A Provably Correct Stackless Intermediate Representation For Java Bytecode

Delphine Demange, Thomas Jensen, David Pichardie

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Provably Correct Stackless Intermediate Representation for Java Bytecode

Delphine Demange — Thomas Jensen — David Pichardie

N° 7021

Juillet 2009

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7021--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

A Provably Correct Stackless Intermediate Representation for Java Bytecode

Delphine Demange*, Thomas Jensen†, David Pichardie‡

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet Celtique

Rapport de recherche n° 7021 — Juillet 2009 — 56 pages

Abstract: The Java virtual machine executes stack-based bytecode. The intensive use of an operand stack has been identified as a major obstacle for static analysis and it is now common for static analysis tools to manipulate a stackless intermediate representation (IR) of bytecode programs. Several algorithms have been proposed to achieve such a transformation, but only little attention has been paid to their formal semantic properties.

This work provides such a bytecode transformation, describes its semantic correctness and evaluates its performance with respect to the transformation time, the compactness of the obtained code and the impact on static analysis precision.

We provide the semantic foundations for proving that an initial program and its IR behave similarly, in particular with respect to object creation and throwing of exceptions. We formalize a notion of semantic preservation: an initial program and its IR have similar execution traces. Since the transformation does not preserve the object allocation order, the similarity between traces is defined using a relation between the two heaps. The correctness of this transformation is proved with respect to this semantic criterion.

Key-words: Static analysis, Bytecode languages, Program transformation

Work partially supported by EU project MOBIUS

* Université de Rennes 1, Rennes, France
† CNRS, Rennes, France
‡ INRIA, Centre Rennes - Bretagne Atlantique, Rennes

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Une Représentation Intermédiaire Basée Registre Prouvée Correcte pour le Bytecode Java

Résumé : La machine virtuelle Java exécute des programmes bytecodes en utilisant une pile d'opérandes. Cet usage intensif d'une pile d'opérandes a été identifié comme un obstacle majeur pour l'analyse statique. Il est désormais courant que les outils d'analyse statique modernes aient recours à une transformation préliminaire qui retire cet usage. Plusieurs algorithmes ont été proposés pour réaliser cette transformation du bytecode vers une représentation intermédiaire (IR), mais très peu d'attention a été portée jusque là à leurs propriétés sémantiques. Ce travail spécifie une telle transformation et propose les fondations sémantiques pour prouver qu'un programme bytecode initial et sa représentation intermédiaire se comportent de façons similaires, en particulier vis à vis de l'initialisation des objets et des lancements d'exceptions. La transformation est basée sur une execution symbolique du bytecode utilisant une pile d'opérandes abstraits. Chaque instruction bytecode modifie la pile symbolique, et donne lieu à la génération d'instructions du langage d'IR. Nous formalisons une notion de préservation sémantique : un programme et son IR ont des traces d'exécution similaires. La transformation ne conservant pas l'ordre d'allocation des objets, cette similarité de trace est exprimée par une relation de correspondance sur les tas. Enfin, la correction sémantique de la transformation est prouvée relativement à ce critère.

Mots-clés : Analyse statique, Langage Bytecode, Transformation de programmes

Chapter 1

Introduction

Java bytecode is sometimes considered simpler to analyse than its source counterpart, due to the absence of features such as inner classes and generics. However several features still complicate the static analysis of a bytecode program and several bytecode optimization and analysis tools work on an *intermediate representation* (IR) of bytecode that makes analyses simpler [VRCG+99, BCF+99]. Using such transformations may simplify the work of the analyser but the overall correctness of the analysis now becomes dependent on the semantic preservation properties of the transformation. Surprisingly, the semantic foundations of these bytecode transformations have received little attention. In this paper, we propose a transformation with a formal correctness proof.

Semantic correctness is particularly crucial when an analysis forms part of the security defense line, as is the case with Java's bytecode verifier (BCV). One of our aim is to provide a transformation that can be used to integrate other static analyses into an "extended bytecode verifier" akin to the stack map-based lightweight bytecode verifier proposed by Rose [Ros03]. For this to work, the transformation must be efficient so a requirement to our transformation algorithm is that it must work in one pass over the bytecode.

The JVM is a stack-based virtual machine and the *intensive use of the operand stack* may make it difficult to adapt standard static analysis techniques that have been first designed for more standard (variable-based) 3-address codes. As noticed by Logozzo and Fähndrich [LF08], a naive translation from a stack-based code to 3-address code may result in an explosion of temporary variables, which in turn may dramatically affect the precision of non-relational static analyses (such as intervals) and render some of the more costly analyses (such as polyhedral analysis) infeasible. Thus, the transformation should improve analysis precision and should keep the number of extra temporary variables at a reasonable level.

Object creation is another feature which is difficult to track because it is done in two distinct steps (i) raw object allocation, (ii) constructor call. References to uninitialized objects are frequently pushed and duplicated on the operand stack, which makes it difficult for an analysis to recover this sequence of actions. The BCV not only enforces type safety of bytecode programs but also a complex *object initialization* property: an object cannot be used before an adequate constructor has been called on it. The BCV is able to verify this by tracking aliases of uninitialized objects in the operand stack, but initialization information is lost for subsequent static analyses.

This paper provides a semantically sound, provably correct specification of a transformation of bytecode into an intermediate representations (IR) of bytecode that (i) removes the use of the operand stack and rebuilds expressions, (ii) rebuilds the initialization chain of an object with a dedicated instruction `x := new C(arg1, arg2, ...)`. A difficulty for such a bytecode transformation is the wealth

of dynamic checks used to ensure intrinsic properties of the Java execution model, such as absence of null-pointer dereferencings, out-of-bounds array accesses, *etc*... The consequence is that many instructions may rise different kinds of exception. Static analyses must take this mechanism into account but their result may be incorrect if the transformation permutes the throwing of exceptions that could arise during the program execution. An additional feature of the transformation formalized here is that it makes explicit the throwing of certain exceptions and takes care of preserving their order.

Figure 1.1 presents an example of a Java program (Figure 1.1(a)) that illustrates these issues. Its corresponding bytecode version (Figure 1.1(c)) shows the standard object initialization scheme: an expression like `new A()` is compiled into the sequence of lines [5; 6; 7]. A new object of class `A` is first allocated in the heap and its address is pushed on top of the operand stack. The address is then duplicated on the stack by the instruction `dup` and the non-virtual method `A()` is called, consuming the top of the stack. The copy is left on the top of the stack and represents from now an *initialized* object. This initialization by side-effect, is particularly challenging for the BCV [FM99] which has to keep track of the alias between not-yet-initialized references on the stack. Using a similar approach, we are able to *fold* the two instructions of object allocation and constructor call into a single IR instruction. Figure 1.1(b) shows a first attempt of such a fusion.

```
B f(int x, int y) {
  return(new B(x/y, new A()));
}
```

(a) source function

```
B f(x, y);
  0 : t1 := new A();
  1 : t2 := new B(x/y, t1);
  2 : vreturn t2;
```

(b) BIR function (not semantics-preserving)

```
B f(x, y);
  0 : new B
  1 : dup
  2 : load y
  3 : load x
  4 : div
  5 : new A
  6 : dup
  7 : constructor A
  8 : constructor B
  9 : vreturn
```

(c) BC function

```
B f(x, y);
  0 : mayinit B;
  1 : nop;
  2 : nop;
  3 : nop;
  4 : notzero y;
  5 : mayinit A;
  6 : nop;
  7 : t1 := new A();
  8 : t2 := new B(x/y, t1);
  9 : vreturn t2;
```

(d) BIR function (semantics-preserving)

```
0 : []
1 : [UR_0^B]
2 : [UR_0^B::UR_0^B]
3 : [y::UR_0^B::UR_0^B]
4 : [x::y::UR_0^B::UR_0^B]
5 : [x/y::UR_0^B::UR_0^B]
6 : [UR_5^A::x/y::UR_0^B::UR_0^B]
7 : [UR_5^A::UR_5^A::x/y::UR_0^B::UR_0^B]
8 : [t1::x/y::UR_0^B::UR_0^B]
9 : [t2]
```

(e) Symbolic stack

Figure 1.1: Example of source code, bytecode and two possible transformations

However, on this example, we generated side-effect free expressions in a naive way and have therefore changed several semantic properties. First, the program does not respect the allocation order. This fact is unavoidable if we want to keep side-effect free expressions and still re-build object constructions. The allocation order may have a functional impact because of the static initializer `A.⟨clinit⟩` that may be called when reaching an instruction `new A`. In Figure 1.1(b) this order is not preserved since `A.⟨clinit⟩` may be called before `B.⟨clinit⟩` while the bytecode program follows an inverse order. In Figure 1.1(d) this problem is solved using a specific instruction `mayinit A` that makes explicit the potential call to a static initializer. The second major semantic problem of Figure 1.1(b) is that the program does not respect the exception throwing order of the bytecode version. In Figure 1.1(b) the call to `A()` may appear before the `DivisionByZero` exception that may be raised

when evaluating `x/y` if the value of `y` is zero. The program in Figure 1.1(d) solves this problem using a specific instruction `notzero y` that explicitly checks that `y` is non-zero and raises a `DivisionByZero` exception if `y` evaluates to zero.

The algorithm presented in Chapter 5 and proved correct in Chapter 6 takes care of these pitfalls. The source (BC) and IR (BIR) languages are presented in Chapters 3 and 4. We present a prototype implementation and evaluate its performance in Chapter 7.

Chapter 2

Related work

Many Java bytecode optimization and analysis tools work on an IR of bytecode that make its analysis much simpler. Soot [VRCG+99] is a Java bytecode optimization framework providing three IR: Baf, Jimple and Grimp. Optimizing Java bytecode consists in successively translating bytecode into Baf, Jimple, and Grimp, and then back to bytecode, while performing diverse optimizations on each IR. Baf is a fully typed, stack-based language. Jimple is a typed stackless 3-address code. Grimp is a stackless code with tree expressions, obtained by collapsing 3-address Jimple instructions. Type inference of Jimple instructions has been addressed in [GHM00]. The static analyser Costa [AAG+07] relies on an IR similar to Jimple by removing explicit uses of the operand stack using additional local variables. Contrary to Soot, it does not address the problem of generating typed instructions. The transformation algorithm proposed in this paper follows the symbolic evaluation technique used by Whaley [Wha99] for the high intermediate representation of the Jalapeño Optimizing Compiler [BCF+99] which is now part of the Jikes virtual machine [Pro]. Whereas Soot provided both stack-based and register-based IR, Jalapeño directly provides register-based IRs, that can contain additional information (e.g a control flow graph). The language provides explicit check operators for common run-time exceptions (`null_check`, `bound_check`...), so that they can be easily moved or eliminated by optimizations. This paper pushes the technique further, generating tree expressions in conditional branchings and folding constructors.

Unlike all works cited above, our transformation does not require iterating on the method code. Still, the number of generated variables keeps small in practice (Chapter 7). All these previous works have been mainly concerned with the construction of effective and powerful tools but, as far as we know, little attention has been paid to the formal semantic properties that are ensured by these transformations.

The use of a symbolic evaluation of the operand stack to recover some tree expressions in a bytecode program has been employed in several context of Java Bytecode analysis. The technique was already used in one of the first Sun Just-In-Time compiler [CFM+97] for direct translation of bytecode to machine instructions. Xi and Xia propose a dependent type system for array bound check elimination. It uses symbolic expressions to type operand stacks with *singleton* types in order to recover relations between lengths of arrays and index expressions used to access them. Besson *et al* [BJP06], and independently Wildmoser *et al* [WCN05], propose an extended interval analysis that verifies there is no out-of-bound array accesses in programs using symbolic decompilation. Besson *et al* give an example that shows how the precision of the standard interval analysis is enhanced by including syntactic expressions in the abstract domain. Barthe *et al* [BKPSF08] also use a symbolic

manipulation for the relational analysis of a simple bytecode language and prove it has the same precision as a similar analysis at source level.

Symbolic evaluation has also been used in translation validation, an alternative approach to the direct proof of correctness of code transformations used here, consisting of *post-hoc* checking of the semantic equivalence of a code and its transformation. Leroy and Tristan [TL08] have proved formally the correctness of a symbolic evaluation used to validate the equality of values of expressions and local registers after list and trace scheduling. This transformation does not target a stack-based language and does not aim at reconstructing method calls. The symbolic evaluation also differs in that it is only concerned with computing the input-output semantics of basic blocks of low-level code. It has been formally verified in Coq.

Chapter 3

The source language: BC

In our work, we consider a restricted version of the Java Bytecode language. Difficulties that are inherent to the object oriented paradigm (e.g. object initialization) have to be addressed before considering e.g. multi-threading. In this chapter, we present the formalization of the subset of Java Bytecode language we consider: BC. We first describe its syntax in Section 3.1. Then, we propose an observational, operational semantics for BC.

3.1 Syntax of BC

The set of bytecodes we consider is given in Figure 3.1. We describe each of them briefly and justify our choices before further formalizing their semantics in Section 3.2.

var	::=		*variables* :
		$\mathtt{x} \mid \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots$	
		`this`	
oper	::=		*operands* :
		$\mathtt{c} \mid \mathtt{c}' \ldots \mid \mathtt{null}$	*constant*
		var	*variable*
		$\mathtt{pc} \mid \mathtt{pc}' \ldots$	*program counter*
		$\mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \ldots$	*class name*
		$\mathtt{f} \mid \mathtt{f}' \mid \ldots$	*field name*
		$\mathtt{m} \mid \mathtt{m}' \mid \ldots$	*method name*

instr	::=	*instructions* :
		`nop` | `push c` | `pop` | `dup` | `add` | `div`
		| `load x` | `store x`
		| `new C` | `constructor C`
		| `getfield f` | `putfield f`
		| `invokevirtual C.m`
		| `if pc` | `goto pc`
		| `vreturn` | `return`

Figure 3.1: Operands and instructions of BC

BC provides simple stack operations: `push c` pushes the constant `c` (who might be `null`) onto the stack. `pop` pops the top element off the stack. `dup` duplicates the top element of the stack. All operands are the same size: we hence avoid the problem of typing `dup` and `pop`. The `swap` bytecode is not considered as it would be treated similarly to `dup` or `pop`. Only two binary arithmetic operators over values is available, the addition `add` and the division `div`: other operators (i.e. substraction, multiplication) would be handled similarely. We also restrict branching instructions to only one, `if pc`: the control flow jumps to the label `pc` if the top element of the stack is zero. Others jumps (e.g. `ifle pc`) would be treated similarely. For the same reason, we choose not to include switch tables in BC.

mcode ::=		*method code* :
	`pc : return`	*method end*
	`| pc : vreturn`	*method end*
	`| (pc :` $instr \backslash \{\texttt{return}, \texttt{vreturn}\})^* mcode$	*instr list*
r ::=		*return type* :
	`void | v`	*void/non void*
msig ::=		*signature* :
	`C` r `m(`$var \ldots var : var \ldots var$`)`	
method ::=		BC *method* :
	msig ; *mcode*	

class	::=	BC *class* :
		`C {`
		$\texttt{f}_1 \ldots \texttt{f}_n$
		$method_1$
		...
		$method_m$ `}`
prog	::=	BC *program* :
		$class_1 \ldots$
		$class_p$

Figure 3.2: BC syntax: methods, classes, programs

The value of a local variable `x` can be pushed onto the stack with the instruction `load x`. The special variable `this` denotes the current object (within a method). We do not distinguish between loading an integer or reference variable, as it is in the real bytecode language: we suppose the bytecode passes the BCV, hence variables are correctly used. The same applies to bytecode `store x` that stores the value of the stack top element in the local variable `x`.

A new object of class `C` is allocated in the heap by the instruction `new C`. Then, it has to be initialized by calling its constructor `constructor C`. The super constructor is called through the same BC instruction, `constructor B`, where `B` is the direct super class of `C`. We will see in Chapter 5 that both cases have to be distinguished during the transformation. The constructor of a class is supposed to be unique. In real Java bytecode, `constructor B` corresponds to `invokespecial B.` < `init` >, but instruction `invokespecial` is used for many other cases. Our dedicated bytecode focus on its role for object constructors. We do not consider static fields or static methods. Class fields are read and assigned with `getfield f` and `putfield f` (we suppose the resolution of field class has been done). A method `m` is called on an object with `invokevirtual C.m`. Finally, methods can either return a value (`vreturn`) or not (`return`). Real bytecode provides one instruction per return value type, but for the reason given above, BC does not. For sake of simplicity, we do not use any constant pool. Hence constants, variables, classes, fields and method identifiers will be denoted by strings – their potential identifier at Java source level (every identifier is unique).

Figure 3.2 gives the syntax of BC. A BC method is made of its signature (class name, method name, value or void return, formal paramaters and local variables) together with its code, a list of BC instructions, indexed by a program counter `pc` starting from `0`. In the following, $instrAt_P(m, pc)$ denotes the instruction at `pc` in the method `m` of the program P. A BC class is made of its name, its fields names and its methods. Finally, a BC program is a set of BC classes. In the next section, we present the operational, observational semantics we defined for BC. Here is an example of BC program. The method `main` computes the sum of its two arguments, stores the result in the local variable `z`, and returns the sum of 1 and `z`.

```
Test {
  f1 f2
  Test v main (x, y : z) {
     0:   load x
     1:   load y
     2:   add
     3:   store z
     4:   push 1
     5:   load z
     6:   add
     7:   vreturn
  }
}
```

3.2 Semantics of BC

3.2.1 Semantic domains

Semantic domains are given in Figure 3.3. A value is either an integer, a reference or the special value *Null*.

$$
\begin{array}{rcl}
\mathit{Value} & = & (\mathit{Num}\ n),\ n \in \mathbb{Z} \\
 & & (\mathit{Ref}\ r),\ r \in \mathit{Reference} \\
 & & \mathit{Null} \\
\overline{\mathit{Value}} & = & \mathit{Value} \cup \{\mathit{Void}\} \\
\mathit{InitTag} & = & \mathbb{C} \cup \widetilde{\mathbb{C}} \cup \widetilde{\mathbb{C}}_{\mathbb{N}} \\
\mathit{Object} & = & (\mathbb{F} \rightarrow \mathit{Value})_{\mathit{InitTag}} \\
\mathit{Heap} & = & \mathit{Reference} \hookrightarrow \mathit{Object}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{Stack} & = & \mathit{Value}^* \\
\mathit{Env}_{\mathrm{BC}} & = & \mathit{var} \hookrightarrow \mathit{Value} \\
\mathit{Frame}_{\mathrm{BC}} & = & \mathbb{M} \times \mathbb{N} \times \mathit{Env}_{\mathrm{BC}} \times \mathit{Stack} \\
\mathit{State}_{\mathrm{BC}} & = & (\mathit{Heap} \times \mathit{Frame}) \\
 & & \cup \left(\mathit{Heap} \times \overline{\mathit{Value}}\right) \\
 & & \cup \left\{ \Omega^{NP}, \Omega^{DZ} \right\} (\mathbb{N} \times \mathbb{M} \times \mathit{Env}_{\mathrm{BC}} \times \mathit{Heap})
\end{array}
$$

Figure 3.3: BC semantic domains

The operand stack is a list of elements of *Value*. Given the set of variable identifiers *var*, that includes the special identifier *this* (denoting the current object), an environment is a partial function from *var* to *Value*. To lighten the notations, we assume in the following that when the variable `x` is accessed in an environment l, written $l(\mathtt{x})$, `x` is as required in the domain of l.

An object is represented as a total function from its fields names $\mathbb{F}$ to values. An initialization status is also attached to every object. Initialization tags were first introduced by Freund and Mitchell [FM03] in order to formalize the object initialization verification pass performed by the Bytecode Verifier. They provide a type system ensuring, amoung other things, that (i) every newly allocated object is initialized before being used and (ii) every constructor, from the declared class of the object up to the `Object` class is called before considering the object as initialized. We further explain initialization tags in a dedicated paragraph below.

The heap is a partial function from non-null references to objects. The special reference *Null* does not point to any object. Each time a new object is allocated in the heap, the partial function is extended accordingly. We do not model any garbage collector and the heap is considered arbitrarily large, or at least sufficiently large for the program execution not to raise `outOfMemory` errors.

