\tau}):t}]\!]^{T_P^n})\ .\end{aligned}$$

Moreover, by Definition 5, $\sigma_{in} = \langle l \,\|\, \mathit{pars} :: s \,\|\, \mu\rangle$ and

$$\langle \underbrace{\boxed{\begin{matrix}\mathtt{ins}_1\\ \cdots\\ \mathtt{ins}_k\end{matrix}}^{\ell} \rightrightarrows \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix}}_{b} \,\|\, \sigma_{in}\rangle \underset{(2)}{\Rightarrow} \langle b_{\kappa.m(\boldsymbol{\tau}):t} \,\|\, \sigma'\rangle :: \langle \boxed{\begin{matrix}\mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_k\end{matrix}}^{\ell} \rightrightarrows \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \,\|\langle l \,\|\, s \,\|\, \mu\rangle\rangle \qquad (9)$$

where $\sigma' = (\mathit{makescope}\ \kappa.m(\boldsymbol{\tau}) : t)(\sigma_{in})$. As $\delta_1 \in [\![\mathtt{ins}_1]\!]^{T_P^{n+1}}$, there exists $\delta_1' \in [\![b_{\kappa.m(\boldsymbol{\tau}):t}]\!]^{T_P^n}$ such that $\delta_1 = (\mathit{extend}\ \kappa.m(\boldsymbol{\tau}) : t)(\delta_1')$. As $\delta_1(\sigma_{in})$ is defined (because $\delta(\sigma_{in})$ is defined) we have (by Definition 8)

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

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

$$\langle b_{\kappa.m(\boldsymbol{\tau}):t} \,\|\, \sigma'\rangle \Rightarrow^* \langle b' \,\|\langle l' \,\|\, vs \,\|\, \mu'\rangle\rangle \not\Rightarrow\ .$$

Note that b' is not a magic block (because no magic block is reachable from a non-magic blocks of a method, like $b_{\kappa.m(\boldsymbol{\tau}):t}$) and that, by Lemma 2, b' has the form $\boxed{}^{k'}$. Therefore, by (9) and Definition 5, we have

$$\begin{aligned}\langle\, b \,\|\, \sigma_{in}\rangle &\underset{(2)}{\Rightarrow} \langle b_{\kappa.m(\boldsymbol{\tau}):t} \,\|\, \sigma'\rangle :: \langle \boxed{\begin{matrix}\mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_k\end{matrix}}^{\ell} \rightrightarrows \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \,\|\langle l \,\|\, s \,\|\, \mu\rangle\rangle \\ &\Rightarrow^* \langle \boxed{}^{k'} \,\|\langle l' \,\|\, vs \,\|\, \mu'\rangle\rangle :: \langle \boxed{\begin{matrix}\mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_k\end{matrix}}^{\ell} \rightrightarrows \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \,\|\langle l \,\|\, s \,\|\, \mu\rangle\rangle \\ &\underset{(3)}{\Rightarrow} \langle \boxed{\begin{matrix}\mathtt{ins}_2\\ \cdots\\ \mathtt{ins}_k\end{matrix}}^{\ell} \rightrightarrows \begin{matrix}b_1\\ \cdots\\ b_m\end{matrix} \,\|\langle l \,\|\, vs :: s \,\|\, \mu'\rangle\rangle\end{aligned}$$

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

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

3. Suppose that b starts with a `blockcall` and then consists of instructions that are not a `call` nor a `blockcall`. Then, $\mathtt{ins_1}$ has the form `blockcall` $mp_1 \cdots mp_l$ and, by Definition 10,

$$\begin{aligned}\llbracket \mathtt{ins_1} \rrbracket^{T_P^{n+1}} &= T_P^{n+1}(P(mp_1)) \cup \cdots \cup T_P^{n+1}(P(mp_l)) \\ &= T_P(T_P^n)(P(mp_1)) \cup \cdots \cup T_P(T_P^n)(P(mp_l)) \\ &= \llbracket P(mp_1) \rrbracket^{T_P^n} \cup \cdots \cup \llbracket P(mp_l) \rrbracket^{T_P^n} \ .\end{aligned}$$

As $\delta_1 \in \llbracket \mathtt{ins_1} \rrbracket^{T_P^{n+1}}$, there is an $i \in \{1, \ldots, l\}$ such that $\delta_1 \in \llbracket P(mp_i) \rrbracket^{T_P^n}$. By Definition 5, we have

$$\langle \underbrace{\boxed{\begin{smallmatrix}\mathtt{ins_1}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel \sigma_{in}\rangle}_{b} \underset{(5)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in}\rangle :: \langle \boxed{\begin{smallmatrix}\mathtt{ins_2}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel \sigma_{in}\rangle \ . \qquad (10)$$

As $\delta_1(\sigma_{in})$ is defined (because $\delta(\sigma_{in})$ is defined) and $\delta_1 \in \llbracket P(mp_i) \rrbracket^{T_P^n}$, 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 5, b' is a magic block labelled with mp_i and, by Lemma 2, b' has the form $\boxed{\ }^{mp_i}$. Therefore, by (10) and Definition 5, we have

$$\begin{aligned}\langle b \parallel \sigma_{in}\rangle &\underset{(5)}{\Rightarrow} \langle P(mp_i) \parallel \sigma_{in}\rangle :: \langle \boxed{\begin{smallmatrix}\mathtt{ins_2}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel \sigma_{in}\rangle \\ &\Rightarrow^* \langle \boxed{\ }^{mp_i} \parallel \delta_1(\sigma_{in})\rangle :: \langle \boxed{\begin{smallmatrix}\mathtt{ins_2}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel \sigma_{in}\rangle \\ &\underset{(6)}{\Rightarrow} \langle \boxed{\begin{smallmatrix}\mathtt{ins_2}\\ \cdots \\ \mathtt{ins_k}\end{smallmatrix}}^{\ell} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots \\ b_m\end{smallmatrix} \parallel \delta_1(\sigma_{in})\rangle \ .\end{aligned}$$

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`. Notice that 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. □