HAL
open science

A Provably Correct Stackless Intermediate Representation for Java Bytecode

Delphine Demange, Thomas Jensen, David Pichardie

To cite this version:

Delphine Demange, Thomas Jensen, David Pichardie. A Provably Correct Stackless Intermediate Representation for Java Bytecode. The 8th Asian Symposium on Programming Languages and Systems (APLAS), 2010, Shangai, China. inria-00537815

HAL Id: inria-00537815

https://inria.hal.science/inria-00537815v1

Submitted on 19 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Provably Correct Stackless Intermediate Representation for Java Bytecode

Delphine Demange[1], Thomas Jensen[2], and David Pichardie[2]

[1] ENS Cachan Antenne de Bretagne / IRISA, France
[2] INRIA, Centre Rennes - Bretagne Atlantique, Rennes, France

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. This paper provides such a bytecode transformation, describes its semantic correctness and evaluates its performance. 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. The correctness of this transformation is proved with respect to a relation on execution traces taking into account that the object allocation order is not preserved by the transformation.

1 Introduction

Several optimization and analysis tools for Java bytecode work on an *intermediate representation* (IR) of the bytecode that makes analyses simpler [3, 14]. Using such transformations may simplify the work of the analyser but the overall correctness of the analysis now becomes dependent on the semantics-preserving properties of the transformation. 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). Surprisingly, the semantic foundations of these bytecode transformations have received little attention. The contribution of this paper is to propose a transformation which at the same time is efficient (in terms of transformation time and produced code) and has a formal correctness proof. The long-term goal motivating this work 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 [11]. 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.

This paper provides a semantically sound, provably correct transformation of bytecode into an intermediate representation (IR). We address in this work three key language features that make a provably correct transformation challenging.

Operand Stack. The Java virtual machine (JVM) is stack-based 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 [8], 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. The current transformation keeps the number of extra temporary variables at a reasonable level without using auxiliary iterated analyses such as copy propagation.

Splitted Object Creation. The object creation scheme of the JVM is another feature which is difficult to track because it is done in two distinct steps: (i) raw object allocation and (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 verifies this by tracking aliases of uninitialized objects in the operand stack, but this valuable alias information is lost for subsequent static analyses. The present transformation rebuilds the initialization chain of an object with the instruction `x := new C(arg1, arg2, ...)`. This specific feature puts new constraints on the formalization because object allocation order is no longer preserved.