Initialization tags Object initialization is a key point in our work. So let us describe what it means for an object to be initialized. In Figure 3.4 is given the life cycle of an object, from its creation to its use. Suppose the direct super class of `C` is class `B` and the direct super class of `B` is `Object` (see class hierarchy on the right). A new object of class `C` is allocated in the heap with `new C` at program counter `pc` in the main method. It is allocated in the heap but still uninitialized. No operation is allowed on this object (no method call on it, nor field access or modification). At some point, the constructor of class `C` is invoked. The initialization process has begun (the object is *being initialized*), and in the JVM specification, from this point, its fields can be written. To simplify, we consider that no field modification is allowed yet (this simplification is also done in [FM03]). The only operation that is allowed on it is to call a super constructor of this object (in the JVM, another constructor of class `C` can be invoked, but in our work constructors are supposed to be unique). Every super constructor in the class hierarchy has to be called, up to the constructor of the `Object` class. As soon as the `Object` constructor is called, the object is considered as initialized: methods can be called on it, and its fields

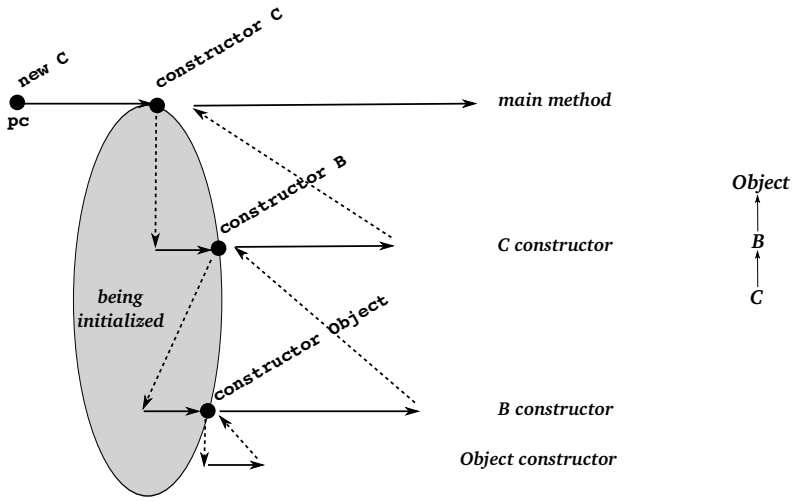

Figure 3.4: Object initialization process: example

can be accessed or modified. In Figure 3.4, the grey area denotes the part of the execution where the created object is considered as being initialized. Before entering this area, the object is said to be uninitialized. After escaping this aera, the object is considered as initialized.

In our work, we suppose that the bytecode passes the Bytecode Verifier. Hence, we know that the constructors chain is correctly called and that each object is initialized before being used. We use the initialization tags introduced in [FM03] to a different end: the semantics preservation needs to distinguish objects that are uninitialized from the other (see Chapter 6). We hence use a similar but simplified version of Freund and Mitchell's initialization tags:

- An object that is just allocated (no constructor has been called yet) has initialization tag $\widetilde{\mathtt{C}}_{\mathtt{pc}}$, meaning that the object was allocated at line `pc` by the instruction `new C`. Keeping track of `pc` strictly identifies uninitialized objects (thanks to the BCV pass, we are ensured that no uninitialized reference is in the stack at backward branchings) and is used to ensure the correctness of the substitutions on abstract stacks during the transformation (see Chapter 6 for further details)
- If the object is being initialized, its tag is $\widetilde{\mathtt{C}}$. The declared class of the object is `C`, and the current constructor is of class `C` or above ([FM03] keeps track of it, but we do not need to)
- As soon as the constructor of the `Object` class is called on a yet unitialized object of class `C`, the initialization tag is updated to `C` and the object is considered as initialized

Execution states There are three kinds of execution states. A *normal execution state* of the current method is made of a heap, the current position in the method code (defined by a program counter in $\mathbb{N}$) and the local memory of the method (local variables and operand stack). A *return state* is made of a heap and a returned value (possibly *Void*). Finally, an *error state* is defined by the program point of the faulty instruction and the current context (heap and environment). We distinguish two kinds of error: divisions by zero (Ω^{DZ}(,pc,l,h)) and null pointer dereferencing (Ω^{NP}(,pc,l,h)). Error states make it possible to state the semantics preservation not only for normal executions. We do not handle exception catching in this work but it will not bring much difficulties thanks to the way error states

are defined. A BC program that passes the BCV may only get stuck on a error state or a return state of the main method.

3.2.2 Semantics

We define the semantics of BC with a view to capturing as much of the program behaviour as possible, so that the semantics preservation property can fits the need of most of static analyses. We hence formalize the BC semantics in terms of a labelled transition system: labels keep track amoung other things of memory modifications, method calls and returns. Hence, the program behaviour is defined in a more observationnal way, and more information than the classical input/output relation is made available.

Basic semantics Before going into more details about labels and all the kinds of transitions we need, let us first present the rules given in Figures 3.5, 3.6 and 3.7 without considering neither transition labels, nor transitions indices. Transitions relate states in $State_{\mathrm{BC}}$ as defined in Figure 3.3. We first describe normal execution rules (Figures 3.5 and 3.6). The general form of a semantic rule is

$$\frac{\begin{array}{c} instrAt_P(m, pc) = instr \\ \text{other conditions} \end{array}}{s \rightarrow s'}$$

where s and s' are execution states, related through the transition relation, which is defined by case analysis on the instruction at the current program point. When needed, other conditions that have to be satisfied (conditions about the content of the stack, of the heap...) are specified below the instruction. Rules for `nop`, `push c`, `pop`, `dup`, `add`, `div` and `load x` are rather simple, and we do not make further comments about them.

In rule for the bytecode `new C`, $newObject(C, h)$ allocates a new object of class C in the heap h, pointed to by (*Ref r*), and returns this new reference. All fields of the object are set to their default value (zero for integers and *Null* for references) by the function *init* and its initialization tag is set to $\widetilde{\mathtt{C}}_{\mathtt{pc}}$. The resulting new heap is h'.

One can access an object field with `getfield f`, or modify it using `putfield f`: the object must be initialized. Modifying an object field does not change its initialization status.

Dynamic methods can only be called on initialized objects (rule for `invokevirtual` in Figure 3.6). The method resolution returns the right method to execute (function *Lookup(m,C')*). The current object (pointed to by the reference (*Ref r*)) is passed to the called method $\mathtt{m}'$ as an argument using the special local variable `this`. Other arguments are passed to the method in variables $\mathtt{x}_1$ to $\mathtt{x}_\mathtt{n}$, assuming these are the variables identifiers found in the signature of $\mathtt{m}'$.

Let us now describe the semantic rules of constructor calls (Figure 3.6). The first rule is used when calling the first constructor on a object: the object is not initialized yet . The constructor is called with the reference to the object in its `this` register. At the beginning of the constructor, the object initialization status is updated. It changes for $\widetilde{\mathtt{C}}$ if the declared class of the object is not `Object`, and the object is considered as initialized otherwise. The second rule is used when calling a constructor on an object whose initialization is ongoing: the initialization tag of the object is $\widetilde{\mathtt{C}}$. The object is initialized only at the beginning of the constructor of class `Object`.

Let us now describe how execution errors are handled (Figure 3.7). The instruction `div` might cause a division by zero if the second top element of the stack is (*Num 0*). In this case, the execution goes into the error state Ω^{DZ}(,pc,l,h). Similarely, reading or writing a field might dereference a null pointer (the kind of error is here *NP*). Finally, concerning method and constructor calls, there are two

$$\frac{instrAt_P(m, pc) = \texttt{nop}}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle} \qquad \frac{\begin{array}{c}instrAt_P(m, pc) = \texttt{push}\ c \\ v = (Num\ c) \Leftrightarrow c \neq \texttt{null} \\ v = Null \Leftrightarrow c = \texttt{null}\end{array}}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v::s\rangle}$$

$$\frac{instrAt_P(m, pc) = \texttt{pop}}{\langle h, m, pc, l, v::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle} \qquad \frac{instrAt_P(m, pc) = \texttt{dup}}{\langle h, m, pc, l, v::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v::v::s\rangle}$$

$$\frac{\begin{array}{c}instrAt_P(m, pc) = \texttt{add} \\ v_1 = (Num\ n_1) \quad v_2 = (Num\ n_2) \\ v' = (Num\ (n_1 + n_2))\end{array}}{\langle h, m, pc, l, v_1::v_2::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v'::s\rangle} \qquad \frac{\begin{array}{c}instrAt_P(m, pc) = \texttt{div} \\ v_1 = (Num\ n_1) \quad v_2 = (Num\ n_2) \\ n_2 \neq 0 \quad v' = (Num\ (n_1/n_2))\end{array}}{\langle h, m, pc, l, v_1::v_2::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v'::s\rangle}$$

$$\frac{instrAt_P(m, pc) = \texttt{load}\ x}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, l(x)::s\rangle} \qquad \frac{instrAt_P(m, pc) = \texttt{store}\ x}{\langle h, m, pc, l, v::s\rangle \xrightarrow{x \leftarrow v}_0 \langle h, m, pc+1, l[x \mapsto v], s\rangle}$$

$$\frac{instrAt_P(m, pc) = \texttt{if}\ pc'}{\langle h, m, pc, l, (Num\ 0)::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle} \qquad \frac{instrAt_P(m, pc) = \texttt{if}\ pc' \qquad n \neq 0}{\langle h, m, pc, l, (Num\ n)::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle}$$

$$\frac{instrAt_P(m, pc) = \texttt{goto}\ pc'}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle}$$

$$\frac{instrAt_P(m, pc) = \texttt{vreturn}}{\langle h, m, pc, l, v::s\rangle \xrightarrow{return(v)}_0 \langle h, v\rangle} \qquad \frac{instrAt_P(m, pc) = \texttt{return}}{\langle h, m, pc, l, s\rangle \xrightarrow{return(Void)}_0 \langle h, Void\rangle}$$

$$\frac{\begin{array}{c}instrAt_P(m, pc) = \texttt{new}\ C \\ (Ref\ r) = newObject(C, h) \\ h' = h[r \mapsto (\lambda f.init(f))_t] \quad t = \widetilde{\mathsf{C}_{\mathsf{pc}}}\end{array}}{\langle h, m, pc, l, s\rangle \xrightarrow{mayinit(\mathsf{C})}_0 \langle h', m, pc+1, l, (Ref\ r)::s\rangle}$$

$$\frac{\begin{array}{c}instrAt_P(m, pc) = \texttt{putfield}\ f \\ h(r) = o_C \quad o' = o[f \mapsto v]\end{array}}{\langle h, m, pc, l, v::(Ref\ r)::s\rangle \xrightarrow{\tau.[r.f \leftarrow v]}_0 \langle h[r \mapsto o'], m, pc+1, l, s\rangle} \qquad \frac{\begin{array}{c}instrAt_P(m, pc) = \texttt{getfield}\ f \\ h(r) = o_C\end{array}}{\langle h, m, pc, l, (Ref\ r)::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, o(f)::s\rangle}$$

Figure 3.5: BC transition system

$$\forall \mathtt{m} \in \mathbb{M}, \mathit{InitLState}(\mathtt{m}) = (\mathtt{m}, 0, [\mathit{this} \mapsto (\mathit{Ref}\ r), \mathtt{x_1} \mapsto v_1 \ldots \mathtt{x_n} \mapsto v_n], \varepsilon)$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(m, pc) = \mathtt{invokevirtual\ C.m'} \\ h(r) = o_{C'} \quad V = v_1 :: \ldots :: v_n \\ \mathit{Lookup}(m', \mathtt{C'}) = \mathtt{mc} \quad \langle h, \mathit{InitLState}(\mathtt{mc})\rangle \stackrel{\vec{\lambda}}{\Rightarrow}_n \langle h', rv\rangle \\ \text{if } rv = \mathit{Void} \text{ then } s' = s \text{ else } s' = rv :: s\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r.C.mc(V)].\vec{\lambda}_h}_{n+1} \langle h', m, pc+1, l, s'\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(m, pc) = \mathtt{constructor}\ C \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}_\mathtt{j}} \\ \text{if } \mathtt{C} = \mathtt{Object} \text{ then } t' = \mathtt{C} \text{ else } t' = \widetilde{\mathtt{C}} \\ h' = h[r \mapsto o_{t'}] \quad V = v_1 :: \ldots :: v_n \\ \langle h', \mathit{InitLState}(\mathtt{C.init})\rangle \stackrel{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{[r \leftarrow C.init(V)].\vec{\lambda}_h}_{n+1} \langle h'', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(m, pc) = \mathtt{constructor}\ C' \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}} \quad \mathtt{C'} \supset \mathtt{C} \\ \text{if } \mathtt{C'} = \mathtt{Object} \text{ then } t' = \mathtt{C} \text{ else } t' = t \\ h' = h[r \mapsto o_{t'}] \quad V = v_1 :: \ldots :: v_n \\ \langle h', \mathit{InitLState}(\mathtt{C'.init})\rangle \stackrel{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r.C'.init(V)].\vec{\lambda}_h}_{n+1} \langle h'', m, pc+1, l, s\rangle}$$

Figure 3.6: BC transition system : object initialization and method calls

$$\frac{\mathit{instrAt}_P(m, pc) = \texttt{div}}{\langle h, m, pc, l, (\mathit{Num}\ n)::(\mathit{Num}\ 0)::s\rangle \xrightarrow{\tau}_0 \Omega^{DZ}(m, pc, l, h)}$$

$$\frac{\mathit{instrAt}_P(m, pc) = \texttt{putfield}\ f}{\langle h, m, pc, l, v::\mathit{Null}::s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)} \qquad \frac{\mathit{instrAt}_P(m, pc) = \texttt{getfield}\ f}{\langle h, m, pc, l, \mathit{Null}::s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)}$$

$$\frac{\mathit{instrAt}_P(m, pc) = \texttt{invokevirtual C.m}' \quad V = v_1::\ldots::v_n}{\langle h, m, pc, l, V::\mathit{Null}::s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(m, pc) = \texttt{invokevirtual C.m}' \quad h(r) = o_{C'} \\ \mathit{Lookup}(m', C') = \texttt{mc} \quad V = v_1::\ldots::v_n \\ \langle h, \mathit{InitLState}(\texttt{mc})\rangle \vec{\Rightarrow}_n \Omega^k(mc, pc', l_e, h_e)\end{array}}{\langle h, m, pc, l, V::(\mathit{Ref}\ r)::s\rangle \xrightarrow{\tau.[r.C.mc(V)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

$$\frac{\mathit{instrAt}_P(m, pc) = \texttt{constructor}\ C \quad V = v_1::\ldots::v_n}{\langle h, m, pc, l, V::\mathit{Null}::s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(m, pc) = \texttt{constructor}\ C \quad h(r) = o_t \quad t = \widetilde{C}_j \\ \text{if C = Object then } t' = C \text{ else } t' = \widetilde{C} \\ V = v_1::\ldots::v_n \quad h' = h[r \mapsto o_{t'}] \\ \langle h', \mathit{InitLState}(\texttt{C.init})\rangle \vec{\Rightarrow}_n \Omega^k(C.init, pc', l_e, h_e)\end{array}}{\langle h, m, pc, l, V::(\mathit{Ref}\ r)::s\rangle \xrightarrow{[r \leftarrow C.init(V)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(m, pc) = \texttt{constructor}\ C' \quad h(r) = o_t \quad t = \widetilde{C} \quad C' \supseteq C \\ \text{if C' = Object then } t' = C \text{ else } t' = t \\ V = v_1::\ldots::v_n \quad h' = h[r \mapsto o_{t'}] \\ \langle h', \mathit{InitLState}(\texttt{C'.init})\rangle \vec{\Rightarrow}_n \Omega^k(C'.init, pc', l_e, h_e)\end{array}}{\langle h, m, pc, l, V::(\mathit{Ref}\ r)::s\rangle \xrightarrow{\tau.[r \leftarrow C'.init(V)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

Figure 3.7: BC transition system : error handling

cases: either the error is raised by the call itself (leading to Ω^{NP}(,pc,l,h)), or the error arises during the execution of the callee, at a given program point pc′ (other side conditions are equal to the normal case). In this case, the error state is propagated to the caller: it ends in the error state of the same kind (*NP* or *DZ*) but parametrised by program point pc (the program point of the faulty instruction, from the point of view of the caller), heap h_e (the heap in which the error arose) and the caller local environment. This mechanism is very similar to Java Bytecode exception handlers.

Up to now, we described a rather classical bytecode semantics. We use the notion of initialization status introduced by [FM03], that has been simplified: we know the object initialization is correct, because we assume our bytecode passes the BCV. We only have to keep track of three initialization status, parametrised by the declared class of the object (uninitialized, being initialized or initialized). Now, we go into further details, describing the particularities of our semantics.

Observational semantics As can be seen in semantic rules, transitions are labelled. Labels are intented to keep track of the most of information about the program behaviour (e.g. memory effects or variable modifications...). Every (relevant) preserved elementary action should be made observable. Program behaviour aspects that are not preserved by the transformation are defined in terms of silent

transitions, written $s \xrightarrow{\tau} s'$ (see e.g rules for `nop`, `dup` or `load x`). From now on, we use λ to denote either the silent event τ or any observable event. Observable events are defined as the union *Evt* of the following sets:

EvtS ::=	$x \leftarrow v$	(*local assignt*)	*EvtH* ::=	$r.f \leftarrow v$	(*field assignt*)
			|	*mayinit*(C)	(*class initializer*)
EvtR ::=	*return*(v)	(*method return*)	|	r.C.m($v_1, \ldots, v_n$)	(*method call*)
|	*return*(*Void*)		|	$r \leftarrow$ C.init($v_1, \ldots, v_n$)	(*constructor*)
			|	r.C.init($v_1, \ldots, v_n$)	(*super constructor*)

Assigning the top element v of the stack to the local variable `x` (rule for `store x`) gives rise to the observable event $x \leftarrow v$. When assigning the value v to the field `f` of the object pointed to by the reference (*Ref* r) (rule for `putfield f`), the transition is labelled by the sequence of events $\tau.[r.\texttt{f} \leftarrow v]$ (the τ event is introduced in order to match the execution of the BIR assertion generated when transforming this instruction – more detail in Chapter 5). When returning from a method (rules for `return` and `vreturn`), *Void* or the return value is made observable.

We do not observe the "object allocation" event – the memory effect of the instruction `new C`: the transformation does not preserve the order in which objects are allocated. Both allocation orders could be related using trace languages, but this would make the trace equivalence statement far too complex in relation to the gain of information: as long as no constructor has been called on a reference, no operation is allowed on it, appart from basic stack operations (e.g `dup`), and passing it as a constructor argument.

Still, we would like the class initialization order to be preserved. In the JVM, classes are initialized at the time of their first use: object creation, static method invocation, or a static field access. BC does not include static methods or fields. Hence, the class initialization only happens when the first object of a class is created. In Java Bytecode, a class initialization consists in executing its static initializer, and its static fields initializers.In BC, we restrict class initialization to the only raising of label *mayinit*(C) (see rule for `new C`). Adding proper class initialization would not bring new difficulties.

Transitions Let us now describe the different kinds of transitions used in the semantics. First, a single transition can give rise to several observable events (see e.g. rule for `putfield f` in Figure 3.5). To this end, we use *multi-label transitions*.

Definition 1 (Multi-label transition)**.** *A multi-label transition* $s_1 \xrightarrow{\vec{\lambda}} s_2$ *between state* s_1 *and* s_2 *is a transition labelled by a (finite) sequence of events* $\vec{\lambda} = \lambda_1.\lambda_2 \ldots \lambda_n$.

In rules for method or constructor calls, the execution of the callee has to be considered on its whole from its starting states to its return (or error) state. We thus define *multi-step transitions* as being the transitive closure of transitions: several steps are performed between two states s_1 and s_n but intermediate states of the computation are not distinguished

Definition 2 (Multi-step transition)**.** *There is a multi-step transition* $s_1 \Rightarrow^{\vec{\lambda}} s_n$ *between states* s_1 *and* s_n *if there exist states* s_2 *up to* s_n *and multi-labels* $\vec{\lambda_1}, \ldots, \vec{\lambda_{n-1}}$ *such that* $\vec{\lambda} = \vec{\lambda_1}.\vec{\lambda_2} \ldots \vec{\lambda_{n-1}}$ *and* $s_1 \xrightarrow{\vec{\lambda_1}} s_2 \xrightarrow{\vec{\lambda_2}} \ldots \xrightarrow{\vec{\lambda_{n-2}}} s_{n-1} \xrightarrow{\vec{\lambda_{n-1}}} s_n$.

Note that definitions of multi-step and multi-label transitions are mutually recursive, from the above definition and the semantic rules for method and constructor calls.

In rule for `invokevirtual C.m'` (Figure 3.6), the whole method `m'` is executed and terminates in state $\langle h'', rv\rangle$, using a multi-step transition. This execution produces an event trace $\vec{\lambda}$. This trace contains events related to the local variables of the method, its method calls, some modifications of the heap, and the final return event. While events in *EvtS* and *EvtR* only concerns `m'` (they are irrelevant for `m`), events related to the heap should be seen outside the method (i.e. from each caller) as well, since the heap is shared between all methods. We hence define the filtering $\vec{\lambda}_c$ of an event trace $\vec{\lambda}$ to a category $c \in \{EvtS, EvtH, EvtR\}$ of events as the maximal subtrace of $\vec{\lambda}$ that contains only events in c. Finally, if the method terminates, then the caller `m` makes a multi-label step, the trace $\vec{\lambda}_{EvtH}$ (written $\vec{\lambda}_h$ in the rules) being exported from the callee `m'`. Constructor calls rules are based on the same idea. Note the semantics does not distinguishes between executions that are blocked and non-terminating.

Finally, each transition of our system is parametrized by a positive integer n representing the *call-depth* of the transition, the number of method calls that arise within this computation step. Concerning *one-step* transitions, this index is incremented when calling a method or a constructor. For multi-step transitions, we define the call-depth index by the following two rules:

$$\frac{s_1 \xrightarrow{\vec{\lambda}}_n s_2}{s_1 \Rightarrow^{\vec{\lambda}}_n s_2} \qquad \frac{s_1 \Rightarrow^{\vec{\lambda_1}}_{n_1} s_2 \qquad s_2 \xrightarrow{\vec{\lambda_2}}_{n_2} s_3}{s_1 \Rightarrow^{\vec{\lambda_1}.\vec{\lambda_2}}_{n_1+n_2} s_3}$$

The call-depth index is mainly introduced for technical reasons. We show the semantics preservation theorem in Chapter 6 by strong induction on the call-depth of the step.

In this chapter, we have defined the source language we consider. BC is a sequential subset of the Java bytecode language, providing object oriented features. We do not include exceptions, but the way we handle execution errors is very similar. The BC semantics is defined in terms of a labelled transition system. Labels are used to keep track of the most possible of behavioural aspects preserved by the transformation. In the next chapter, we define the language BIR targeted by the tansformation. Its semantics is intentionally very similar to the one of BC: it is based on the same ideas and uses the same kinds of transitions, so as to easily formulate the semantics preservation in terms of a simulation property.

Chapter 4

The target language: BIR

In this chapter, we describe the intermediate representation language we propose. The BIR language provides a language for expressions, and a dedicated instruction folding object allocation and the corresponding constructor call. In the first section, we describe its syntax and motivate our choices. The second section presents the operational semantics we give to BIR, which is very similar to the BC semantics (Section 3.2).

4.1 Syntax of BIR

We already saw in the introduction how expression trees could increase the precision of static analyses. Many analyses first reconstruct expressions before analysing the bytecode. The BIR language provides a language for expressions. The language of BIR expressions and instructions is given in Figure 4.1. BIR distinguishes two kinds of variables: local variables in *var* are identifiers that are also used at the BC level, while local variables in *tvar* are fresh variables that are introduced in the BIR. An expression is either a constant (an integer or `null`), a variable, an addition or division of two expressions, or an access to a field of an arbitrary expression (e.g `x.f.g`).

In BIR, a variable or the field of a given expression can be assigned with $x := e$ and $e.\mathtt{f} := e'$. We do not aim at recovering control structures: BIR is unstructured and provides conditional and unconditional jumps to a given program point `pc`. Constructors are fold in BIR: new objects are created with the instruction `new C(`$e, \ldots e$`)`, and are directly stored in a variable. The reason for folding method and constructor calls is twofold: first, to ease the analyses that often need to relate the allocated reference and the corresponding constructor. Then, as no operation is permitted on an unitialised object, there would be no need to keep the unitialised reference available in a variable. In the constructor of class `C`, the constructor of the super class has to be called. This is done with the instruction e`.super(C',`e`,...,`e`)`, where `C'` is the super class of `C`. We need to pass `C'` as an argument (unlike in Java source) in order to identify which constructor has to be called . The same remarks applies for method calls. Every method ends with a return instruction: `vreturn` e or `return` depending on whether the method returns a value or not (*Void*). When calling a method on an object, the result, if any, must be directly stored in a local variable (as there is no stack in BIR anymore). If the method returns *Void*, the instruction e`.m(C,`$e, \ldots, e$`)` is used.

We model class initialization in a restricted way. The instruction `mayinit C` behaves as `nop`, but raises a specific event in the observational semantics described below that makes possible to show that the class initialization order is preserved. A possible extension taking into account a more realistic class initialization would consist in giving the appropriate semantics to this instruction, as well to

var	::=		*local variables* :
		$\mathtt{x} \mid \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots$	
		`this`	
tvar	::=		*temporary variables* :
		$\mathtt{t} \mid \mathtt{t}_1 \mid \mathtt{t}_2 \mid \ldots$	
x	::=		*variables* :
		var | *tvar*	
oper	::=		*operands*
		$\mathtt{pc} \mid \mathtt{pc}' \mid \ldots$	*program counter*
		$\mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \ldots$	*class name*
		$\mathtt{f} \mid \mathtt{f}' \mid \ldots$	*field name*
		$\mathtt{m} \mid \mathtt{m}' \mid \ldots$	*method name*

expr	::=		*expressions* :
		`c` | `null`	*constants*
		x	*variables*
		expr + *expr*	*addition*
		expr.`f`	*field access*
instr	::=		*instructions* :
		`nop`	
		| `notnull` *expr* | `notzero` *expr*	
		| `mayinit` `C`	
		| *x* := *expr*	
		| *expr*.`f` := *expr*	
		| *x* := `new` `C`(*expr*, ..., *expr*)	
		| *expr*.`super` (`C`, *expr*, ..., *expr*)	
		| *x* := *expr*.`m`(`C`, *expr*, ..., *expr*)	
		| *expr*.`m`(`C`, *expr*, ..., *expr*)	
		| `if` *expr* `pc` | `goto` `pc`	
		| `vreturn` *expr* | `return`	

Figure 4.1: Expressions and instructions of BIR

mcode	::=		*method code* :
		`pc` : `vreturn` *expr*	*method end*
		`pc` : `return`	*method end*
		`pc` : $(instr \backslash \{\mathtt{return}, \mathtt{vreturn}\ expr\})^*$	
		mcode	*instr list*
r	::=		*return type* :
		`void` | `v`	*void*/*non void*
msig	::=		*signature* :
		`C` *r* `m` (*var* ... *var* : *var* ... *var*)	

method	::=		BIR *method* :
		msig {	
		`1` : $instr^*$	
		mcode	
		}	

class	::=		BIR *class* :
		`C` {	
		$\mathtt{f}_1 \ldots \mathtt{f}_n$	
		$method_1 \ldots method_m$ }	
prog	::=		BIR *program* :
		$class_1 \ldots class_p$	

Figure 4.2: BIR syntax: methods, classes, programs

the BC instruction `new C` (execute the static initializer whenever it is required), and to match both executions (here, only the event they raise are matched) in the theorem.

Finally, BIR provides two assertions: `notzero e` and `notnull e` respectively check if the expression `e` evaluates to zero or null. In this case, the execution produces a corresponding error state. These assertions are used to ensure both BC and BIR error states are reached, in that event, at the same program point, in equivalent contexts and because of the same kind of error.

Figure 4.2 gives the syntax of BIR programs, which is very similar (apart from instructions) from the BC syntax. Like in BC, a BIR method is made of its signature together with its code. The code of a BIR method can be seen as a list of lists of BIR instructions: the transformation algorithm can generate several instructions for a single BC instruction. To make the semantic preservation more easy to state and prove, we keep track of this instruction mapping: BIR program instructions are grouped into lists which are indexed by a program counter `pc`: `pc` is the label of the initial BC instruction. In the following, we denote by $instrsAt_P(m, pc)$ the list of instructions at label `pc` in the method `m` of program `P`. A BIR class is made of its name, its fields names and its methods. Finally, a BIR program is a set of BIR classes. Here is the BIR version of our example program of Section 3.1. Expressions `x+y` and `z+1` are reconstructed.

```
Test {
  f1 f2
  Test v main (x, y : z) {
     4: z := x + y
     8: vreturn 1+z
  }
}
```

4.2 Semantics of BIR

4.2.1 Semantic domains

Semantic domains of BIR are rather similar to those of BC, except that BIR is stackless. They are given in Figure 4.3. BIR and BC share the same definition for values, objects and heaps.

$$\begin{array}{rcl} Env_{\text{BIR}} & = & var \cup tvar \hookrightarrow Value \\ State_{\text{BIR}} & = & (Heap \times \mathbb{M} \times (\mathbb{N} \times instr^*) \times Env_{\text{BIR}}) \\ & & \cup\ (Heap \times \overline{Value}) \\ & & \cup\ \left\{ \Omega^{NP}, \Omega^{DZ} \right\} (\mathbb{N} \times \mathbb{M} \times Env_{\text{BIR}} \times Heap) \end{array}$$

Figure 4.3: The BIR's semantic domains

As already mentionned, BIR program instructions are organized into lists. Hence, the program counter does not index a single instruction. In a BIR semantic state, the current program point is defined as a pair $(\mathtt{pc}, \ell) \in \mathbb{N} \times instr^*$ where `pc` is the program counter and ℓ is the list of instructions being executed. The head element of the list defines the next instruction to execute. More details is given in the semantic rules about the way the execution flows from one instruction to its successor.

Note that error states are defined as in BC. Still, the $\mathbb{N}$ parameter uniquely determines the faulty program point. As will be seen in the next chapter, at most one assertion is generated per instruction list, and it is always the first instruction of the list.

4.2.2 Semantics of expressions

The semantics of expressions is defined as is standard by induction of the structure of the expression, given an environment and a heap. Expressions are intended to be side-effect free, as it makes easier their treatment in static analyses. As it is clear from the context, we use the same symbols + and / for both syntaxic and semantic versions of the addition and division operators. The semantics of expression is defined by the relation defined on $Heap \times Env \times expr \times Value$:

$$\frac{}{h, l \vDash \mathtt{c} \Downarrow (Num\ c)} \qquad \frac{}{h, l \vDash \mathtt{null} \Downarrow Null} \qquad \frac{x \in dom(l)}{h, l \vDash x \Downarrow l(x)}$$

$$\frac{h, l \vDash e_i \Downarrow (Num\ n_i) \text{ for } i = 1, 2}{h, l \vDash e_1 + e_2 \Downarrow (Num\ (n_1 + n_2))} \qquad \frac{h, l \vDash e_i \Downarrow (Num\ n_i) \text{ for } i = 1, 2 \qquad n_2 \neq 0}{h, l \vDash e_1 / e_2 \Downarrow (Num\ (n_1 / n_2))}$$

$$\frac{h, l \vDash e \Downarrow (Ref\ r),\ r \in Reference \quad r \in dom(h) \quad h(r) = o_{\mathsf{C}} \quad f \in dom(o_{\mathsf{C}})}{h, l \vDash e.f \Downarrow o_{\mathsf{C}}(f)}$$

Figure 4.4: Semantics of BIR expressions

4.2.3 Semantics of instructions

It is very similar to the semantics of BC: it is an observational, operational semantics. We model observational events the same way than in BC. Although they are not relevant in the trace equivalence statement, temporary variables modifications are made observable: we need to distinguish them from the τ event in order to be able to match execution traces. We thus split events in *EvtS* into two event subsets:

$$\begin{array}{rcl} EvtS & = & EvtSLoc \cup EvtSTmp \\ & = & \{x \leftarrow v \mid x \in var\} \cup \{x \leftarrow v \mid x \in tvar\} \end{array}$$

Transition rules are given in Figures 4.5, 4.6 and 4.7. Let us now explain how the flow of execution goes from one instruction to the other. Suppose the instruction list being executed is $\ell = i; \ell'$. As can be seen in Figures 4.5 and 4.6, the first instruction $i = hd(\ell)$ of ℓ is first executed. If the flow of control does not jump, then we use the function *next* defined as follows:

$$next(pc, i; \ell') = \begin{cases} (pc + 1, instrsAt_P(m, pc + 1)) & \text{if } \ell' = nil \\ (pc, \ell') & otherwise \end{cases}$$

A program P is initially run on $instrsAt_P(main, 0)$. As will be seen in the next chapter, the generated BIR instruction lists are never empty. Hence, the function *next* is well defined. Moreover, when the control flow jumps, the instruction list to execute is directly identified by the label of the jump target (e.g. rule for `goto`).

In the rule for object creation (Figure 4.6), note that the freshly created object is directly considered as being initialized, thanks to the constructor folding: as soon as the object has been allocated, its constructor is called thus its status updated. No instruction can be executed between the object allocation and its constructor call.

Semantic rules for assertions are also rather intuitive: either the assertion passes, and the execution goes on, or it fails and the execution of the program is aborted in the corresponding error state.

Concerning error handling, notice that the BIR semantics suggests more blocking states than BC. For instance, no semantic rule can be applied when trying to execute a method call on a null pointer. Here, we do not need to take into account this case: the transformation algorithm generates an assertion when translating method call instruction. This assertion catches the null pointer dereferencing attempt.

Apart from these points, rules of BIR use the same principles as BC rules. Hence, we do not further comment them. Syntaxes and semantics of BC and BIR are now defined. The next chapter

$$\frac{hd(\ell) = \texttt{nop}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, next(pc, \ell), l\rangle} \qquad \frac{hd(\ell) = x := expr \quad h, l \vDash expr \Downarrow v}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{x \leftarrow v}_0 \langle h, m, next(pc, \ell), l[x \mapsto v]\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = expr.\texttt{f} := expr' \\ h, l \vDash expr \Downarrow (Ref\ r) \quad h(r) = o_C \quad h, l \vDash expr' \Downarrow v \quad o' = o[f \mapsto v] \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{r.f \leftarrow v}_0 \langle h[r \mapsto o'], m, next(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{if}\ expr\ \texttt{pc'} \\ h, l \vDash expr \Downarrow (Num\ 0) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, (pc', instrsAt_P(m, pc')), l\rangle} \qquad \frac{\begin{array}{c} hd(\ell) = \texttt{if}\ expr\ \texttt{pc'} \\ h, l \vDash expr \Downarrow (Num\ n) \quad n \neq 0 \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, next(pc, next(pc, \ell)), l\rangle}$$

$$\frac{hd(\ell) = \texttt{goto pc'}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, (pc', instrsAt_P(m, pc')), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{vreturn}\ expr \\ h, l \vDash expr \Downarrow v \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{return(v)}_0 \langle h, v\rangle} \qquad \frac{hd(\ell) = \texttt{return}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{return(Void)}_0 \langle h, Void\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notnull}\ expr \\ h, l \vDash expr \Downarrow (Ref\ r) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, next(pc, \ell), l\rangle} \qquad \frac{\begin{array}{c} hd(\ell) = \texttt{notnull}\ expr \\ h, l \vDash expr \Downarrow Null \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notzero}\ expr \\ h, l \vDash expr \Downarrow (Num\ n) \quad n \neq 0 \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, next(pc, \ell), l\rangle} \qquad \frac{\begin{array}{c} hd(\ell) = \texttt{notzero}\ expr \\ h, l \vDash expr \Downarrow (Num\ 0) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \Omega^{DZ}(m, pc, l, h)}$$

$$\frac{hd(\ell) = \texttt{mayinit C}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{mayinit(\mathsf{C})}_0 \langle h, next(pc, \ell), l\rangle}$$

Figure 4.5: BIR transition system

defines the transformation algorithm BC2BIR, and shows that the semantics of BC is preserved by the transformation.

$$\forall \mathtt{m} \in \mathbb{M}, \mathit{InitLState}(\mathtt{m}) = \mathtt{m}, (0, \mathit{instrsAt}_P(m, 0)), [\mathit{this} \mapsto (\mathit{Ref}\ r), \mathtt{x_1} \mapsto v_1 \ldots \mathtt{x_n} \mapsto v_n]$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = x{:=}\mathtt{new\ C}\ (e_1, \ldots, e_n) \\ h, l \vDash e_i \Downarrow v_i \quad (\mathit{Ref}\ r) = \mathit{newObject}(C, h) \\ h' = h[r \mapsto (\lambda f.\mathit{init}(f))_t] \quad t = \widetilde{\mathtt{C}} \\ \langle h', \mathit{InitLState}(\mathtt{C.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow C.\mathit{init}(v_1,\ldots,v_n)].\vec{\lambda}_h.[x \leftarrow (\mathit{Ref}\ r)]}_{n+1} \langle h'', (m, \mathit{next}(pc, \ell), l[x \mapsto (\mathit{Ref}\ r)]\rangle}$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = \mathtt{e.super(C, e_1, \ldots, e_n)} \\ h, l \vDash e \Downarrow (\mathit{Ref}\ r) \quad h, l \vDash e_i \Downarrow v_i \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}'} \\ \mathtt{C} \neq \mathtt{Object} \Rightarrow \mathtt{C} \supset \mathit{mttC}' \\ \text{if } \mathtt{C} = \mathtt{Object} \text{ then } t' = \mathtt{C}' \text{ else } t' = t \\ h' = h[r \mapsto o_{t'}] \\ \langle h', \mathit{InitLState}(\mathtt{C.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.\mathit{mttC}.\mathtt{init}(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \langle h'', m, \mathit{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = y := \ \mathtt{e.m'(C, e_1, \ldots, e_n)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (\mathit{Ref}\ r) \\ h(r) = o_{\mathtt{C}'} \qquad \mathit{Lookup}(m', \mathtt{C}') = mc \\ \langle h, \mathit{InitLState}(\mathtt{mc})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', rv\rangle \\ \text{if } rv = \mathit{Void} \text{ then } l' = l \text{ else } l' = l[y \mapsto rv] \\ \text{if } rv = \mathit{Void} \text{ then } \vec{\lambda'} = \varepsilon \text{ else } \vec{\lambda'} = y \leftarrow rv \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda_h}.\vec{\lambda'}}_{n+1} \langle h', m, \mathit{next}(pc, \ell), l'\rangle}$$

Figure 4.6: BIR transition system: object initialisation and method calls

$$\frac{\begin{array}{c} hd(\ell) = x{:=}\texttt{new C}\,(e_1,\ldots,e_n) \\ h, l \vDash e_i \Downarrow v_i \quad (Ref\ r) = newObject(C,h) \\ h' = h[r \mapsto (\lambda f.init(f))_t] \quad t = \widetilde{\texttt{C}} \\ \langle h', InitLState(\texttt{C.init})\rangle \vec{\Rightarrow}_n\ \Omega^k(C.init, pc', l_e, h_e) \end{array}}{\langle h, m, (pc,\ell), l\rangle \xrightarrow{[r \leftarrow C.init(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{e.super(C, e}_1,\ldots,\texttt{e}_n\texttt{)} \\ h, l \vDash e \Downarrow (Ref\ r) \quad h, l \vDash e_i \Downarrow v_i \\ h(r) = o_t \quad t = \widetilde{\texttt{C}'} \\ C \neq Object \Rightarrow C \supset C' \\ \text{if } C = Object \text{ then } t' = C' \text{ else } t' = t \\ h' = h[r \mapsto o_{t'}] \\ \langle h', InitLState(\texttt{C.init})\rangle \vec{\Rightarrow}_n\ \Omega^k(C.init, pc', l_e, h_e) \end{array}}{\langle h, m, (pc,\ell), l\rangle \xrightarrow{[r.C.init(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

$$\frac{\begin{array}{c} hd(\ell) = y := \ \texttt{e.m}'\texttt{(C, e}_1,\ldots,\texttt{e}_n\texttt{)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (Ref\ r) \\ h(r) = o_{C'} \qquad Lookup(m', \texttt{C}') = mc \\ \langle h, InitLState(\texttt{mc})\rangle \vec{\Rightarrow}_n\ \Omega^k(mc, pc', l_e, h_e) \end{array}}{\langle h, m, (pc,\ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{e.m}'\texttt{(C, e}_1,\ldots,\texttt{e}_n\texttt{)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (Ref\ r) \\ h(r) = o_{C'} \qquad Lookup(m', \texttt{C}') = \texttt{mc} \\ \langle h, InitLState(\texttt{mc})\rangle \vec{\Rightarrow}_n\ \Omega^k(mc, pc', l_e, h_e) \end{array}}{\langle h, m, (pc,\ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k(m, pc, l, h_e)}$$

Figure 4.7: BIR transition system: error cases

Chapter 5

The BC2BIR algorithm

In this chapter, we present the transformation algorithm from BC to BIR and the properties it satisfies. It is based on a symbolic execution of the bytecode, using a symbolic operand stack. Each bytecode instruction modifies the abstract stack and gives rise to the generation of BIR instructions. It is very similar to the BC2IR algorithm of the Jalapeño Optimizing Compiler [Wha99]. Many other similar algorithms exist (see e.g [CFM+97], [XX99] or [WCN05]). Our contribution mainly lies in its formalization and especially the proof of its semantics preservation property (Chapter 6).

This algorithm is based on a symbolic execution using an abstract stack made of symbolic expressions:

Definition 3 (*AbstrStack*)**.** *Abstract stacks are defined as AbstrStack = SymbExpr*, where SymbExpr = expr* $\cup \{UR^{\mathtt{C}}_{\mathtt{pc}} \mid \mathtt{C} \in \mathbb{C}, \mathtt{pc} \in \mathbb{N}\}$.