Exception Throwing Order. A last 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 raise different kinds of exception and any sound transformation must take care to preserve the exception throwing order.

```
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)

Fig. 1. Example of source code, bytecode and two possible transformations

Illustrating Example. Figure 1 presents an example program illustrating these issues. For more readability, we will also refer to Figure 1(a) that gives the corresponding Java

source code. Its corresponding bytecode version (Figure 1(c)) shows the JVM object initialization scheme: an expression `new A()` is compiled to 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 on an *initialized* object. This initialization by side-effect is particularly challenging for the BCV [6] which has to keep track of the alias between uninitialized 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(b) shows a first attempt of such a fusion. However, in this example, side-effect free expressions are generated in a naive way which *changes the semantics* in several ways. First, the program does not respect the *allocation order*. This 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(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(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 the program in Figure 1(b) is that it does not respect the *exception throwing order* of the bytecode version. In Figure 1(b) the call to `A()` may appear before the `DivByZero` exception may be raised when evaluating `x/y`. The program in Figure 1(d) solves this problem using a specific instruction `notzero y` that explicitly checks if `y` is non-zero and raises a `DivByZero` exception if this is not the case.

The algorithm presented in Section 3 and proved correct in Section 4 takes care of these pitfalls. The input (BC) and IR (BIR) languages are presented in Section 2. The transformation demands that input programs pass the BCV and use uninitialized objects in a slightly restricted way (see Section 3). Our algorithm uses the technique of symbolic execution of the input code, which allows dealing simultaneously with the aforesaid challenges, while the main alternative techniques, briefly overviewed in Section 5, proceed in at least two distinct phases on the code: naive code is first generated, it is then optimized in a second phase, using traditional compiler optimization techniques. We believe the symbolic execution scheme gives rise to a rather elegant correctness proof, compared to the one we would obtain by combining correctness proofs of separate phases. This transformation has been implemented for the full Java bytecode language (meeting the same requirements), as part of the SAWJA[3] static analysis framework. Its experimental evaluation [5] of this transformation shows it competes well with other state-of-the-art bytecode transformation tools.

2 Source and Target Languages

Our source language BC is an untyped stack-based Java-like bytecode language with object construction, exceptions and virtual calls. In the formalization part of this work, the main missing feature is multi-threading. Other missing features, e.g. 64 bits values,

[3] `http://sawja.inria.fr/`

static elements (static fields and static methods) or method overloading would make the current formalization heavier but do not introduce any new difficulties. The set of bytecodes we consider is given in Figure 2. They are familiar Java bytecodes and will not be explained. In order for the transformation to succeed, additional structural constraints on the bytecode must be satisfied. They are described in the dedicated paragraph *Relative BCV-Completeness* (Section 3).

The BIR target language (Figure 2) provides expressions and instructions for variable and field assignments. BIR distinguishes two kinds of variables: local variables in var_{BC} are identifiers already used at the BC level, while *tvar* is a set of fresh identifiers introduced in BIR. Like BC, BIR is unstructured. What BIR brings here is that conditional jumps now depend on structured expressions.