Expressions in *expr* are decompiled expressions of BC, while $UR^{\mathtt{C}}_{\mathtt{pc}}$ is used as a placeholder for freshly allocated references in order to fold constructor calls. It denotes a reference pointing to an uninitialized object allocated in the heap by the instruction `new C` at program point `pc`. We need to keep the class (`C`) and program counter (`pc`) information for the consistency of the substitution operation on abstract stacks used in the transformation when folding constructors.

5.1 The basic transformation BC2BIR$_{instr}$

The heart of the algorithm is a transformation, BC2BIR$_{instr}$, that converts a BC instruction into a list of BIR instructions. This basic transformation is then somewhat iterated on the whole code of a method. BC2BIR$_{instr}$ is defined as a function

$$\text{BC2BIR}_{instr} : \mathbb{N} \times instr_{\text{BC}} \times AbstrStack \to \left(instr^{*}_{\text{BIR}} \times AbstrStack\right) \cup Fail$$

Given an abstract stack, BC2BIR$_{instr}$ modifies it according to the BC instruction at program point `pc`, and returns a list of BIR instructions. We will need the program counter parameter for object creation and initialization. BC2BIR$_{instr}$ is given Figure 5.1, where the $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ denote fresh temporary variables introduced at point `pc`. They are used to keep the abstract stack elements coherent with the value they should represent.

Every undescribed case yields *Fail*. Some of them (e.g. the stack height mismatches, or field assignment of an uninitialised reference) are ruled out by the BCV. A particular case of failure is the storing of an uninitialised reference in a local variable (`store x`). We explain this case below.

Inputs		Outputs	
Instr	Stack	Instrs	Stack
`nop`	*as*	[`nop`]	*as*
`pop`	e::*as*	[`nop`]	*as*
`push c`	*as*	[`nop`]	c::*as*
`dup`	e::*as*	[`nop`]	e::e::*as*
`load x`	*as*	[`nop`]	x::*as*
`if pc'`	e::*as*	[`if e pc'`]	*as*
`goto pc'`	*as*	[`goto pc'`]	*as*

Inputs		Outputs	
Instr	Stack	Instrs	Stack
`add`	e_1::e_2::*as*	[`nop`]	$e_1 + e_2$::*as*
`div`	e_1::e_2::*as*	[`notzero` e_2]	e_1/e_2::*as*
`new C`	*as*	[`mayinit C`]	UR^C_{pc}::*as*
`getfield f`	e::*as*	[`notnull e`]	e.f::*as*
`return`	*as*	[`return`]	*as*
`vreturn`	e::*as*	[`return e`]	*as*

Inputs		Outputs		Cond
Instr	Stack	Instrs	Stack	
`store x`	e::*as*	[`x := e`]	*as*	`x` $\notin$ *as*[a]
		[t^0_{pc}`:= x; x := e`]	*as*[t^0_{pc}/`x`]	`x` $\in$ *as*[a]
`putfield f`	e′::e::*as*	[`notnull e;` *Fsave*(`pc, n`)`; e.f := e′`]	*as*[t^i_{pc}/e_i]	[a b]
`invokevirtual C.m`	$e'_1 \ldots e'_n$::e::*as*	[`notnull e;` *Hsave*(`pc, m`)`;` t^0_{pc} `:= e.m(`$e'_1 \ldots e'_n$`)`]	t^0_{pc}::*as*[t^j_{pc}/e_j]	`m` returns a value [a c]
		[`notnull e;` *Hsave*(`pc, m`)`; e.m(`$e'_1 \ldots e'_n$`)`]	*as*[t^j_{pc}/e_j]	`m` returns *Void*[a c]
`constructor C`	$e'_1 \ldots e'_n$::e_0::*as*	[*Hsave*(`pc, m`)`;` t^0_{pc} `:= new C(`$e'_1 \ldots e'_n$`)`]	*as*[t^j_{pc}/e_j]	`e=`$UR^C_{pc'}$ [c]
		[`notnull e;` *Hsave*(`pc, m`)`; e.super(C,` $e'_1 \ldots e'_n$`)`]	*as*[t^j_{pc}/e_j]	otherwise [a c]

Figure 5.1: BC2BIR$_{instr}$ – Transformation of a BC instruction at `pc`

[a] where for all `C` and `pc'`, `e` $\neq UR^C_{pc'}$

[b] where e_i, $i = 1 \ldots n$ are all the elements of *as* such that `f` $\in e_i$

[c] where e_j, $j = 1 \ldots m$ are all the elements of *as* that read a field

Transforming the bytecode `push c` consists in pushing the symbolic expression `c` on the abstract stack *as* and generating the BIR instruction `nop`. Cases of `pop`, `dup`, `add` and `load x` are similar. We generate `nop` to make the step-matching easier in the simulation argument. All `nop` could be removed without changing the semantics of the BIR program. Transformations of return and jump instructions are straightforward.

For instruction `div`, generating `nop` is not sufficient for properly preserving execution errors: an assertion `notzero e` has to be generated. Let us illustrate with the example in Figure 1.1 how this assertion is used to preserve the potential division by zero. Figure 1.1(e) gives, for each program point, the entry abstract used by the transformation.

At program point 4 in Figure 1.1(c), if the second top element of the (concrete) stack, i.e the value of `y` is zero, executing the instruction `div` leads to an error state Ω^{DZ}(,4,l,h). This execution error is matched in BIR at program point 4, because the generated assertion `notzero y` will fail, leading to an BIR error state Ω^{DZ}(,4,l',h'), where environments l and l' are related, as well as heaps h ans h. Otherwise, the assertion successes, and the execution goes on. Without this assertion, the potential division by zero would happen at program point 8 (when the expression is evaluated), leading to Ω^{DZ}(,8,l',h'). Even worst, it could never happen, in case the code were dead.

The `getfield f` instruction reads the field `f` of the object pointed to by the reference on the top of the stack. The BIR assertion `notnull e` is generated. Here again, the goal of this assertion is to make sure that, if a null pointer is dereferenced by `getfield f`, the BIR program reaches the error state at the same program point. Similar assertions are generated for constructor and method calls.

There are two ways to transform `store x`. The simpliest one is when the symbolic expression `x` is not used in the abstract stack *as*: the top expression of the stack is popped and the BIR instruction `x := e` is generated. Now, if `x` is used in *as*. Suppose we only generate `x := e` and pop `e` off the stack,

the remainding expression `x` would not represent the old value of the variable `x` anymore, but the new one. Hence, before generating `x := e`, we have to store the old value of `x` in a fresh temporary variable $\mathtt{t}^{\emptyset}_{\mathtt{pc}}$. Then, every occurrence of the variable `x` in *as* has to be substituted for the new temporary $\mathtt{t}^{\emptyset}_{\mathtt{pc}}$. This substitution is written $as[\mathtt{t}^{\emptyset}_{\mathtt{pc}}/\mathtt{x}]$ and is defined in a standard way (inductively on the length of the abstract stack, and on the structure of the expression). Notice that for this instruction, we additionaly demand that the expression `e` is not $UR^{\mathtt{C}}_{\mathtt{pc}'}$: no valid BIR instruction could match this case, as constructors are fold. This hypothesis is made explicit: it is not part of the constraints checked by the BCV, as it fits the JVM specification. The same remark can be done about writing an object field (`putfield f`) and method calls.

The `putfield f` case is similar, except that there is more to save than the only `e.f`: because of aliasing, any expression $\mathtt{e}_\mathtt{i}$ could be evaluated to the same value (reference) than the variable `e`. Hence, when modifying `e.f`, $\mathtt{e}_\mathtt{i}$`.f` could be modified as well. We choose to store in fresh temporaries every element of *as* where the field `f` is used. Here, the substitutions by $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ is made element-wise. More temporary variables than needed are introduced. This could be refined using an alias analysis. Note there is no conflict between substitutions as all $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are fresh.

There are two cases when transforming object method calls: either the method returns a value, or not (*Void*). This information is available in the method signature of the bytecode. In the former case, a fresh variable is introduced in the BIR in order to store the value of the result, and this variable is pushed onto the abstract stack as the result of the method call. The execution of the callee might modify the value of all heap cells. Hence, before executing the method call, every abstract stack element that accesses the heap must be stored in a fresh variable, in order to remember its value.

We now illustrate how BC2BIR$_{instr}$ is dealing with object creation and initialization with the example of Figure 1.1. When symbolically executing `new A` at point `5` in Figure 1.1(c), $UR^{\mathtt{A}}_{5}$, corresponding to the freshly allocated reference, is pushed on the abstract stack (see Figure 1.1(e)). Class initialization could take place here which is why we generate instruction `mayinit A`.

At point 7 in Figure 1.1(d), `constructor A` is the first constructor call on the reference (and in the abstract stack in Figure 1.1(e), on $UR^{\mathtt{A}}_{5}$), the constructor call is hence folded into $\mathtt{t}_2$ `:= new A()` at this point (in Figure 1.1(d), for sake of simplicity, the identifier $\mathtt{t}^{\emptyset}_{7}$ has been replaced by `t2`). Note that for a given `C` and `j`, all $UR^{\mathtt{C}}_{\mathtt{j}}$ denote the same object, thanks to the alias information provided by *j*: we assume the BC program to pass the BCV, no such expression could be in the stack after a backward branching. The newly created (and initialized) object is stored in $\mathtt{t}_2$, each occurrence of $UR^{\mathtt{A}}_{5}$ is hence replaced by $\mathtt{t}_2$ in the resulting abstract stack (see point 8 in Figure 1.1(e)). Only $UR^{\mathtt{A}}_{5}$ has to be substituted, as we do not allow to store it in local variables in the transformation.

Given a class `C`, `constructor C` has to be decompiled another way when it corresponds to a super constructor call (see last line in Figure 5.1). Here, no new object should be created. We generate the instruction $\mathtt{e.super(C, e_1, \ldots, e_n)}$, where `e` contains the reference of the current object, and `C` is a super class of the class of `e` (its name is available in the BC instruction).

5.2 The method transformation BC2BIR

The basic instruction-wise transformation BC2BIR$_{instr}$ is used in the algorithm to generate the IR of a method. An entire BC program is obtained by translating each method of the class, and so for each class of the program. Figure 5.2 gives the algorithm transforming a whole method `m` of a given program `P`, where *length*(`m`) is the length of the code of *m* and *succ*(`pc`) is the set of all the successors of `pc` in the method *m*. We write *stackSize*(`pc`) for the (precomputed) size of the abstract stack at

program point pc. We additionaly need to compute the set of branching points of the method:

$$jmpTgt^{\mathrm{P}}_{\mathtt{m}} = \{\ j \mid \exists \mathtt{pc}, instrAt_P(m, pc) = \mathtt{if\ j}\ \text{or}\ \mathtt{goto\ j}\ \}$$

All this information can be easily, statically computed and is thus supposed available at hand.

```
1  function BC2BIR(P,m) =
2  AS_in[m,0] := nil
3  for (pc = 0, pc ≤ length(m), pc++) do
4
5    // Compute the entry abstract stack
6    if (pc ∈ jmpTgt^P_m) then
7      if (not C_UR(pc)) then fail end
8      AS_in[m,pc] := newStackJmp(pc, AS_out[m]) end
9    if (pc ∉ jmpTgt^P_m ∧ succ(pc) ∩ jmpTgt^P_m ≠ ∅) then
10     as_in := newStack(pc, AS_in[m,pc])
11   else as_in := AS_in[m,pc] end
12
13   // Decompile instruction
14   AS_out[m,pc], IR[m,pc] := BC2BIR_i(pc, instrAt_P(m,pc), as_in)
15   if ( ∀b ∈ succ(pc) ∩ jmpTgt^P_m, ∀k.T^k_b ∉ AS_in[m,pc] ) then
16     IR[m,pc] := TAssign(succ(pc) ∩ jmpTgt^P_m, AS_out[m,pc]) ++ IR[m,pc]
17   else IR[m,pc] := T̃Assign(pc, AS_in[m,pc]) ++ TAssign(succ(pc) ∩ jmpTgt^P_m, AS_out[m,pc]) ++ IR[m,pc]
18   end
19
20  // Pass around the output abstract stack
21   if (pc + 1 ∈ succ(pc) ∧ pc + 1 ∉ jmpTgt^P_m) then AS_in[m,pc+1] := AS_out[m,pc]  end
22 end
```

Figure 5.2: BC2BIR – BC method transformation

Along the algorithm, three arrays are computed. IR[m] contains the intermediate representation of the method m: for each pc, IR[m,pc] is the list of generated instructions. $\mathtt{AS_{in}}$[m] is an array of entry symbolic stacks, required to compute IR and $\mathtt{AS_{out}}$[m] contains the output abstract stack resulting from the instruction transformation.

Basically, transforming the whole code of a BC method consists in iterating the BC2BIR$_{instr}$ function, passing on the abstract stack from one instruction to its successors. If the basic transformation fails, so does the algorithm on the whole method. The algorithm consists in: (i) computing the entry abstract stack $\mathtt{as_{in}}$ used by BC2BIR$_{instr}$ (from Line 5 to 11) to transform the instruction, (ii) performing the BIR generation (from Line 14 to 17) and (iii) passing on the output abstract stack (Line 21)

Notice the transformation is performed on a single, linear pass on the bytecode. When the flow of control is linear (from pc to pc+1), the abstract stack resulting from BC2BIR$_{instr}$ is transmitted as it is (Line 21). The case of control flow joins must be handled more carefully.

Thanks to the BCV hypothesis we make on the bytecode, we already know that at every branching point, the size of the stack is the same regardless of the predecessor point. Still, for this one-pass transformation to be correct, the content of the abstract stack must be uniquely determined at these points: stack elements are expressions used in the generated instructions and hence must be independant of the control flow path leading to these program points.

Let us illustrate this point with the example function of Figure 5.3. It returns 1 or -1 depending of whether the argument x is zero or not. Let us focus on program point 5, i.e a branching point : its predecessors are points 3 and 4. The abstract stack after having executed the instruction goto 5 is

```
int f(int x) {return (x==0) ? 1 : -1;}
```
(a) source function

```
int f(x);
0: load x
1: if 4
2: push -1
3: goto 5

4: push 1

5: vreturn
```
(b) BC function

```
0: []
1: [x]
2: []
3: [-1]

4: []

5: [T_5^1]
```
(c) symbolic stack

```
int f(x);
0: nop;
1: if x 4;
2: nop;
3: T_5^1 := -1;
   goto 5;
4: T_5^1 := 1;
   nop
5: vreturn T_5^1;
```
(d) BIR function

Figure 5.3: Example of bytecode transformation – non-empty stack jumps

`[-1]` (point 3), while it becomes `[1]` after program point 4. But, when transforming the `vreturn` at point 5, the abstract stack should be uniquely determined, since the control flow is unknown.

The idea here is to store, before reaching a branching point, every stack element in a temporary variable and then to use an abstract stack made of all of these variables at the branching point. A naming convention has to be decided so that (i) identifiers do not depend on the control flow path and (ii) each variable corresponds to exactly one stack element: we use the identifier $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ to store the i^{th} element of the stack when the jump target point is pc. Hence, for each $\mathtt{pc} \in \mathit{jmpTgt}^{\mathtt{P}}_{\mathtt{m}}$, the abstract stack used by BC2BIR$_{\mathit{instr}}$ is (almost) entirely determined by pc and the size of the entry stack at this point. In Figures 5.3(c) and 5.3(d), at program points 3 and 4, we respectively store 1 and -1 in $\mathtt{T}^1_5$, a temporary variable that will be used at point 5 in the entry abstract stack of the transformation.

Thus, in the algorithm, when transforming a BC instruction that preceeds a branching point, the list or BIR instructions provided by BC2BIR$_{\mathit{instr}}$ is no longer sufficient: we must prepend to it a list of assignments of each abstract stack element to $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ variables, and so for each potential target point pc. These assignments must happen before the instruction list given by BC2BIR$_{\mathit{instr}}$, in case a jumping instruction were generated by BC2BIR$_{\mathit{instr}}$ at this point (see e.g program point 3 in Figure 5.3(d)).

Suppose now the stack before the branching point pc contains an uninitialized reference, represented by $\mathit{UR}^{\mathtt{C}}_{\mathtt{pcn}}$. As this element is not a BIR expression, it cannot be replaced by any temporary variable – the assignment would not be a legal BIR instruction. Here, we need to assume the following structural constraint on the bytecode: before a branching point pc, if the stack contains any uninitialized reference at position i, then it is the case for every predecessor of pc. More formally, this hypothesis can be formulated as a constraint on the array $\mathtt{AS}_{\mathtt{out}}$, that we check when transforming a join point (Line 7).

$$\begin{aligned}\forall \mathtt{pc}, \mathtt{pcn}, \mathtt{C}, \mathtt{i}.\quad & (\exists \mathtt{pc}'.\mathtt{pc} \in \mathit{succ}(\mathtt{pc}') \wedge \mathtt{pc}' < \mathtt{pc} \wedge \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = \mathit{UR}^{\mathtt{C}}_{\mathtt{pcn}}) \\ \Rightarrow\quad & (\forall \mathtt{pc}'.\mathtt{pc} \in \mathit{succ}(\mathtt{pc}') \wedge \mathtt{pc}' < \mathtt{pc} \wedge \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = \mathit{UR}^{\mathtt{C}}_{\mathtt{pcn}})\end{aligned}$$

Without this requirement, because constructors are fold in BIR, the transformation would fail. We use the function *newStackJmp* (Line 5) defined as follows to compute the entry abstract stack at branching point pc, where n is the size of the abstract stack at pc and AS is an abstract stack array:

$$\mathit{newStackJmp}(\mathtt{pc}, \mathtt{AS}) \quad = \quad \mathtt{e}_1 :: \ldots :: \mathtt{e}_n$$

$$\text{where } \forall i = 1 \ldots n, \mathtt{e_i} = \begin{cases} \mathtt{AS}[\mathtt{pc'}]_\mathtt{i} & \text{if } \exists \mathtt{pcn}, \mathtt{pc'}, \mathtt{C}. \quad \mathtt{pc} \in \mathit{succ}(\mathtt{pc'}) \wedge \mathtt{pc'} < \mathtt{pc} \\ & \text{such that } \mathtt{AS}[\mathtt{pc'}]_\mathtt{i} = \mathit{UR}^{\mathtt{C}}_{\mathtt{pcn}} \\ \mathtt{T}^{\mathtt{i}}_{\mathtt{pc}} & \text{otherwise} \end{cases}$$

Notice the use of this function is coherent with $\mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{0}]$: even if $\mathtt{0}$ is a branching point, the stack at the beginning of the method is empty.

Now, before reaching the branching point, we have to make sure all the $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ have been assigned. Given an abstract stack *as* and a set S of program points, T*Assign*$(\mathtt{S}, \mathit{as})$ (Lines 16 and 17) returns the list of such assignments which are ensured to be mutually conflict-free – in case *as* contained some of the $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$, new variables would be used.

A last precaution has to be taken here. In case where some $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ appeared in the entry stack used by the basic transformation, the value of these variables must be remembered: the semantics of the instruction list generated by BC2BIR$_{\mathit{instr}}$ depends on them. This can only happens at non-branching points. In this case, the entry abstract stack is changed to *newStack*$(\mathtt{pc}, \mathit{as})$ (Line 10), where n is the size of the stack *as*. The corresponding assignments are generated by $\tilde{\mathtt{T}}$*Assign*$(\mathtt{pc}, \mathit{as})$ (Line 17).

$$\mathit{newStack}(\mathtt{pc}, \mathit{as}) \quad = \quad \mathtt{e_1} :: \ldots :: \mathtt{e_n}$$

$$\text{where } \forall i = 1 \ldots n, \mathtt{e_i} = \begin{cases} \tilde{\mathtt{T}}^{\mathtt{i}}_{\mathtt{pc}} & \text{if } \exists \mathtt{pc'}, \mathtt{k}. \quad \mathtt{pc'} \in \mathit{succ}(\mathtt{pc}) \wedge \quad \mathtt{T}^{\mathtt{k}}_{\mathtt{pc'}} \in \mathtt{as_i} \\ \mathtt{as_i} & \text{otherwise} \end{cases}$$

In the following chapter, we make further remarks on the algorithm and formalize the semantics preservation property of the above algorithm.

Chapter 6

Correctness of BC2BIR

The BC2BIR algorithm we presented in the previous chapter is semantics-preserving. In this section, we formalize this notion of semantics preservation. The basic idea is that the BIR program BC2BIR(P) simulates the initial BC program P and both have similar execution traces. The similarity between traces is defined using a relation over the two heaps. This non-equality is due to the fact that the transformation does not preserves the order in which objects are allocated. Section 6.1 demonstrates this point. In Section 6.2, we define the semantic relations induced by this heap similarity. We need them in the propositions of Section 6.3 to express the execution trace preservation.

6.1 Object allocation orders

Starting from the same heap, execution of P and P' = BC2BIR(P) will not preserve the heap equality. However, the two heaps keep isomorphic: there exists a partial bijection [1] between them.

This is illustrated by the example given in Figure 1.1. In Figure 1.1(c), the object of class `B` is allocated before the object of class `A` is passed as argument to the former's constructor. In the BIR version of the program (Figure 1.1(d)), as constructors are folded, and because object creation is not an expression, the object of class `A` has to be created (and initialized) before passing the temporary variable `t1` (that contains its reference) as an argument to the constructor of the object of class `B`. Consider both executions of these programs (P is the bytecode version and P' its BIR version), starting from the same heap. When executing P, one object of class `B` is allocated at program point `0` through the reference r_1 and next, at program point `5`, a second object is allocated in the heap, with the reference r_2. When executing P', one object of class `A` is first allocated at program point `7` through the reference r_3 and next, at program point `8`, a second object is allocated in the heap, with the reference r_4. Whereas in P the `A` object is pointed to by r_2, it is pointed to by r_3 in P', and similarly for the `B` object pointed to by r_4. Heaps are hence not equal along the execution of the two programs: after program point `5` in P, the heap contains two objects that are not in the heap of P'. However, after program points `7`, we know that each time the reference r_3 is used in P', it corresponds to the use of r_2 in P (both constructors have been called, so both references can be used freely). The same reasoning can be applied just after program points `8`: r_1 in P corresponds to r_4 in P'. A bijection thus exists between references of programs P and P'. The partial bijection between the BC and BIR heaps relates allocated objects as soon as their initialization has begun. In Figure 1.1, given the initial partial

[1] The rigorous definition of a bijective function demands that it is totally defined on its domain. The term "partial bijection" is however widely used and we consider it as equivalent to "partial injection". While having a totally different end, a similar notion is used in [BN05] and [BR05] in the context of information flow of Java and Java bytecode programs.

bijection β, it is first extended at points 7 with $\beta(r_2) = r_3$, and then again at programs points 8 with $\beta(r_1) = r_4$.

6.2 Semantic relations

As stated in the previous section, the object allocation order is not preserved by the algorithm, but there exists a partial bijection between the BC and BIR heaps that relates allocated objects as soon as their initialization has begun. In order to relate heaps, environments and execution traces, semantic relations need to be defined to state and prove the semantics preservation. All of these are parametrised by the current partial bijection $\beta : \mathit{Reference} \hookrightarrow \mathit{Reference}$. First, a relation is defined over values:

Definition 4 (Value relation: $\overset{\text{V}}{\sim}_\beta$)**.**
The relation $\overset{\text{V}}{\sim}_\beta \subseteq \mathit{Value} \times \mathit{Value}$ is defined inductively by:

$$\frac{}{\mathit{Null} \overset{\text{V}}{\sim}_\beta \mathit{Null}} \qquad \frac{n \in \mathbb{Z}}{(\mathit{Num}\ n) \overset{\text{V}}{\sim}_\beta (\mathit{Num}\ n)} \qquad \frac{\beta(r_1) = r_2}{(\mathit{Ref}\ r_1) \overset{\text{V}}{\sim}_\beta (\mathit{Ref}\ r_2)}$$

The interesting case is for references. Only references related by β are related. Concerning objects on which no constructor has been called yet (i.e. a reference that is not in the domain of β), references pointing to them cannot be related.

We can now define the relation on heaps. First, only objects existing in the two heaps must be related by β. Secondly, the related objects are at least being initialized and must have the same initialization status, hence the same class. Finally, their fields must have related values. Here we write $\mathit{tag}_h(r)$ for the tag t such that $h(r) = o_t$.

Definition 5 (Heap relation: $\overset{\text{H}}{\sim}_\beta$)**.**
Let h_1 and h_2 be two heaps. We have $h_1 \overset{\text{H}}{\sim}_\beta h_2$ if and only if:

- $\mathit{dom}(\beta) = \{r \in \mathit{dom}(h_1) \mid \forall \mathsf{C}, \mathsf{pc},\ \mathit{tag}_{h_1}(r) \neq \widetilde{\mathsf{C}_{\mathsf{pc}}}\}$
- $\mathit{rng}(\beta) = \mathit{dom}(h_2)$
- $\forall r \in \mathit{dom}(h_1)$, *let* $o_t = h_1(r)$ *and* $o'_{t'} = h_2(\beta(r))$ *then* $(i)\ t = t'$ *and* $(ii)\ \forall f,\ o_t(f) \overset{\text{V}}{\sim}_\beta o'_t(f)$

A BIR environment is related to a BC environment if and only if both local variables (temporary variables are not taken into account) have related values.

Definition 6 (Environment relation: $\overset{\text{E}}{\sim}_\beta$)**.**
Let $l_1 \in \mathit{Env}_{\text{BC}}$ and $l_2 \in \mathit{Env}_{\text{BIR}}$ be two environments. We have $l_1 \overset{\text{E}}{\sim}_\beta l_2$ if and only if

$$\mathit{dom}(l_1) \subseteq \mathit{dom}(l_2) \text{ and } \forall x \in \mathit{dom}(l_1).\ l_1(x) \overset{\text{V}}{\sim}_\beta l_2(x)$$

Finally, in order to relate execution traces, we need to define a relation over events, and we extend it pointwise as is standard to event traces.

Definition 7 (Event relation: $\overset{!}{\sim}_\beta$)**.**
The relation over events $\overset{!}{\sim}_\beta$ is inductively defined:

$$\frac{}{\tau \overset{!}{\sim}_\beta \tau} \qquad \frac{}{\mathit{mayinit}(\mathsf{C}) \overset{!}{\sim}_\beta \mathit{mayinit}(\mathsf{C})} \qquad \frac{x \in \mathit{var} \quad v_1 \overset{\text{V}}{\sim}_\beta v_2}{x \leftarrow v_1 \overset{!}{\sim}_\beta x \leftarrow v_2} \qquad \frac{\beta(r_1) = r_2 \quad v_1 \overset{\text{V}}{\sim}_\beta v_2}{r_1.f \leftarrow v_1 \overset{!}{\sim}_\beta r_2.f \leftarrow v_2}$$

$$\frac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\text{v}}{\sim}_\beta v'_i}{r_1 \leftarrow C.init(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2 \leftarrow C.init(v'_1, \ldots, v'_n)} \qquad \frac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\text{v}}{\sim}_\beta v'_i}{r_1.C.init(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2.C.init(v'_1, \ldots, v'_n)}$$

$$\frac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\text{v}}{\sim}_\beta v'_i}{r_1.C.m(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2.C.m(v'_1, \ldots, v'_n)}$$

Relating run-time and abstract stacks is the first step towards bridging the gap between the two representations of P. We thus define a relation over stacks. This relation holds between s and as if the ith element of as is either an expression that evaluates to a value that is in relation w.r.t $\overset{\text{v}}{\sim}_\beta$ with the ith element of s, or is a symbol $UR^{\mathsf{C}}_{\mathsf{pc}}$ and the ith element of s is a reference r of tag $\widetilde{\mathsf{C}}_{\mathsf{pc}}$. In this last case, we furthermore require that all references that appear in the stack with the same status tag must be equal to r. This strong property is enforced thanks to the restrictions that are imposed by the BCV on uninitialised references in the operand stack.

Definition 8 (Stack correctness: $\approx_{h,l,\beta}$)**.** *Let s be in Stack, as be in AbstrStack, h be in $Heap_{\text{BIR}}$ and l in Env_{BIR}. The stack correctness is defined inductively as:*

$$\frac{}{\varepsilon \approx_{h,l,\beta} \varepsilon} \qquad \frac{h, l \vDash e \Downarrow v' \quad v \overset{\text{v}}{\sim}_\beta v' \quad s \approx_{h,l,\beta} as}{v::s \approx_{h,l,\beta} e::as} \qquad \frac{tag_h(r) = \widetilde{\mathsf{C}}_{\mathsf{pc}} \quad s \approx_{h,l,\beta} as \quad \forall (Ref\ r') \in s,\ tag_h(r') = \widetilde{\mathsf{C}}_{\mathsf{pc}} \Rightarrow r = r'}{(Ref\ r)::s \approx_{h,l,\beta} UR^{\mathsf{C}}_{\mathsf{pc}}::as}$$

6.3 Semantics preservation

The correctness proof of the transformation is organized as follows. We show that one-step transitions are preserved by the basic transformation BC2BIR_{instr}, which is used in the proof of the one-step transition preservation by BC2BIR: the proposition holds, regarless of the potential assignments added in the algorithm. Finally, multi-step transitions will be shown to be preserved by BC2BIR using a strong induction of the call-depth. Propositions are first stated in their normal and then in their error-execution variant. In the following, to lighten the notations, we write $\vec{\lambda}_{proj}$ for $\vec{\lambda}_{EvtSLoc \cup EvtH \cup EvtR}$, i.e. the projection of the trace $\vec{\lambda}$ to any category of events but *EvtSTmp*.

Proposition 1 (BC2BIR_{instr} - zero call-depth one-step preservation - normal case)**.**
Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', m, pc', l', s'\rangle$. Let ht, lt, as, β be such that:

$$h \overset{\text{H}}{\sim}_\beta ht \qquad l \overset{\text{E}}{\sim}_\beta lt \qquad s \approx_{ht,lt,\beta} as \qquad \text{BC2BIR}_{instr}(pc, instrAt_P(m, pc), as) = (\ell, as')$$

Then, there exist unique ht', lt' and $\vec{\lambda}'$ such that $\langle ht, m, (pc, \ell), lt\rangle \overset{\vec{\lambda}'}{\Rightarrow}_0 \langle ht', m, (pc', instrsAt_P(m, pc')), lt'\rangle$ with:

$$h' \overset{\text{H}}{\sim}_\beta ht' \qquad l' \overset{\text{E}}{\sim}_\beta lt' \qquad \vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda}'_{proj} \qquad s' \approx_{ht',lt',\beta} as'$$

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', rv\rangle$, then there exists $\langle ht', rv'\rangle$ such that: $\langle ht, m, (pc, \ell), lt\rangle \overset{\vec{\lambda}'}{\Rightarrow}_0 \langle ht', rv'\rangle$, with $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda}'_{proj}$, $h' \overset{\text{H}}{\sim}_\beta ht'$ and $rv \overset{\text{v}}{\sim}_\beta rv'$.

Proof. We proceed by case analysis on the BC instruction at program point pc. Here, only interesting cases are detailed. Others are trivial or can be treated a similar way.

- **push** c We have $\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, (Num\ c)::s\rangle$. Let as, ht, lt be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} as$. We have $\text{BC2BIR}_{instr}(pc, \texttt{push}\ c, as) = ([\texttt{nop}], c::as)$. Hence, $\langle ht, m, (pc, [\texttt{nop}]), lt\rangle \xrightarrow{\tau} \langle ht, m, (pc+1, instrsAt_P(m, pc+1)), lt\rangle$. The heaps and environments are unchanged, both transitions are silent. Stacks stay trivially related since $ht, lt \vDash c \Downarrow (Num\ c)$.

- **div** Here, because the execution does not reach the error state, only one case is possible : $n_2 \neq 0$, and $\langle h, m, pc, l, (Num\ n_1) :: (Num\ n_2) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, (Num\ n_1/n_2) :: s\rangle$. Let as, ht, lt be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $(Num\ n_1) :: (Num\ n_2) :: s \approx_{ht,lt,\beta} e_1 :: e_2 :: as$. We have $\text{BC2BIR}_{instr}(pc, \texttt{div}, e_1 :: e_2 :: as) = ([\texttt{notzero}(\texttt{e}_2)], e_1/e_2 :: as)$. But $ht, lt \vDash e_2 \Downarrow (Num\ n_2')$ with $(Num\ n_2) \overset{V}{\sim}_\beta (Num\ n_2')$. Thus, $n_2' \neq 0$ and $\langle ht, m, (pc, [\texttt{notzero}(\texttt{e}_2)]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc+1, instrsAt_P(m, pc+1)), lt\rangle$. Heaps and environment are unchanged, and both transitions are silent. Finally, since $ht, lt \vDash e_1 \Downarrow (Num\ n_1')$ with $(Num\ n_1) \overset{V}{\sim}_\beta (Num\ n_1')$ and $ht, lt \vDash e_2 \Downarrow (Num\ n_2')$ with $(Num\ n_2) \overset{V}{\sim}_\beta (Num\ n_2')$, we have $ht, lt \vDash e_1/e_2 \Downarrow (Num\ n_1'/n_2')$ and $(Num\ n_1/n_2) \overset{V}{\sim}_\beta (Num\ n_1'/n_2')$.

- **load x** We have $\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, l(x)::s\rangle$. Let as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} as$. We have $\text{BC2BIR}_{instr}(pc, \texttt{load x}, as) = ([\texttt{nop}], x :: as)$. Hence, $\langle ht, m, (pc, [\texttt{nop}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc+1, instrsAt_P(m, pc+1)), lt\rangle$. Heaps and environments are unchanged and both transitions are silent. We now have to prove that stacks stay related, i.e. that $l(x)::s \approx_{ht,lt,\beta} x::as$. We have $ht, lt \vDash x \Downarrow lt(x)$, $l \overset{E}{\sim}_\beta lt$ and $x \in var$. Hence, by the definition of $\overset{E}{\sim}_\beta$, we have $l(x) \overset{V}{\sim}_\beta lt(x)$.

- **store x** We have $\langle h, m, pc, l, v::s\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, pc+1, l[x \mapsto v], s\rangle$. Let as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $v::s \approx_{ht,lt,\beta} e::as$. We distinguish two cases, whether x is already in as or not:

  - If $x \notin as$ then $\text{BC2BIR}_{instr}(pc, \texttt{istore x}, e::as) = ([\texttt{x} := \texttt{e}], as)$. But $v::s \approx_{ht,lt,\beta} e::as$ and $ht, lt \vDash e \Downarrow v'$ with $v \overset{V}{\sim}_\beta v'$. Hence $\langle ht, m, (pc, [\texttt{x} := \texttt{e}]), lt\rangle \xrightarrow{[x \leftarrow v']}_0 \langle ht, m, (pc+1, instrsAt_P(m, pc+1)), lt[x \mapsto v']\rangle$. Now, heaps are not modified, and stay related. Labels are related: we have $x \in var$ because it is used in a bytecode instruction, and $v \overset{V}{\sim}_\beta v'$. Thus $[x \leftarrow v] \overset{!}{\sim}_\beta [x \leftarrow v']$. Environment stay related: $l[x \mapsto v] \overset{E}{\sim}_\beta lt[x \mapsto v']$ since $l \overset{E}{\sim}_\beta lt$ by hypothesis and $v \overset{V}{\sim}_\beta v'$. We finally have to prove that $s \approx_{ht,lt',\beta} as$, where $lt' = lt[x \mapsto v']$. Stacks are the same height. Moreover, as $x \notin as$, for all abstract stack elements as_i, we have: $ht, lt' \vDash as_i \Downarrow v_i'$ and $ht, lt \vDash as_i \Downarrow v_i'$ with $v_i \overset{V}{\sim}_\beta v_i'$.
  - If $x \in as$ then $\text{BC2BIR}_{instr}(pc, \texttt{istore x}, e::as) = ([\texttt{t}^{\texttt{0}}_{\texttt{pc}} := \texttt{x}; \texttt{x} := \texttt{e}], as[\texttt{t}^{\texttt{0}}_{\texttt{pc}}/\texttt{x}])$. We hence have that $\langle ht, m, (pc, [\texttt{t}^1_{\texttt{pc}} := \texttt{x}; \texttt{x} := \texttt{e}]), lt\rangle \xrightarrow{[\texttt{t}^1_{\texttt{pc}} \leftarrow lt(x)]}_0 \langle ht, m, (pc, [\texttt{x} := \texttt{e}]), lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x)]\rangle$. $\texttt{t}^1_{\texttt{pc}}$ is fresh, so $\texttt{t}^1_{\texttt{pc}} \notin e$.

    Hence $ht, lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x)] \vDash e \Downarrow v'$ where v' is such that $ht, lt \vDash e \Downarrow v'$, and $v \overset{V}{\sim}_\beta v'$ by hypothesis. Thus, we have $\langle ht, m, (pc, [\texttt{t}^1_{\texttt{pc}} := \texttt{x}; \texttt{x} := \texttt{e}]), lt\rangle \xrightarrow{[\texttt{t}^1_{\texttt{pc}} \leftarrow lt(\texttt{x})].[\texttt{x} \leftarrow v']}_0 \langle ht, m, (pc+1, instrsAt_P(m, pc+1)), lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x), \texttt{x} \mapsto v']\rangle$.

    Heaps are not modified. We have $[x \leftarrow v] \overset{E}{\sim}_\beta \left([\texttt{t}^1_{\texttt{pc}} \leftarrow lt(x)].[x \leftarrow v']\right)_{proj} = [x \leftarrow v']$ because only $\texttt{t}^1_{\texttt{pc}} \in tvar$ and $v \overset{V}{\sim}_\beta v'$. Environments stay related because $\texttt{t}^1_{\texttt{pc}} \in tvar$ and $x \in var$ is assigned the value v' with $v \overset{V}{\sim}_\beta v'$.

We now have to show that $s \approx_{ht,lt',\beta} as[\mathtt{t}^1_{\mathtt{pc}}/x]$, where $lt' = lt[\mathtt{t}^1_{\mathtt{pc}} \mapsto lt(x), x \mapsto v']$. But for all elements $as[\mathtt{t}^1_{\mathtt{pc}}/x]_i$ of the abstract stack, we have: $ht, lt' \vDash as[\mathtt{t}^1_{\mathtt{pc}}/x]_i \Downarrow v_i$ where v_i is such that $ht, lt \vDash as_i \Downarrow v_i$ because $lt'(\mathtt{t}^1_{\mathtt{pc}}) = lt(x)$ and $\mathtt{t}^1_{\mathtt{pc}}$ is fresh, so $\mathtt{t}^1_{\mathtt{pc}} \notin as_i$.

- **`if pc'`** According to the top element of the stack, there are two cases. We only treat the case of a jump, the other one is similar. We have $\langle h, m, pc, l, (Num\ 0)::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle$. Let as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $(Num\ 0)::s \approx_{ht,lt,\beta} e::as$. We have $\text{BC2BIR}_{instr}(pc, \mathtt{if\ pc'}, e::as) = ([\mathtt{if}\ e\ pc'], as)$. But stacks are related by hypothesis, thus e evaluates to zero and $\langle ht, m, (pc, [\mathtt{if}\ e\ pc']), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc', instrsAt_P(m, pc')), lt\rangle$ and labels are related. Heaps and environments are unchanged. Stacks stay trivially related.

- **`new C`** We have $\langle h, m, pc, l, s\rangle \xrightarrow{mayinit(\mathsf{C})}_0 \langle h', m, pc+1, l, (Ref\ r) :: s\rangle$, with $(Ref\ r)$ freshly allocated and $h' = h[r \mapsto (\lambda f.init(f))_{\widetilde{\mathsf{C}}_{\mathtt{pc}}}]$. Let as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} as$. We have that $\text{BC2BIR}_{instr}(pc, \mathtt{new}\ C, as) = ([\mathtt{mayinit(C)}], UR^{\mathsf{C}}_{\mathtt{pc}} :: as)$. Hence $\langle ht, m, (pc, [\mathtt{mayinit\ C}]), lt\rangle \xrightarrow{[mayinit(\mathsf{C})]}_0 \langle ht, m, (pc{+}1, instrsAt_P(m, pc{+}1)), lt\rangle$. Labels are equal and environments are not modified. The reference $(Ref\ r)$ is pointing to an uninitialized object in h', so β is not extended, and heaps keep related. Finally, we have $(Ref\ r)::s \approx_{ht,lt,\beta} UR^{\mathsf{C}}_{\mathtt{pc}}::as$: by the BCV hypothesis on the BC program, r is the only reference pointing to the uninitialized object (when executing `new C` at this point, we are ensured that no reference pointing to an uninitialized object of class `C` allocated at pc is already in the stack).