Object Creation and Initialization. The Java bytecode object creation scheme, as explained in Section 1, forces static analyses to deal with alias information between uninitialized references. But this precise work is already done by the BCV when checking for object initialization. Folding constructor calls into `x := new C`$(e_1, \ldots e_n)$ in BIR avoids this redundant task in later static analyses.

Another ambiguous feature of bytecode is that `constructor C` corresponds to either a constructor or a super-constructor call according to the initialization status of the receiver object. This kind of information is rather costly for static analyses if they need to distinguish both situations. BIR removes this ambiguity by providing a distinct super constructor call instruction (e.`super`(`C`$'$, e_1,...,e_n), where `C`$'$ is the super class of `C`).

Explicit Checks and Class Initialization. The side-effect free expressions requirement sometimes forces the transformation to revert the expression evaluation order, and thus of the exception throwing order. The solution provided by BIR is to use assertions: instructions `notzero e` and `notnull e` respectively check if the expression `e` evaluates to zero or null and raise an exception if the check fails.[4] By the same token, we obtain that the BIR expression evaluation is error-free. As illustrated by the example in the Introduction (Figure 1), folded constructors and side-effect free expressions cause the object allocation order to be modified. Still, preserving the class initialization order must be taken care of, as static class initializers `C`.⟨`clinit`⟩ impact the program semantics. The BIR extra instruction `mayinit C` solves this problem by calling `C`.⟨`clinit`⟩ whenever it is required.

2.1 Semantic Domains of BC and BIR

Our goal is to express in the correctness of the BC2BIR transformation not only the input/output preservation. We want to be as precise as possible, i.e. all what is preserved by BC2BIR should be clearly stated in the theorem. BC and BIR semantics are designed to this end. Semantic domains are given in Figure 3.

[4] In our formalization, heaps are infinite. Dealing with finite heaps would require preserving `OutOfMemory` exceptions. BIR would need to be extended with an instruction `checkheap C`, generated when transforming the BC instruction `new C` and checking if the heap available space is sufficient to allocate a `C` object.

$$
\begin{array}{rll}
\mathbb{C} & ::= & \textit{class names}: \\
 & \texttt{C} \mid \ldots & \\
\mathbb{F} & ::= & \textit{field names}: \\
 & \texttt{f} \mid \ldots & \\
\mathbb{M} & ::= & \textit{method names}: \\
 & \texttt{m} \mid \ldots & \\
\textit{var}_{\mathrm{BC}} & ::= & \text{BC } \textit{variables}: \\
 & \texttt{x} \mid \texttt{x}_1 \mid \texttt{x}_2 \mid \ldots \texttt{this} & \\
\textit{instr}_{\mathrm{BC}} & ::= & \text{BC } \textit{instructions}: \\
 & \texttt{nop} \mid \texttt{push c} \mid \texttt{pop} \mid \texttt{dup} \mid \texttt{add} \mid \texttt{div} & \\
 & \mid \texttt{load } \textit{var}_{\mathrm{BC}} \mid \texttt{store } \textit{var}_{\mathrm{BC}} & \\
 & \mid \texttt{new C} \mid \texttt{constructor C} & \\
 & \mid \texttt{getfield f} \mid \texttt{putfield f} & \\
 & \mid \texttt{invokevirtual C.m} & \\
 & \mid \texttt{if pc} \mid \texttt{goto pc} & \\
 & \mid \texttt{vreturn} \mid \texttt{return} &
\end{array}
$$

$$
\begin{array}{rll}
\textit{tvar} & ::= & \textit{temporary variables:} \\
 & \texttt{t} \mid \texttt{t}_1 \mid \texttt{t}_2 \mid \ldots & \\
\textit{var}_{\mathrm{BIR}} & ::= & \text{BIR } \textit{variables}: \\
 & \textit{var}_{\mathrm{BC}} \mid \textit{tvar} & \\
\textit{expr} & ::= & \textit{side-effect free expressions:} \\
 & \texttt{c} \mid \texttt{null} \mid \textit{var}_{\mathrm{BIR}} & \\
 & \mid \textit{expr}\texttt{+}\textit{expr} \mid \textit{expr}\texttt{/}\textit{expr} \mid \textit{expr}\texttt{.f} & \\
\textit{instr}_{\mathrm{BIR}} & ::= & \text{BIR } \textit{instructions}: \\
 & \texttt{nop} \mid \texttt{mayinit C} & \\
 & \mid \texttt{notnull } \textit{expr} \mid \texttt{notzero } \textit{expr} & \\
 & \mid \textit{var}_{\mathrm{BIR}}\texttt{:=}\textit{expr} \mid \textit{expr}\texttt{.f :=}\textit{expr} & \\
 & \mid \textit{var}_{\mathrm{BIR}}\texttt{:= new C(}\textit{expr}, \ldots, \textit{expr}\texttt{)} & \\
 & \mid \textit{expr}\texttt{.super (C,}\textit{expr}, \ldots, \textit{expr}\texttt{)} & \\
 & \mid \textit{var}_{\mathrm{BIR}}\texttt{:=}\textit{expr}\texttt{.m(C,}\textit{expr}, \ldots, \textit{expr}\texttt{)} & \\
 & \mid \textit{expr}\texttt{.m(C,}\textit{expr}, \ldots, \textit{expr}\texttt{)} & \\
 & \mid \texttt{if } \textit{expr} \texttt{ pc} \mid \texttt{goto pc} & \\
 & \mid \texttt{vreturn } \textit{expr} \mid \texttt{return} &
\end{array}
$$

Fig. 2. Instructions of BC and BIR

$$
\begin{array}{rl}
\textit{Value} = & \mid (\textit{Num } n),\ n \in \mathbb{Z} \\
 & \mid (\textit{Ref } r),\ r \in \textit{Ref} \\
 & \mid \textit{Null} \\
\overline{\textit{Value}} = & \textit{Value} \cup \{\textit{Void}\}
\end{array}
\qquad
\begin{array}{rl}
\textit{InitTag} = & \widetilde{\mathbb{C}}_{\mathbb{N}} \cup \mathbb{C} \\
\textit{Object} = & (\mathbb{F} \to \textit{Value})_{\textit{InitTag}} \\
\textit{Heap} = & \textit{Ref} \hookrightarrow \textit{Object} \\
\textit{Error} = & \{\, \Omega^{NP}, \Omega^{DZ} \}
\end{array}
$$

$$
\begin{array}{rl}
\textit{Stack} = & \textit{Value}^* \\
\textit{State}_{\mathrm{BC}} = & (\textit{Heap} \times \mathbb{M} \times \mathbb{N} \times \textit{Env}_{\mathrm{BC}} \times \textit{Stack}) \\
 & \cup \left(\textit{Heap} \times \overline{\textit{Value}}\right) \\
 & \cup (\textit{Error} \times \mathbb{M} \times \mathbb{N} \times \textit{Env}_{\mathrm{BC}} \times \textit{Heap})
\end{array}
\qquad
\begin{array}{rl}
\textit{Env}_{\mathrm{BC}} = & \textit{var}_{\mathrm{BC}} \hookrightarrow \textit{Value} \\
\textit{State}_{\mathrm{BIR}} = & (\textit{Heap} \times \mathbb{M} \times (\mathbb{N} \times \textit{instr}^*_{\mathrm{BIR}}) \times \textit{Env}_{\mathrm{BIR}}) \\
 & \cup \left(\textit{Heap} \times \overline{\textit{Value}}\right) \\
 & \cup (\textit{Error} \times \mathbb{M} \times \mathbb{N} \times \textit{Env}_{\mathrm{BIR}} \times \textit{Heap})
\end{array}
$$

Fig. 3. BC and BIR semantic domains

One of the subtleties of BC2BIR is that, although the object allocation order is modified, it takes care of preserving a strong relation between objects allocated in the heap, as soon as their initialization has begun. Thus, we attach to objects, seen as functions from fields $\mathbb{F}$ to *Value*, an initialization tag $\in$ *InitTag*. This was first introduced by Freund and Mitchell in [6], but we adapt it to our purpose. Following the Java convention, an object allocated at point `pc` by `new C` is uninitialized (tagged $\widetilde{\mathtt{C}}_{\mathtt{pc}}$) as long as no constructor has been called on it; an object is tagged `C` either if its initialization is ongoing (all along the constructor call chain) or completed when the `Object` constructor is called. Note that, unlike [6], *InitTag* does not track intermediate initialization status, but this can be recovered from the observational trace semantics (Section 2.2).

A *normal execution state* consists of a heap, the current method, the next instruction to execute, and the local memory of the method (local variables and operand stack for BC, only local variables, but with more variable names for BIR). We do not model the usual call stack in execution states, but rely on a so-called mostly-small-step semantics

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

$$\frac{\begin{array}{c} instrAt_P(m,pc) = \texttt{constructor C} \\ V = v_1 :: \cdots :: v_n \quad h(r) = o_{\widetilde{\texttt{C}}_{\texttt{j}}} \quad h' = h[r \mapsto o_{\texttt{C}}] \\ \langle h', InitLState(\texttt{C.init}, (Ref\ r) :: V)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h'', Void\rangle \end{array}}{\langle h,m,pc,l,V{::}(Ref\ r){::}s\rangle \xrightarrow{[r\leftarrow \texttt{C.init}(V)].\Lambda_h}_{n+1} \langle h'',m,pc{+}1,l,s\rangle}$$

Fig. 4. BC semantic rules for object allocation and initialization (excerpt)

(see Section 2.2). In the correctness theorem (Section 4), one BC step is matched by a sequence of BIR steps. The way we define BIR program points avoids awkwardness in this matching by tracking BIR instructions with a pair $(\texttt{pc}, \ell) \in \mathbb{N} \times instr^*$. A *return state* is made of a heap and a returned value.