- **`getfield f`** The execution does not reach the error state. Hence, we have $\langle h, m, pc, l, (Ref\ r)::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc{+}1, l, h(r)(f)::s\rangle$, with $h(r) = o_{\mathsf{C}}$. Let e, as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $(Ref\ r)::s \approx_{ht,lt,\beta} e::as$. We have $\text{BC2BIR}_{instr}(pc, \mathtt{getfield\ f}, e::as) = ([\mathtt{notnull(e)}], e.f::as)$. By hypothesis on the stacks, we have that $ht, lt \vDash e \Downarrow (Ref\ r')$. Hence, `e` does not evaluates to *Null* and $\langle ht, m, (pc, [\mathtt{notnull(e)}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc+1, instrsAt_P(m, pc+1)), lt\rangle$. Heaps and environments are not modified, labels are related. We now have to show that stacks keep related. By hypothesis, we have $\beta(r) = r'$ since the object pointed to by r is initialized. Besides, $ht, lt \vDash e.f \Downarrow ht(r')(f)$ since $ht, lt \vDash e \Downarrow (Ref\ r')$ and $ht(r')(f) = ht(\beta(r))(f)$. We know that $h \overset{\text{H}}{\sim}_\beta ht$ by hypothesis, hence $h(r)(f) \overset{\text{V}}{\sim}_\beta ht(\beta(r'))(f)$. Stacks are hence related.

- **`putfield f`** We have $\langle h, m, pc, l, v::(Ref\ r)::s\rangle \xrightarrow{\tau.[r.f \leftarrow v]}_0 \langle h[r(f) \mapsto v], m, pc+1, l, s\rangle$ (the field of the object pointed to by r is modified), with $h(r) = o_{\mathsf{C}}$. Let e, e', as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $v::(Ref\ r)::s \approx_{ht,lt,\beta} e'::e::as$. There are two cases:

  - If `f` is not in any expression of the abstract stack, we have $\text{BC2BIR}_{instr}(pc, \mathtt{putfield\ f}, e'::e::as) = ([\mathtt{notnull(e)}; \mathtt{e.f := e'}], as)$. But $v :: (Ref\ r) :: s \approx_{ht,lt,\beta} \mathtt{e'} :: \mathtt{e} :: as$. We get that $v \overset{\text{V}}{\sim}_\beta v'$ where $ht, lt \vDash \mathtt{e'} \Downarrow v'$ and that there exists r' such that $ht, lt \vDash \mathtt{e} \Downarrow (Ref\ r')$ with $(Ref\ r) \overset{\text{V}}{\sim}_\beta (Ref\ r')$, and r' points in ht to an initialized object, since the BC field assignment is permitted.

    We hence have $\langle ht, m, (pc, [\mathtt{notnull(e)}; \mathtt{e.f := e'}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc, [\mathtt{e.f := e'}]), lt\rangle \xrightarrow{[r'.f \leftarrow v']}_0 \langle ht[r'(f) \mapsto v'], m, (pc+1, instrsAt_{pc+1}(), ,)lt\rangle$. Environments are unchanged and stay related. We have to show that $h' = h[r(f) \mapsto v] \overset{\text{H}}{\sim}_\beta ht' = ht[r'(f) \mapsto v']$. We have $(Ref\ r) \overset{\text{V}}{\sim}_\beta (Ref\ r')$, hence $\beta(r) = r'$. Besides, $v \overset{\text{V}}{\sim}_\beta v'$ with $ht, lt \vDash e' \Downarrow v'$. Fields of the two objects pointed to by r and r' have hence related values w.r.t β. Finally, we have $\tau.[r.f \leftarrow v] \overset{!}{\sim}_\beta \tau.[r'.f \leftarrow v']$ since $v \overset{\text{V}}{\sim}_\beta v'$ and $(Ref\ r) \overset{\text{V}}{\sim}_\beta (Ref\ r')$.

– If $\mathtt{f} \in as$, we have $\text{BC2BIR}_{instr}(pc, \mathtt{putfield\ f}, e' :: e :: as) = ([\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := as_i; e.f := e'], as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i])$. But $v{::}(\mathit{Ref}\ r){::}s \approx_{ht,lt,\beta} e'{::}e{::}as$ hence, as in the previous case: $v \overset{\mathrm{v}}{\sim}_\beta v'$ where v' is such that $ht, lt \vDash e' \Downarrow v'$ and there exists r' such that $ht, lt \vDash e \Downarrow (\mathit{Ref}\ r')$ with $(\mathit{Ref}\ r) \overset{\mathrm{v}}{\sim}_\beta (\mathit{Ref}\ r')$.

Suppose now that n elements of as are expressions using the field $\mathtt{f}$. For all $i \in [1; n]$, let v_i be such that $ht, lt \vDash as_i \Downarrow v_i$. Thus, $\langle ht, m, (pc, [\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := as_i; e.f := e']), lt\rangle \xrightarrow{\tau.[\mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} \leftarrow v_1]\ldots[\mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \leftarrow v_n]}_0 \langle ht, m, (pc, [e.f := e']), lt[\mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} \mapsto v_1, \ldots \mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \mapsto v_n]\rangle$.

All $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are fresh, they hence do not appear in e or e'. Let $lt' = lt[\mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} \mapsto v_1, \ldots \mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \mapsto v_n]$. Thus $ht, lt' \vDash e' \Downarrow v'$ with $ht, lt \vDash e' \Downarrow v'$ and $ht, lt' \vDash e \Downarrow (\mathit{Ref}\ r')$. Thus, $\langle ht, m, (pc, [e.f := e']), lt'\rangle \xrightarrow{[r'.f \leftarrow v']}_0 \langle ht[r'(f) \mapsto v'], m, (pc + 1, \mathit{instrsAt}_P(m, pc + 1)), lt'\rangle$.

Events are related: $\tau.[r.f \leftarrow v] \overset{!}{\sim}_\beta (\tau.[\mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} \mapsto v_1] \ldots [\mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \leftarrow v_n].[r'.f \leftarrow v'])_{proj}$ because all $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are in $tvar$, $\beta(r) = r'$ and $v \overset{\mathrm{v}}{\sim}_\beta v'$. Environments stay related: $l \overset{\mathrm{E}}{\sim}_\beta lt'$ because all $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are fresh. Besides, since $\beta(r) = r'$ and $v \overset{\mathrm{v}}{\sim}_\beta v'$, we have $h[r(f) \mapsto v] \overset{\mathrm{H}}{\sim}_\beta ht[r'(f) \mapsto v']$. Finally, we have that $s \approx_{ht',lt',\beta} as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i]$, where $ht' = ht[r'(f) \mapsto v']$, by the definition of $as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i]$.

□

Proposition 2 (BC2BIR_{instr} - zero call-depth one-step preservation - error case)**.**
Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \Omega^k(m, pc', l_e, h_e)$. *Let* ht, lt, as, β *be such that:*

$$h \overset{\mathrm{H}}{\sim}_\beta ht \qquad l \overset{\mathrm{E}}{\sim}_\beta lt \qquad s \approx_{ht,lt,\beta} as \qquad \text{BC2BIR}_{instr}(pc, \mathit{instrAt}_P(m, pc), as) = (\ell, as')$$

Then, there exist unique $\vec{\lambda}', h'_e, l'_e$ *such that* $\langle ht, m, (pc, \ell), lt\rangle \xrightarrow{\vec{\lambda}'}_0 \Omega^k(m, pc', l'_e, h'_e)$ *with*

$$\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda}'_{proj} \quad h_e \overset{\mathrm{H}}{\sim}_\beta h'_e \quad l_e \overset{\mathrm{E}}{\sim}_\beta l'_e$$

Proof. Here again, we proceed by case analysis on the instruction at program point pc.

- **div** Here, only one case is possible: $\langle h, m, pc, l, (\mathit{Num}\ n_1){::}(\mathit{Num}\ 0){::}s\rangle \xrightarrow{\tau}_0 \Omega^{DZ}(m, pc, l, h)$. Let as, ht, lt be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $(\mathit{Num}\ n_1){::}(\mathit{Num}\ 0){::}s \approx_{ht,lt,\beta} e_1{::}e_2{::}as$. We have $\text{BC2BIR}_{instr}(pc, \mathtt{div}, e_1{::}e_2{::}as) = ([\mathtt{notzero(e_2)}], e_1/e_2{::}as)$. But $ht, lt \vDash e_2 \Downarrow (\mathit{Num}\ n'_2)$ with $(\mathit{Num}\ 0) \overset{\mathrm{v}}{\sim}_\beta (\mathit{Num}\ n'_2)$. Thus, $n'_2 = 0$ and $\langle ht, m, (pc, [\mathtt{notzero(e_2)}]), lt\rangle \xrightarrow{\tau}_0 \Omega^{DZ}(m, pc, lt, ht)$. Heaps and environments are not modified and both transitions are silent.

- **getfield** The execution reaches the error state. Hence, we have $\langle h, m, pc, l, \mathit{Null} :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)$. Let e, as, ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $\mathit{Null} :: s \approx_{ht,lt,\beta} e :: as$. We have $\text{BC2BIR}_{instr}(pc, \mathtt{getfield\ f}, e{::}as) = ([\mathit{notnull}(e)], e.f{::}as)$. By hypothesis on the stacks, we have that $ht, lt \vDash e \Downarrow \mathit{Null}$ and $\langle ht, m, (pc, [\mathit{notnull}(e)]), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, lt, ht)$.

- **putfield f** We have $\langle h, m, pc, l, v{::}\mathit{Null}{::}s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)$. Let e, e', as, ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $v{::}\mathit{Null}{::}s \approx_{ht,lt,\beta} e'{::}e{::}as$. We have $\text{BC2BIR}_{instr}(pc, \mathtt{putfield\ f}, e'{::}e{::}as) = ([\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := as_i; e.f := e'], as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i])$. But $v{::}\mathit{Null}{::}s \approx_{ht,lt,\beta} e'{::}e{::}as$, hence $ht, lt \vDash e \Downarrow \mathit{Null}$. Thus, $\langle ht, m, (pc, [\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := as_i; e.f := e']), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, lt, ht)$.

- **`invokevirtual`** We only treat here the case where the method returns *Void*. We have $\langle h, m, pc, l, v_1 :: \ldots :: v_n :: Null :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)$. Let $e_i, e, as, ht, lt, \beta$ be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $v_1 :: \ldots :: v_n :: Null :: s \approx_{ht,lt,\beta} e_1 :: \ldots :: e_n :: e :: as$. We have $\text{BC2BIR}_{instr}(pc, \texttt{invokevirtualC.m}', e_1 :: \ldots :: e_n :: e :: as) = ([\texttt{notnull(e)}; \texttt{t}^1_{\texttt{pc}} := \texttt{e}'_1; \ldots; \texttt{t}^{\texttt{m}}_{\texttt{pc}} := \texttt{e}'_{\texttt{m}}; \texttt{e.m}(\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})], as[\texttt{t}^{\texttt{j}}_{\texttt{pc}}/\texttt{e}'_{\texttt{j}}])$. $ht, lt \vDash e \Downarrow Null$. Thus, $\langle ht, m, (pc, [\texttt{notnull(e)}; \texttt{t}^{\texttt{j}}_{\texttt{pc}} := e'_j; \texttt{e.m}(\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})]), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, lt, ht)$.

- **`constructor C`** We have $\langle h, m, pc, l, v_1 :: \ldots :: v_n :: Null :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, l, h)$. Let $e_i, e, as, ht, lt, \beta$ be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $v_1 :: \ldots :: v_n :: Null :: s \approx_{ht,lt,\beta} e_1 :: \ldots :: e_n :: e :: as$. By the hypothesis on the stacks, we know that $\texttt{e} \neq UR^{\texttt{C}}_{\texttt{pc}'}$ since `e` should evaluate to *Null*. Then, $\text{BC2BIR}_{instr}(pc, \texttt{constructorC}, e_1 :: \ldots :: e_n :: e :: as) = ([\texttt{notnull(e)}; \texttt{t}^1_{\texttt{pc}} := \texttt{e}'_1; \ldots; \texttt{t}^{\texttt{m}}_{\texttt{pc}} := \texttt{e}'_{\texttt{m}}; \texttt{e.super}(\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})], as[\texttt{t}^{\texttt{j}}_{\texttt{pc}}/\texttt{e}'_{\texttt{j}}])$. But $ht, lt \vDash e \Downarrow Null$. Thus, $\langle ht, m, (pc, [\texttt{notnull(e)}; \texttt{t}^{\texttt{j}}_{\texttt{pc}} := e'_j; \texttt{e.super}(\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})]), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}(m, pc, lt, ht)$.

□

With the two above propositions, we can now show that the algorithm given in Figure 5.2 preserves zero-call depth one-step transitions.

Proposition 3 (BC2BIR - zero call-depth one-step preservation - normal case)**.**
Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', m, pc', l', s'\rangle$ *and* ht, lt, β *are s.t.*

$$h \overset{\text{H}}{\sim}_\beta ht \qquad l \overset{\text{E}}{\sim}_\beta lt \qquad s \approx_{ht,lt,\beta} \texttt{AS}_\texttt{in}[\texttt{m}, \texttt{pc}]$$

Then there exist unique $ht', lt', \vec{\lambda}'$ *such that* $\langle ht, m, (pc, \texttt{IR}[\texttt{m}, \texttt{pc}]), lt\rangle \overset{\vec{\lambda}'}{\Rightarrow}_0 \langle ht', m, (pc', \texttt{IR}[\texttt{m}, \texttt{pc}']), lt'\rangle$ *with:*

$$h' \overset{\text{H}}{\sim}_\beta ht' \qquad l' \overset{\text{E}}{\sim}_\beta lt' \qquad \vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda}'_{proj} \qquad s' \approx_{ht',lt',\beta} \texttt{AS}_\texttt{in}[\texttt{m}, \texttt{pc}']$$

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', rv\rangle$ *and* $s \approx_{ht,lt,\beta} \texttt{AS}_\texttt{in}[\texttt{m}, \texttt{pc}]$ *then there exists a unique* $\langle ht', rv'\rangle$ *such that:* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda}'}{\Rightarrow}_0 \langle ht', rv'\rangle$, *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda}'_{proj}$, $h' \overset{\text{H}}{\sim}_\beta ht'$ *and* $rv \overset{\text{V}}{\sim}_\beta rv'$.

Proof. Suppose $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', m, pc', l', s'\rangle$ and ht, lt, β are such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} \texttt{AS}_\texttt{in}[\texttt{m}, \texttt{pc}]$. What differs from basic BC2BIR_{instr} transformation is that (i) the entry abstract stack is not always transmitted as it is from one instruction to its successors and (ii) additional assignments might be preprended to the BIR instruction basically generated.

The proof is hence organized as follows. We first have to show that s and $\texttt{as}_\texttt{in}$ (the actual abstract stack used in the basic transformation) keep related in the possibly modified environment. This intermediate result makes us able to use Proposition 1. Finally, we must ensure that the transmitted abstract stack is related to s' with respect to the new BIR heap and environment obtained by executing the basic BIR instructions.

First, we show that s and $\texttt{as}_\texttt{in}$ keep related with regards to the potentially modified environment. There are two cases whether `pc` is a branching point or not.

- If $\mathtt{pc} \in \mathit{jmpTgt}^{\mathtt{P}}_{\mathtt{m}}$, then $\mathtt{as}_{\mathtt{in}}$ is set to $\mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}]$. Now, assignments potentially generated by $\tilde{\mathtt{T}}\mathit{Assign}(\mathtt{pc}, \mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}])$ and $\mathtt{T}\mathit{Assign}(\mathit{succ}(\mathtt{pc}), \mathtt{AS}_{\mathtt{out}}[\mathtt{m},\mathtt{pc}])$ have to be taken into account. We show they do not alterare the stack relation between s and $\mathtt{as}_{\mathtt{in}}$. There are two cases according to whether successors of pc are branching points or not.
  - If none of them is a branching point, then no additional assignment is generated. Hence, the local environment *lt* is not modified and stacks keep related.
  - Suppose now some successors of pc are branching points (denoted by pcb). First, because pc is a branching point, the entry stack $\mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}]$ has been normalized.
    * if $\mathtt{pc} \neq \mathtt{pcb}$, the condition Line 15 is satisfied: none of the assigned $\mathtt{T}^{\mathtt{k}}_{\mathtt{pcb}}$ can be used in the elements of $\mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}] = \mathtt{as}_{\mathtt{in}}$. Hence, assignments do not modify the stack relation.
    * if $\mathtt{pc} = \mathtt{pcb}$, then the condition is not meet. In this case, the instruction at point pc is `goto pc`. Then assignments are $\tilde{\mathtt{T}}^{\mathtt{j}}_{\mathtt{pc}} := \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}; \ldots; \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}} := \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}$. If pc is not its only predecessor, $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}$ are already defined in the environment, and assignments do not modify their value. Now, if pc is its only predecessor, then $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}$ are not yet defined in the environment: the semantics of the program is stuck. However, the only case where this instruction is reacheable is when it is the first instruction of the method, and in this case, the stack is empty, hence no assignments are preprended to the BIR. Hence, the stack relation still holds.
- If $\mathtt{pc} \notin \mathit{jmpTgt}^{\mathtt{P}}_{\mathtt{m}}$, we distinguish two cases:
  - If $\mathit{succ}(\mathtt{pc}) \cap \mathit{jmpTgt}^{\mathtt{P}}_{\mathtt{m}} \neq \emptyset$, then $\mathtt{as}_{\mathtt{in}} = \mathit{newStack}(\mathtt{pc}, \mathit{AS}_{\mathit{in}}[m, pc])$. The stack relation has to be checked in the environment modified by $\tilde{\mathtt{T}}\mathit{Assign}(\mathtt{pc}, \mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}])$ and or $\mathtt{T}\mathit{Assign}(\mathit{succ}(\mathtt{pc}), \mathtt{AS}_{\mathtt{out}}[\mathtt{m},\mathtt{pc}])$.

    First, none of the assigned $\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}}$ are used in $\mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}]$: they are put onto the abstract stack only at point pc and the stack is normalised with a different naming convention on backward branches. Hence, stacks keep related with regards to the environment $\mathit{lt}[\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}} \mapsto v_k]$, where v_k is the value of the k^{th} element of $\mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}]$.

    Now, assignments generated by $\mathtt{T}\mathit{Assign}(\mathit{succ}(\mathtt{pc}), \mathtt{AS}_{\mathtt{out}}[m, pc])$ modify $\mathit{lt}[\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}} \mapsto v_k]$ but without changing the stack relation: all assigned $\mathtt{T}^{\mathtt{j}}_{\mathtt{pcb}}$ (where $\mathtt{pcb} \in \mathit{succ}(\mathtt{pc})$ is a branching point) have different identifiers from the $\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}}$.
  - Otherwise, $\mathtt{as}_{\mathtt{in}}$ is set to $\mathtt{AS}_{\mathtt{in}}[\mathtt{m},\mathtt{pc}]$. But no assignment is preprended to the BIR. Hence, the environment is not modified and stacks are related.

Thus $s \approx_{ht,\widetilde{lt},\beta} \mathtt{as}_{\mathtt{in}}$, where $\widetilde{lt}$ is equal to *lt* that has been potentially modified by assignments preprended to the BIR. In addition, the heap *ht* is not modified. The hypotheses of Proposition 1 are thus satisfied, and we obtain that:

$$\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda_1}}{\Longrightarrow}_0 \langle ht, m, (pc, \mathtt{instrs}), \widetilde{lt}\rangle \overset{\vec{\lambda_2}}{\Longrightarrow}_0 \langle ht', m, (pc', IR[m, pc']), lt'\rangle$$

where the intermediate state $\langle ht, m, (pc, \mathtt{instrs}), \widetilde{lt}\rangle$ is obtained by executing potential additional assignments. By Proposition 1, we have that resulting heaps and environments are related through β, and $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda_2}_{proj}$. Furthermore, $\vec{\lambda_1}$ is only made of temporary variable assignment events, hence $\vec{\lambda_1}_{proj}$ is empty, and $\vec{\lambda} \overset{!}{\sim}_\beta (\vec{\lambda_1}.\vec{\lambda_2})_{proj}$.

We conclude the proof by showing the transmitted abstract stack is related to s' with regards to ht', lt' and β. Here again, there are two cases:

- If $\mathtt{pc}'$ is not a branching point, then the transmitted abstract stack is $\mathtt{AS_{out}[m, pc]}$, resulting from the basic transformation BC2BIR$_{instr}$. The stack relation is here simply given by Proposition 1.
- If $\mathtt{pc}' \in jmpTgt^{\mathtt{P}}_{\mathtt{m}}$, the transmitted abstract stack is $newStackJmp(\mathtt{pc}', AS_{out}[m])$. All of the $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}'}$ have been assigned, but we must show that they have not been modified since then by the BIR instructions generated by BC2BIR$_{instr}$. An environment can be modified by BIR instructions that are either obtained by transforming a `store x` instruction, or instructions that could modify the value of $\mathtt{AS_{out}[m, pc]}$ elements (see Figure 5.1 for variable or field assignment). In the first case, the variable is used at BC level and is hence different from all $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}'}$. In the second case, temporary variables are $\mathtt{t}^{\mathtt{k}}_{\mathtt{pc}}$ and have also different identifiers.