We also want the semantic preservation to deal with execution errors. We do not model exception catching in this work but it will not bring much difficulty thanks to the way we define *error states*. These include the method program point of the faulty instruction and the current context (heap and environment), and also keep track of the kind of error: division by zero (Ω^{DZ}) and null pointer dereferencing (Ω^{NP}). BC programs passing the BCV only get stuck in an error or return state of the main method.

2.2 Observational Semantics of BC and BIR

We achieve a fine-grained preservation criterion by using a mostly small-step operational semantics. Indeed, a correctness criterion only stating the preservation of returned values would not bring much information to static analyses dealing with intermediate program points. We push further this approach by labelling transitions with observable events, keeping track of all the program behavior aspects that are preserved by the transformation (even local variable assignments). Observable events are defined as *Evt*, the union of the following sets ($v, v_1, \ldots, v_n \in Value$, $r \in Ref$):

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

Actions irrelevant to the correctness of the transformation are silent transitions labelled with τ. These include expression evaluation steps, as expressions are side-effect and error free. Note that, due to the modification of the object allocation order , the memory effect of the BC instruction `new C` is kept silent. This is harmless thanks to the strong restrictions imposed by the BCV on the use of uninitialized references [6].

$$\frac{\begin{array}{c} hd(\ell) = x{:=}\texttt{new C}\,(e_1,\ldots,e_n) \\ h,l \vDash e_i \Downarrow v_i \quad (Ref\ r) = newObject(C,h) \quad h' = h[r \mapsto (\lambda f.init(f))_{\mathtt{C}}] \\ V = v_1 :: \cdots :: v_n \quad \langle h', InitLState(\texttt{C.init}, (Ref\ r) :: V)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h'', Void\rangle \end{array}}{\langle h, m, (pc,\ell), l\rangle \xrightarrow{[r \leftarrow C.init(V)].\Lambda_h.[x \leftarrow (Ref\ r)]}_{n+1} \langle h'', (m, next(pc,\ell), l[x \mapsto (Ref\ r)]\rangle}$$

Fig. 5. BIR semantic rule for object allocation and initialization (excerpt)

Program execution generates traces of events, which permit expressing sequences of events. We illustrate how event traces are managed intra and inter-procedurally with object allocation and initialization (Figures 4 for BC rules and Figure 5 for BIR rules).

In rule for `new C`, $newObject(C,h)$ returns the reference $(Ref\ r)$ freshly allocated in h. All object fields are set to their default values (zero for integers and *Null* for references) by the function *init* and the object tag is set to $\widetilde{\mathtt{C}}_{\mathtt{pc}}$, leading to the new heap h'. No "object allocation" event is observed. However, the class initialization order will be preserved[5]: observing $mayinit(\mathtt{C})$ in the BC and BIR execution traces (when respectively executing `new C` and `mayinit C`) helps us proving this property. When executing `constructor C` on an object tagged with $\widetilde{\mathtt{C}}_{\mathtt{j}}$ (pointed to by $(Ref\ r)$), the method `C.init`[6] is entirely executed (in a mostly-small step style) starting from a heap h' where the object tag has been updated to `C`. The starting local memory $InitLState(\texttt{C.init}, args)$ consists of an empty stack, and local registers (`this` for $(Ref\ r)$ and others registers for arguments). The execution trace of `C.init` restricted to events in $EvtH$, denoted by Λ_h is then exported to the caller (as it contains events related to the heap, which is shared by methods) and appended to the event $r \leftarrow \texttt{C.init}(V)$. We write $\dot{\Rightarrow}$ for the transitive closure of the small-step relation $\dot{\rightarrow}$.

In Section 4, we will rely on an inductive reasoning to prove the semantics preservation of the transformation. Therefore, we index transitions with a natural number counting the maximal execution call depth: it is zero whenever no method is called, and incremented each time a method is called.

3 Transformation Algorithm