Thus, we have $s' \approx_{ht',lt',\beta} \mathtt{AS_{in}[m, pc']}$. □

Proposition 4 (BC2BIR - zero call-depth one-step preservation - error case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \Omega^k(m, pc', l_e, h_e)$ *and* ht, lt, β *are s.t.* $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS_{in}[m, pc]}$. *Then there exist unique* $\vec{\lambda'}, ht_e, lt_e$ *such that* $\langle ht, m, (pc, \mathtt{IR[m, pc]}), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \Omega^k(m, pc', lt_e, ht_e)$ *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$ $\quad l_e \overset{\mathrm{E}}{\sim}_\beta lt_e \quad h_e \overset{\mathrm{H}}{\sim}_\beta ht_e$

Proof. Similar to Proposition 3, but using Proposition 2. □

Proposition 5 (BC2BIR - zero call-depth preservation - normal case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_0 \langle h', m, pc', l', s'\rangle$ *and* ht, lt, β *are such that* $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS_{in}[m, pc]}$.

Then there exist unique $ht', lt', \vec{\lambda'}$ *such that* $\langle ht, m, (pc, \mathtt{IR[m, pc]}), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', m, (pc', \mathtt{IR[m, pc']}), lt'\rangle$ *with* $h' \overset{\mathrm{H}}{\sim}_\beta ht'$, $l' \overset{\mathrm{E}}{\sim}_\beta lt'$, $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$ *and* $s' \approx_{ht',lt',\beta} \mathtt{AS_{in}[m, pc']}$.

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_0 \langle h', rv\rangle$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS_{in}[m, pc]}$ *then there exists a unique* $\langle ht', rv'\rangle$ *such that:* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', rv'\rangle$, *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$, $h' \overset{\mathrm{H}}{\sim}_\beta ht'$ *and* $rv \overset{\mathrm{v}}{\sim}_\beta rv'$.

Proof. Similar to Proposition 3, using an induction on the number of steps of the BC computation. □

Proposition 6 (BC2BIR - zero call-depth preservation - error case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_0 \Omega^k(m, pc', l_e, h_e)$ *and* ht, lt, β *are s.t.* $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS_{in}[m, pc]}$.

Then there exist unique $\vec{\lambda'}, ht_e, lt_e$ *such that* $\langle ht, m, (pc, \mathtt{IR[m, pc]}), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \Omega^k(m, pc', lt_e, ht_e)$ *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

Proof. As the error state is reached after a given number of normal execution steps, we use here Propositions 5 and 2. □

We now have to state propositions similar to Propositions 5 and 6, dealing with an arbitrary call-depth.

Proposition 7 (BC2BIR - multi-step preservation - normal case)**.**

Let $n \in \mathbb{N}$. Suppose that $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', m, pc', l', s'\rangle$ and ht, lt, β are such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist unique $ht', lt', \vec{\lambda'}$ and a unique β' extending β such that $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht', m, (pc', IR[m, pc']), lt'\rangle$ with $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_{\beta'} ht'$, $l' \overset{\text{E}}{\sim}_{\beta'} lt'$ and $s' \approx_{ht',lt',\beta'} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}']$.

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', rv\rangle$ and $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$ then there exists a unique $\langle ht', rv'\rangle$ such that: $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht', rv'\rangle$, with $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_\beta ht'$ and $rv \overset{\text{V}}{\sim}_\beta rv'$.

Proposition 8 (BC2BIR - multi-step preservation - error case)**.**

Let $n \in \mathbb{N}$. Suppose that $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \Omega^k(m, pc', l_e, h_e)$ and ht, lt, β are such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist unique $\vec{\lambda'}, ht_e, lt_e$ and a unique β' extending β such that $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \Omega^k(m, pc', lt_e, ht_e)$ with $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$, $h_e \overset{\text{H}}{\sim}_{\beta'} ht_e$ and $l_e \overset{\text{E}}{\sim}_{\beta'} lt_e$.

The proof of Propositions 7 and 8 will be done by strong induction on the call-depth. For the sake of clarity, let $\mathcal{P}(n, m)$ and $\mathcal{P}_\Omega(n, m)$ denote respectively Propositions 7 and 8, where n is the call depth and m denotes the method that is being executed. Here, Propositions 5 and 6 are respectively the base cases $\mathcal{P}(0, m)$ and $\mathcal{P}_\Omega(0, m)$. Concerning induction cases, we use an induction on the number of steps of the BC computation. The base cases are shown using Proposition 9 and 10.

Proposition 9 (BC2BIR - one-step preservation - normal case)**.**

Let $n \in \mathbb{N}$. Suppose that $\mathcal{P}(k, m)$ for all m and $k < n$. Suppose that $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\rightarrow}_n \langle h', m, pc', l', s'\rangle$. Let ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist unique ht', lt' and a unique β' extending β such that $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht', m, (pc', IR[m, pc]), lt'\rangle$ with $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_{\beta'} ht'$, $l' \overset{\text{E}}{\sim}_{\beta'} lt'$ and $s' \approx_{ht',lt',\beta'} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\rightarrow}_n \langle h', rv\rangle$ and $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$ then there exists a unique $\langle ht', rv'\rangle$ such that: $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\rightarrow}_n \langle ht', rv'\rangle$, with $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_\beta ht'$ and $rv \overset{\text{V}}{\sim}_\beta rv'$.

Proof. The case for return execution states is Proposition 3. For non-returning execution states, we use the same proof structure than for Proposition 3. Arguments are the same for showing that the stack relation between s and $\texttt{as}_{\texttt{in}}$ is preserved by the potential additional assignments. Hence, we have $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda_1}}{\Rightarrow}_0 \langle ht, m, (pc, instrs), \widetilde{lt}\rangle$ with $\widetilde{lt}$ is the new environment, $l \overset{\text{E}}{\sim}_\beta \widetilde{lt}$ and $s \approx_{ht,\widetilde{lt},\beta} \texttt{as}_{\texttt{in}}$. The instruction list `instrs` is obtained by the basic transformation BC2BIR_{instr}. We now have to match the BC execution step. We proceed by case analysis on the BC instruction $instrAt_P(m, pc)$.

- **`constructor C`** Here, $s = V{::}(Ref\ r){::}s'$. By case analysis on the semantic rule that is used and on the condition on `C`, we have four cases. In the first possible case, we have $h(r) = o_t$ with

$t = \widetilde{\mathtt{C}_\mathtt{j}}$ where $C \neq Object$. Let h_1 be the heap such that $h_1 = h[r \mapsto o_{\widetilde{\mathtt{C}}}]$. We have that

$$\langle h_1, C.init, 0, [\mathtt{this} \mapsto (Ref\ r), \mathtt{x_1} \mapsto v_1, \ldots, \mathtt{x_n} \mapsto v_n], \varepsilon\rangle \overset{\vec{\lambda_2}}{\Rightarrow}_{n-1} \langle h_2', Void\rangle$$

Only one form of abstract stack is compatible with the stack relation we just show. Thus, $\mathtt{as_{in}}$ is of the form: $e_1{::}\ldots{::}e_n{::}UR^{\mathtt{C}}_{\mathtt{j}}{::}as$ and $v_1{::}\ldots{::}v_n{::}(Ref\ r){::}s \approx_{ht,\widetilde{lt},\beta} \mathtt{as_{in}}$.

We have $\text{BC2BIR}_{instr}(pc, \mathtt{constructor\ C}, e_1 :: \ldots :: e_n :: UR^{\mathtt{C}}_{\mathtt{j}} :: as) = ([\mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} := e_1'; \ldots; \mathtt{t}^{\mathtt{m}}_{\mathtt{pc}} := e_m'; \mathtt{t}^{\mathtt{0}}_{\mathtt{pc}} := new\ C(e_1, \ldots, e_n)], as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/\ \mathtt{e_i}][\mathtt{t}^{\mathtt{0}}_{\mathtt{pc}}/UR^{\mathtt{C}}_{\mathtt{j}}])$

Let us follow the semantics of BIR. Let $(Ref\ r') = newObject(C, ht)$ and $ht_1' = ht_1[r' \mapsto (\lambda f.init(f))_{\widetilde{\mathtt{C}}}$.

By hypothesis, stacks are related. Thus, for all i, $ht_1', \widetilde{lt} \vDash e_i \Downarrow v_i'$ and $v_i \overset{\mathrm{v}}{\sim}_\beta v_i'$ and constructors are called on related environments.

We extend β to β' to take r' into account: $\beta'(r) = r'$, and we have that $h_1 \overset{\mathrm{H}}{\sim}_{\beta'} ht_1'$: objects pointed to by r and r' have the same initialization status, their class is equal, and each of their field has default values (we have $h_1(r) = (\lambda f.init(f))_{\widetilde{\mathtt{C}}}$ since before the call to the constructor, nothing can be done on this object). Hence, both constructors are called on related initial configurations.

We can now apply $\mathcal{P}(n-1, C.init)$. Hence, we get that there exists β'' extension of β' relating the two resulting heaps and constructor execution traces.

Now, from m point of view, the traces are related: $r \leftarrow C.init(v_1, \ldots, v_n) \overset{!}{\sim}_{\beta''} r' \leftarrow C.init(v_1', \ldots, v_n')$ and the remainder of both traces are related by $\mathcal{P}(n-1, C.init)$.

The heaps have been shown to be related w.r.t β'', and the environments keep related (only fresh temporary variables have been introduced).

We now have to show the stack relation. Every $UR^{\mathtt{C}}_{\mathtt{j}}$ is substituted with the new temporary $\mathtt{t}^{\mathtt{t_0}}_{\mathtt{pc}}$. But it is now evaluated to r', which is an related value to r w.r.t β''. Now, concerning the storing of stack elements reading fields, they have been introduced before the execution of the constructor and none of them could have been modified. Their value hence stay related to the corresponding element of the concrete stack after the constructor call.

For other rules, the proof is similar (for the last one, references are already related through the bijection, which does not need to be extended). Special care has to be taken with assertions: they do not fail, since the constructor call is not faulty.

- **`invokevirtual m`** We proceed similarely. The current objects are already initialized, hence the bijection is not extended. We can distinguish between void and value-returning methods: this information is available in the bytecode program, and we use the relation over the last label of method execution trace given by $\mathcal{P}(n-1, mc)$ to deduce that the BIR method has the same signature.

Similarely to Proposition 3, we conclude by showing the transmitted abstract stack $\mathtt{AS[m, pc']}$ is related to s'. Here again, arguments are the same. □

Proof of Proposition 7. We proceed by strong induction on n.

- As already said, $\mathcal{P}(0, m)$ is exactly Proposition 5.

- Now suppose that $\forall m$ and $k < n, \mathcal{P}(k, m)$. We show $\mathcal{P}(n, m)$ by induction on np, the number of steps of the BC computation.
  - If $np = 1$, then we use Proposition 9.
  - Suppose now that $np > 1$ and $\mathcal{P}(n, m)$ holds for all np'-step computations with $np' < np$. We only treat here non-returning execution states. Thus, by the definition of call-depth indices, we have that

$$\langle h, m, pc, l, s\rangle \underset{(1)}{\overset{\vec{\lambda_1}}{\Rightarrow_{n_1}}} \langle h_1, m, pc_1, l_1, s_1\rangle \underset{(2)}{\overset{\vec{\lambda_2}}{\longrightarrow_{n_2}}} \langle h_2, m, pc_2, l_2, s_2\rangle$$

    with $n_1 + n_2 = n$. Step (1) of the computation is made of $np - 1$ steps, and step (2) is a one-step transition. In both cases, we can use the induction hypothesis (respectively on $np - 1$ and 1) to get the result.

□

Proposition 10 (BC2BIR - one-step preservation - error case)**.**

Let $n \in \mathbb{N}$. Suppose that $\mathcal{P}_\Omega(k, m)$ for all m and $k < n$. Suppose that $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\rightarrow}_n \Omega^k(m, pc', l_e, h_e)$. Let ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \mathtt{AS_{in}[m, pc]}$.

Then there exist a unique β' extending β and unique lt_e, ht_e such that $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \Omega^k(m, pc', lt_e, ht_e)$ with $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$.

Proof. Here again the structure of the proof follows the one of Proposition 3. Suppose that $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\rightarrow}_n \Omega^k(m, pc', l_e, h_e)$. By similar arguments than before, we get that $s \approx_{ht,\widetilde{lt},\beta} \mathtt{as_{in}}$. To match the BC computation step, we now proceed by case analysis on the BC instruction at program point `pc`.

- **`constructor C`** Here, four error computation steps are possible, according to the initialization status of the object pointed to by the reference on which the constructor is called and whether `C` is `Object` or not.

  If the initialization tag of the object is $\widetilde{\mathtt{C_j}}$. A similar reasoning than in the proof of Proposition 9 can be done to deduce the form of the abstract stack $\mathtt{as_{in}}$ and to state that initial configurations on which the constructor executions start are related (the bijection is extended to β' to take into account the newly allocated object in the BIR heap).

  Now, the execution of the BC constructor fails in the state $\Omega^k(C.init, pc', l_e, h_e)$. We use here proposition $\mathcal{P}_\Omega(n-1, \mathtt{C.init})$ to obtain β'' extending β' and that the BIR constructor execution fails too in $\Omega^k(C.init, pc', lt_e, ht_e)$, with $h_e \overset{\text{H}}{\sim}_{\beta''} ht_e, l_e \overset{\text{E}}{\sim}_{\beta''} lt_e$ and traces related with regards to β''.

  Traces are related also from the point of view of the method `m` (the projection of traces preserves their relation). Finally, error states are related (their program point parameter is `pc`, environments are related by hypothesis, and heaps are related by $\mathcal{P}_\Omega(n-1, \mathtt{C.init})$).

  Suppose now the reference pointing to the object on which the constructor is called is already tagged as being initialized. Here, the bijection does not need to be extended. Second, the τ event at the head of the BC trace is matched by the normal execution of the assertion generated: the initialization of the object is ongoing, thus the reference pointing to it is in the domain of β, and its related value is also a non-null reference. The rest of the proof is similar.

- **`invokevirtual`** Here, the reference pointing to the object on which the method is called is initialized. Hence the bijection does not need to be extended. Arguments are similar to the previous case.

□

Proof of Proposition 8. We proceed by strong induction on n.

- As already said, $\mathcal{P}_\Omega(0, m)$ is exactly Proposition 6.
- Now suppose that $\forall m$ and $k < n, \mathcal{P}_\Omega(k, m)$. We show $\mathcal{P}_\Omega(n, m)$ by case analysis on np, the number of steps of the BC computation.
  - If $np = 1$, then we use Proposition 10.
  - Suppose now that $np > 1$. Here again, only non-returning execution states are treated. Thus, by the definition of call-depth indices, we have that

$$\langle h, m, pc, l, s\rangle \underset{(1)}{\overset{\vec{\lambda_1}}{\Rightarrow_{n_1}}} \langle h_1, m, pc_1, l_1, s_1\rangle \underset{(2)}{\overset{\vec{\lambda_2}}{\longrightarrow_{n_2}}} \langle h_2, m, pc_2, l_2, s_2\rangle$$

    with $n_1 + n_2 = n$. For the step (1) of this computation, we apply Proposition 7. Then, we are back in the base case for step (2), and we use Proposition 10.

□

From this proposition, we can derive the final theorem of semantics preservation. For the sake of simplicity, we only consider the main method of the program. We denote by BC2BIR(P) the mapping of BC2BIR(P, m) on all the methods m of P.

Theorem 1 (Semantics preservation - normal case)**.**
Let P be a BC *program and P' be its* BIR *version P' = BC2BIR(P). Let h_0 denote the empty heap and l_0 an arbitrary environment. Let $\langle h, \texttt{main}, pc, l, s\rangle$ be a reachable state of P, i.e.*

$$\langle h_0, \texttt{main}, 0, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h, \texttt{main}, pc, l, s\rangle$$

Then, there exist a heap ht, an environment lt and partial bijection β such that

$$\langle h_0, \texttt{main}, (0, instrsAt_{P'}(main, 0)), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht, \texttt{main}, (pc, instrsAt_{P'}(main, pc)), lt\rangle$$

with $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

Under the same hypotheses, if $\langle h_0, \texttt{main}, 0, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h, rv\rangle$ Then, there exist a heap ht and partial bijection β such that $\langle h_0, \texttt{main}, (0, instrsAt_{P'}(main, 0)), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht, rv'\rangle$ with $h \overset{\text{H}}{\sim}_\beta ht$, $rv \overset{\text{V}}{\sim}_\beta rv'$ and $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

Proof. We use Proposition 7. Starting states fullfill the hypotheses since the heaps are empty, the environments are equal and the translation algorithm begins on an empty abstract stack. □

A similar theorem is obtained about executions leading to an error state.

Theorem 2 (Semantics preservation - error case)**.**
Let P be a BC *program and P' be its* BIR *version P' = BC2BIR(P). Let h_0 denote the empty heap and l_0 an arbitrary environment. If $\langle h_0, \texttt{main}, 0, l_0, \varepsilon\rangle \stackrel{\vec{\lambda}}{\Rightarrow}_n \Omega^k(main, pc', l_e, h_e)$, then there exist a partial bijection β and unique ht_e, lt_e such that $\langle h_0, \texttt{main}, (0, instrsAt_{P'}(main, 0)), l_0\rangle \stackrel{\vec{\lambda'}}{\Rightarrow}_n \Omega^k(main, pc', lt_e, ht_e)$ with $\vec{\lambda} \stackrel{!}{\sim}_\beta \vec{\lambda'}_{proj}$ $h_e \stackrel{\text{H}}{\sim}_\beta ht_e$ $l_e \stackrel{\text{E}}{\sim}_\beta lt_e$.*

We presented the translation algorithm BC2BIR and proved it preserves the semantics of the initial BC program: the BIR program BC2BIR(P) simulates the program P, and their execution trace are related. This result gives us some guarantees about the IR of a given program, and brings the hope that static analyses results obtained about the BIR program could be shifted back to the initial BC program. In the next section we make a first step towards this "safety preservation", through three examples.

6.4 Application example

In this section, we aim at demonstrating how the result of a static analysis on a BIR program can be translated back to the initial BC program. We illustrate this on three examples of safety property.

Null-pointer error safety In [HJP08], Hubert *et al* propose a null-pointer analysis on a subset of Java source and show it correct. Their analysis is based on abstract interpretation and infers, for each field of each class of the program, whether the field is definitely non-null or possibly null after object initialization. Adapting their definition for BC, we obtain the following safety property:

Definition 9 (BC Null-Pointer error safety)**.** *A* BC *program is said to be null pointer error safe if, for all pc', $\langle h_0, \texttt{main}, 0, l_0, \varepsilon\rangle \stackrel{\vec{\lambda}}{\Rightarrow}_n s$ implies $s \neq \Omega^{NP}(main, pc', l_e, h_e)$ where h_0 is the empty heap and l_0 is the empty environment (the main method is assumed to have no parameters).*

Hubert later proposed a Bytecode version for the analysis in [Hub08]. It uses expression reconstruction to improve the accuracy of the analysis. Additionnaly, as it deals with object initialization, this analysis definitely needs to reconstruct the link between freshly allocated reference in the heap and the call of the constructor on it. The BIR language provides this information, and this would have eased the analysis. The safety property shifted to BIR is defined as follows:

Definition 10 (BIR Null-Pointer error safety)**.** *A* BIR *program P is said to be null pointer error safe if, for all pc', $\langle h_0, \texttt{main}, (0, instrsAt_P(main, 0)), l_0\rangle \stackrel{\vec{\lambda}}{\Rightarrow}_n s$ implies $s \neq \Omega^{NP}(main, pc', l_e, h_e)$ where h_0 is the empty heap and l_0 is the empty environment.*

Suppose now given a correct BIR analysis, with regards to the Definition 10. As a direct consequence of Theorem 1, we can show that if a program is deemed safe by the BIR analysis, then the initial BC program is also null-pointer error safe.

Bounded field value Suppose we want to ensure that in a given program, the integer field `f` of each object of class `C` always has a value within the interval $[0, 10]$. Interval analyses are the typical solution for this problem: they determine at each program point an interval in which variables and object fields take their values. Then, the analysis would check that at every interest program points, i.e. `f` fields assignments, the value given to the field is in an appropriate interval. The precision of such

an analysis is increased with the help of symbolic expressions. Hence, it would be easier to perform this analysis on the BIR version of the program. Here we prove that if the program BC2BIR(P) is shown to satisfy this safety property (by mean of a correct static analysis), then the initial BC program P is also safe. The proof is performed at a rather intuitive level of details. Further formalization on this is ongoing work.

The problem can be formulated this way. Let $\mathcal{L}$ be one of our two languages BC or BIR. A program p is safe if and only if $[\![p]\!]_{\mathcal{L}} \subseteq Safe_{\mathcal{L}}$, where $[\![p]\!]_{\mathcal{L}}$ is the set of all reachable states of p (as defined in Theorem 1) and $Safe_{\mathcal{L}}$ is the set of all states in $State_{\mathcal{L}}$ that are safe:

Definition 11 (Safe states)**.**
Let p be a $\mathcal{L}$ program. A state $s \in State_{\mathcal{L}}$ is in $Safe_{\mathcal{L}}$ if:

- $\langle h_0, main_p, 0, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n s$, *where h_0 is empty, and l_0 is an arbitrary environment,*
- $\vec{\lambda} = \lambda_1\lambda_2\ldots\lambda_n$ *and* $\forall i, Safe_E(\lambda_i)$

where $Safe_E(\lambda) \Leftrightarrow \lambda \notin \{r.\mathtt{f} \leftarrow (Num\ n) \mid r \in Reference, n \in [-\infty; -1] \cup [11; +\infty]\}$

Note that safe states could here include error states, as the safety property only deals with field assignments. A static analysis can be seen as a function $prog_{\mathcal{L}} \rightarrow \{fail, ok\}$, that takes a program written in $\mathcal{L}$ as argument, and returns $fail$ or ok depending on whether an invalid value can be assigned to an `f` field. A given analysis $Analyse$ is said to be correct if the following holds:

$$\forall p \in prog_{\mathcal{L}}, Analyse(p) = ok \Rightarrow [\![p]\!]_{\mathcal{L}} \subseteq Safe_{\mathcal{L}}$$

Now, suppose we are given a correct analysis on BIR, $Analyse_{\text{BIR}}$. Let P be a BC program and $P' = \text{BC2BIR}(P)$ its BIR version. Suppose that $Analyse_{\text{BIR}}(P') = ok$, and hence that the BIR program is safe. We show that P is also safe. Let s be a reachable state of P, i.e. $\langle h_0, main, 0, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n s$ with h_0 the empty heap, l_0 an arbitrary environment, and $\vec{\lambda} = \lambda_1\lambda_2\ldots\lambda_n$. Applying Theorems 1, we get that there exist a state s', a partial bijection β and a trace $\vec{\lambda'}$ such that

$$\langle h_0, main, (0, instrsAt_{P'}(main, 0)), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n s' \text{ with } \vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$$

But P' is safe, hence for all $i \in [1; n]$, we have that $Safe_E(\lambda'_i)$ where $\vec{\lambda'}_{proj} = \lambda'_1\ldots\lambda'_n$. Take j and n such that $\lambda_j = r_j.\mathtt{f} \leftarrow (Num\ n_j)$. By the definition of $\overset{!}{\sim}_\beta$, we have that $\lambda'_j = r'_j.\mathtt{f} \leftarrow (Num\ n_j)$. We know that $Safe_E(\lambda'_j)$, thus $n \in [0; 10]$ and $Safe_E(\lambda_j)$. Hence, P is safe.

File access safety Our claim is that the correctness propositions of the last section suits well safety properties expressed as FSM languages. To illustrate this, we take the example of checking whether a given program correctly accesses a given file: every writing to it is performed after having it opened, and before closing it. Here again, an analysis on the BIR version of the program appears to be more easy to performed, thanks to the method call folding – the content of the stack does not need to be analysed anymore.

Let $\mathcal{A} = (\Sigma, Q, \delta, \mathcal{I}, \mathcal{F})$ be a FSM, as is standardly defined. Transitions in δ relates states of Q and are labelled with elements of the alphabet Σ. Initial and final states are respectively in subsets $\mathcal{I}$ and $\mathcal{F}$ of Q. The safety property we are interested in can be expressed as a language $\mathcal{L}(\mathcal{A})$ of the FSM $\mathcal{A}$ given in Figure 6.1, where the entry word is extracted from the execution trace of the program. More formally,

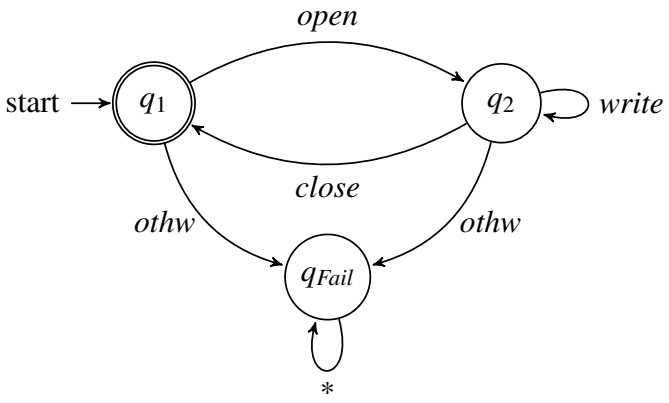

Figure 6.1: The FSM $\mathcal{A}$ accepting only safe file accesses

Definition 12 (File access safety)**.**
During the execution of the BIR *program P, the object file pointed to in the heap by the reference (Ref r) is accessed safely if, whenever* $\langle h_0, \texttt{main}, (0, instrsAt_P(main, 0)), l_0\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle ht, Void\rangle$ *then* $ToSigma(\vec{\lambda}_{r\ file}) \in \mathcal{L}(\mathcal{A})$

where the projection $\vec{\lambda}_{r\ file}$ is

$$\vec{\lambda} \text{ restricted to events in } \{r.File.open(), r.File.write(v_1, \ldots, v_n), r.File.close()\}$$

and the function *ToSigma* is defined as:

$$ToSigma(\lambda_1.\vec{\lambda}) = \begin{cases} open.ToSigma(\vec{\lambda}) & \text{if } \lambda_1 = [r.File.open()] \\ write.ToSigma(\vec{\lambda}) & \text{if } \lambda_1 = [r.File.write(v_1, \ldots, v_n)] \\ close.ToSigma(\vec{\lambda}) & \text{if } \lambda_1 = [r.File.close()] \end{cases}$$

It follows from Theorem 1 that if a BIR program execution trace $\vec{\lambda}$ is safe, then the initial BC program executes producing a trace $\vec{\lambda'}$ and that $ToSigma(\vec{\lambda'}_{\beta^{-1}(r)\ file}) \in \mathcal{L}(\mathcal{A})$ (events of trace $\vec{\lambda'}$ have been filtered to file accesses to the object pointed to by the corresponding reference in the BC heap).

In this section, we demonstrate on an example how the semantics preservation property of BC2BIR algorithm could help shifting the results of a given analysis on a BIR program back to the initial BC program. Our claim is that a similar reasoning could be applied to many other analyses. We also believe that there exist analyses for which nothing can be said about the initial BC program, given its result on the BIR version of the program. A direct example would be an analysis that deals with the allocation history: the bijection β is never made explicit, we only ensure its existence. Investigating this intuition, and further formalizing this "safety shifting" is left as future work.

So far, we formalized both source and target languages, as well as the transformation itself. In this chapter, we proved that the semantics of the initial BC program is preserved by the transformation. The proof argument is based on a simulation argument, which is a rather classical technique to prove the correctness of program transformations. The notion of simulation is defined relatively to a partial bijection that relates both heaps throughout the execution of P and BC2BIR(P).

Chapter 7

Implementation

BC2BIR has been implemented in a prototype that takes as input a class file and returns an intermediate representation for all the methods in the class. The prototype is written in OCaml, using the class file parser JavaLib [Jt07]. It is available online[1] for evaluation using a web interface.

While the formalization of the previous section considers a strict subset of Java, our prototype handles the full set of Java bytecodes. Some extensions have been necessary. When dealing with 64 bits values, the behavior of polymorphic stack operations like `pop2` requires to recover type information in order to predict if the current operand stack will start with two values of 32 bits or one value of 64 bits. This information is easily computed in one pass, and we obtain by the same token the size of the operand stack at each program point and the set of join points. The transformation we have formalized in unsound with respect to multi-threading because it keeps symbolic stack field expressions that may be invalidated by concurrent threads. It is straightforward to make the transformation correct, by systematically replacing such expressions by an extra variable and generate a suitable assignment.

The algorithm presented in the previous sections has been optimised in order to (i) compute the previous information on the fly (except the set of branching points that still requires a preliminary scan), (ii) only keep in memory the symbolic operand stack of the current program point, (iii) simplify some trivial sequences like `t = new A(); x = t` where the temporary variable `t` is unnecessary, (iv) reuse some variable names when fresh variable names are needed. We rely on a strong invariant on the symbolic operand stack that says that any identifiers of temporary variable that is not in the symbolic operand stack is dead at the current program point.

As explained in in the previous sections, the transformation may fail for some programs that have nevertheless passed the BCV. However we have run the translator on the 82352 classes (corresponding to 609209 methods) of the Eclipse distribution and the only cases (1793 methods) where the translation has failed were due to the presence of subroutines. These could be treated using standard subroutine inlining techniques.

In the remainder of this section we present our experimental of the performance of the tool with respect to transformation time, the compactness of the obtained code and the impact on static analysis precision. Our experiments have been performed on the Java runtime environment (`rt.jar`), a scientific computation library (`jscience.jar`, 517 classes), the Java Compiler Compiler (`javacc.jar`, 154 classes) and the Soot library (`soot.jar`, 2272 classes). These experiments have been performed on a MacBook Pro with 3.06 Ghz Intel Core 2 Duo processor and 4 Go 1067 MHz DDR3 RAM.

[1] `http://www.irisa.fr/celtique/ext/bir`

Transformation time

In order to be usable for lightweight verification, our transformation algorithm must be efficient. It is mainly for this reason that we do not rely on iterative techniques for transformation. We have compared the transformation time of our tool with respect to the transformation time of the Soot framework when transforming class files into Grimp representations. The results are given in Figure 7.1. For each benchmark library, we compare our running time for transforming all methods with the running time of the Soot tool when transforming[2] class files into their Grimp representation. The Soot tool provides several time measures. We only retain three phases in our results since the other phases (like local variables type inference) are not directly relevant. The retained phases are (i) generation of Jimple 3-address code (P1), (ii) local def/use static analysis that Soot uses to simplify its naive 3-address code (P2), (iii) aggregation of expressions to build Grimp syntax (P3). These experiments show that our Ocaml tool (both in bytecode and native mode) is very competitive with respect to the Soot framework, in terms of computation efficiency. This is mainly due to the non-iterative nature of our algorithm.

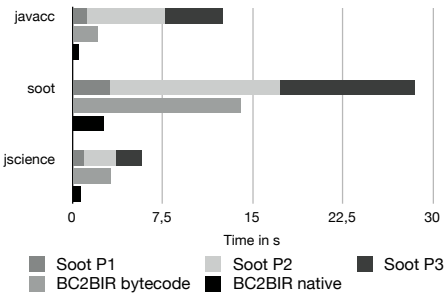

Figure 7.1: Efficiency of the transformation compared with the Soot Framework

Compactness of the obtained code

Intermediate representations rely on temporary variables in order to remove the use of operand stack and generate side-effect free expressions. The major risk here is an explosion in the number of new variables when transforming large programs.

In practice our tool stays below doubling the number of local variables. Figure 7.2 presents the percentage of local variable increase for each method of our benchmarks, sorting each result with respect to the size of the methods. Numbers in brackets represent here the number of bytecodes in the methods. The number of new variables increases with the size of the methods but stays manageable. We distinguish two kinds of temporary variables: those which are introduced for control flow joins with non-empty operand stacks, and all the other variables. The first category is particularly prominent in our measures. We believe this number could be reduced by using stan-

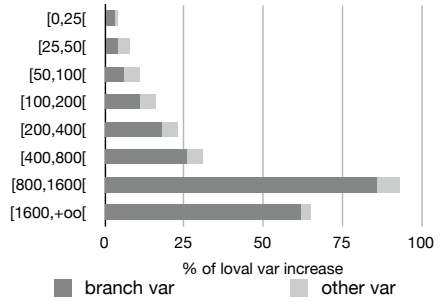

Figure 7.2: Measure of the local variable increase

[2]The transformation is without any optimisation option.

dard optimizations techniques, as those employed by Soot, but this will then require to iterate on each method.

We have made a direct comparison with the Soot framework to see how our tool compares with respect to the increase in local variables. Figure 7.3 presents two measures. For each method of our benchmarks we count the number N_{BC2BIR} of local variables in our IR code and the number N_{soot} of local variables in the Soot Jimple code. A direct comparison with Grimp is difficult because the language uses expressions with side-effects. For the purpose of this comparison, we generate with our tool only 3-address instructions. For each method we draw a point of coordinate $(N_{\text{BC2BIR}}, N_{\text{soot}})$ and see how the points are distributed around the first bisector. For the top diagram, Soot has been launched without default options. For the bottom diagram, we have added the local packer (`-p jb.lp enabled:true` Soot option) that reallocates local variables using use/def informations. Our transformation competes well, even when Soot uses this last optimization.

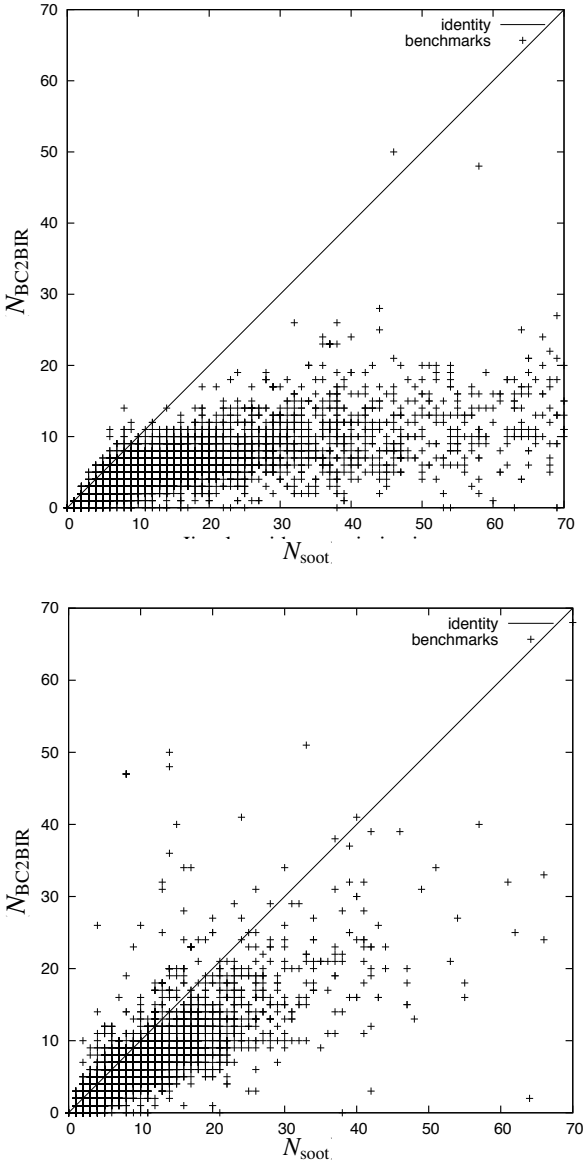

Figure 7.3: Local variables increase ratio

Impact on static analysis precision

In order to get an indication of the gain in analysis precision that the transformation obtains, we have conducted an experiment in which we compare the precision of an interval analysis before and after transformation. We have developed two intra-procedural interval analyses that track ranges of local variables of type `int`. The first analysis is directly made on `.class` files. At each program point we abstract integer local variables and operand stack elements by an interval. The lack of information between the operand stack and local variables makes it pointless to obtain extra information at conditional jump statement (using an abstract backward test [Cou99]). The second analysis is performed on the intermediate representation that is generated by our tool. This time, we benefit from the `if` to gain information at each conditional jump.

We have run these analyses on our benchmark libraries. Figure 7.4 presents two experimental comparisons. On the left part, we count the total number of `iload x` instructions (left column) and the number of these instructions for which the IR analysis finds a strictly more precise interval for the variable `x` than the bytecode analysis does. The result of `rt.jar` is similar but not displayed here for scaling considerations. Note that the IR analysis is sometimes less precise than the bytecode analysis,

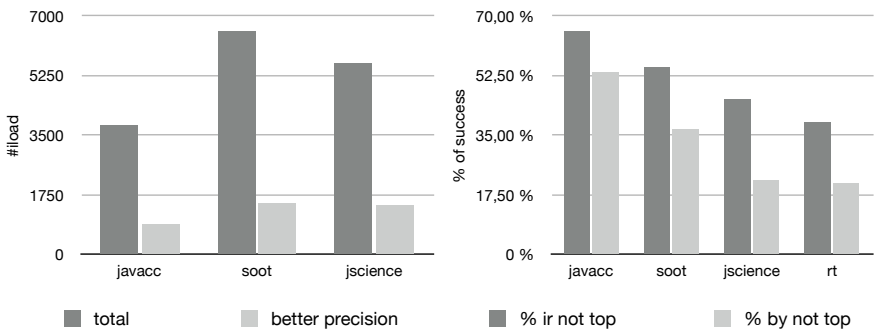

Figure 7.4: Impact on interval analysis precision

due to the intrinsically non-monotonic nature of widening operators. Among the 184927 `iload x` instructions we have encountered in our benchmark libraries, this particular case only happens 18 times. On the right part we take as success criterion the percentage of `iload x` instruction where the interval of `x` is not a $\top$ ("don't know") information. Again the IR analysis improves a lot the precision and, for some benchmarks, doubles the success rate.

Chapter 8

Conclusions

As noticed by Logozzo and Fähndrich [LF08], static analysis of bytecode programs is made specially difficult because of their intensive use of the operand stack. This paper provides a semantically sound, provably correct specification of a transformation of bytecode into an intermediate representation of bytecode that (i) removes the use of the operand stack and rebuilds tree expressions, (ii) makes more explicit the throwing of exception and takes care of preserving their order, (iii) rebuilds the initialization chain of an object with a dedicated instruction `x := new C(arg1, arg2, ...)`. In Section 6.4, we demonstrate how some soundness proofs of BIR static analysis can be translated back to the initial BC program. We illustrate this on several examples of safety properties.

The transformation has been designed to work in one pass in order to make it useful in a scenario of "lightweight bytecode analysis" applied to analyses other than type checking. It has been implemented in a tool that accepts full Java bytecode. Our benchmarks show clearly that the expected efficiency is obtained in practice.

The comparison with a static (interval) analysis of bytecode shows that the transformation obtains a important gain in analysis precision. This means that byte code transformation is a viable alternative to using more sophisticated domains, which is one of the solutions put forward by Fähndrich and Logozzo [LF08]. This is of interest also because the complexity of the transformation is easier to control than the cost of using the more sophisticated and hence computationally costly abstract domains.

Several extensions are possible. First we would like to extend this work into a multi-threading context. This is a challenging task, especially for the formalization part that must deal with the complex Java Memory Model. The second extension concerns mechanization of the development. We believe the current transformation would be a valuable layer on top of Bicolano, a formal JVM semantics that has been developped during the European MOBIUS project. We would like to use the Coq extraction mechanism to extract certified and efficient Caml code for the algorithm from a Coq formalization of the algorithm.

Bibliography

[AAG+07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *Proc. of ESOP'07*, volume 4421, pages 157–172. Springer-Verlag, 2007.

[BCF+99] M G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proc. of JAVA '99*, pages 129–141. ACM, 1999.

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3):273–291, 2006.

[BKPSF08] G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of proof pbligations for hybrid verification methods. In *Proc. of SEFM 2008*, pages 127–136. IEEE Computer Society, 2008.

[BN05] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[BR05] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proc. of TLDI '05*, pages 103–112, New York, NY, USA, 2005. ACM.

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[Cou99] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[FM99] Stephen N. Freund and John C. Mitchell. The type system for object initializatiion in the Java bytecode language. *ACM Trans. Program. Lang. Syst.*, 21(6):1196–1250, 1999.

[FM03] S. N. Freund and J. C. Mitchell. A type system for the Java bytecode language and verifier. *J. Autom. Reason.*, 30(3-4):271–321, 2003.

[GHM00] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Proc. of SAS'00*, pages 199–219. Springer-Verlag, 2000.

[HJP08] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of FMOODS 2008*, volume 5051 of *LNCS*, pages 132–149. Springer Berlin, June 2008.

[Hub08] Laurent Hubert. A Non-Null annotation inferencer for Java bytecode. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'08)*. ACM, November 2008.

[Jt07] The JavaLib team. *JavaLib*. Inria, March 2007. `javalib.gforge.inria.fr`.

[LF08] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Proc. of CC 2008*, pages 197–212. Springer LNCS 4959, 2008.

[Pro] The Jikes RVM Project. Jikes rvm - home page. `http://jikesrvm.org`.

[Ros03] E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[TL08] J.B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proc. of POPL'08*, pages 17–27. ACM Press, 2008.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of CASCON '99*. IBM Press, 1999.

[WCN05] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of BYTECODE 2005, Electronic Notes in Computer Science*, 2005.

[Wha99] J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[XX99] H. Xi and S. Xia. Towards array bound check elimination in Java tm virtual machine language. In *Proc. of CASCON '99*, page 14. IBM Press, 1999.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399