In this section we describe the BC2BIR algorithm (given in Figure 7) for converting BC code into BIR code. A central feature of our algorithm is the use of a symbolic stack to decompile stack-oriented code into three-address code. In the following we explain how the symbolic stack is used in decompiling BC instructions and how it is managed at control flow join points. Another distinguishing feature of the algorithm is the merging of instructions for object allocation and initialization into one compound BIR instruction which is also performed quite elegantly thanks to the symbolic stack.

The core of the algorithm is the function BC2BIR_{instr} that maps a BC instruction into a list of BIR instructions and at the same time symbolically executes BC code

[5] In order to lighten the formalization, `mayinit C` behaves in the present work as `nop` but raises a specific $mayinit(\mathtt{C})$ event.

[6] `C.init` is the JVM conventional name for the `C` constructors.

using an abstract stack of symbolic expressions:

$$\text{BC2BIR}_{instr} : \mathbb{N} \times instr_{\text{BC}} \times AbstrStack \rightarrow \left(instr^*_{\text{BIR}} \times AbstrStack\right) \cup Fail$$
$$AbstrStack = SymbExpr^* \qquad SymbExpr = expr \cup \{UR^{\mathtt{C}}_{\mathtt{pc}} \mid C \in \mathbb{C}, \mathtt{pc} \in \mathbb{N}\}$$

Expressions in *expr* are BC decompiled expressions and $UR^{\mathtt{C}}_{\mathtt{pc}}$ is a placeholder for a reference to an uninitialized object, allocated at point `pc` by the intruction `new C`. BC2BIR_{instr} is given in Figure 6, where $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ denote fresh temporary variables introduced at point `pc`. A paragraph at the end of this section describes the failure cases.

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*

Inputs		Outputs	
Instr	Stack	Instrs	Stack
`if pc'`	e::*as*	[`if e pc'`]	*as*
`goto pc'`	*as*	[`goto pc'`]	*as*
`return`	*as*	[`return`]	*as*
`vreturn`	e::*as*	[`return e`]	*as*

Inputs		Outputs	
Instr	Stack	Instrs	Stack
`add`	$\mathtt{e}_1$::$\mathtt{e}_2$::*as*	[`nop`]	$\mathtt{e}_1 + \mathtt{e}_2$::*as*
`div`	$\mathtt{e}_1$::$\mathtt{e}_2$::*as*	[`notzero` $\mathtt{e}_2$]	$\mathtt{e}_1/\mathtt{e}_2$::*as*
`new C`	*as*	[`mayinit C`]	$UR^{\mathtt{C}}_{\mathtt{pc}}$::*as*
`getfield f`	e::*as*	[`notnull e`]	e.f::*as*

Inputs		Outputs		Cond
Instr	Stack	Instrs	Stack	
`store x`	e::*as*	[`x := e`]	*as*	$\mathtt{x} \notin as^a$
		[$\mathtt{t}^0_{\mathtt{pc}}$:= `x`; `x := e`]	$as[\mathtt{t}^0_{\mathtt{pc}}/\mathtt{x}]$	$\mathtt{x} \in as^a$
`putfield f`	e′::e::*as*	[`notnull e`; *Fsave*(`pc`, `f`, *as*); `e.f := e'`]	$as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/\mathtt{e}_{\mathtt{i}}]$	^{ab}
`invokevirtual C.m`	$\mathtt{e}'_1 \ldots \mathtt{e}'_n$::e::*as*	[`notnull e`; *Hsave*(`pc`, `as`); $\mathtt{t}^0_{\mathtt{pc}}$:= `e.m`($\mathtt{e}'_1 \ldots \mathtt{e}'_n$)]	$\mathtt{t}^0_{\mathtt{pc}}$::$as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}_{\mathtt{j}}]$	value return ^{ac}
		[`notnull e`; *Hsave*(`pc`, `as`); `e.m`($\mathtt{e}'_1 \ldots \mathtt{e}'_n$)]	$as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}_{\mathtt{j}}]$	*Void* return^{ac}
`constructor C`	$\mathtt{e}'_1 \ldots \mathtt{e}'_n$::$\mathtt{e}_0$::*as*	[*Hsave*(`pc`, `as`); $\mathtt{t}^0_{\mathtt{pc}}$:= `new C`($\mathtt{e}'_1 \ldots \mathtt{e}'_n$)]	$as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}_{\mathtt{j}}]$	$\mathtt{e}_0 = UR^{C}_{\mathtt{pc}'}$ ^{c}
		[`notnull e`; *Hsave*(`pc`, `as`); `e.super`(`C`, $\mathtt{e}'_1 \ldots \mathtt{e}'_n$)]	$as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}_{\mathtt{j}}]$	otherwise $^{a\,c}$

Fig. 6. BC2BIR_{instr} – Transformation of a BC instruction at `pc`

[a] where for all `C` and `pc'`, $\mathtt{e} \neq UR^{\mathtt{C}}_{\mathtt{pc}'}$
[b] where $\mathtt{e}_{\mathtt{i}}, \mathtt{i} = 1 \ldots n$ are all the elements of *as* such that $\mathtt{f} \in \mathtt{e}_{\mathtt{i}}$
[c] where $\mathtt{e}_{\mathtt{j}}, \mathtt{j} = 1 \ldots m$ are all the elements of *as* that read a field

We now explain the main cases of BC2BIR_{instr}. For instruction `load x` , the symbolic expression `x` is pushed on the abstract stack *as* and the BIR instruction `nop` is generated. We generate `nop` to make the step-matching easier in the proof of the theorem. Transformations of return and jump instructions are straightforward. Before going into more technicality, we give a simple example of symbolic execution. Successively symbolically executing `load x` and `load y` will lead to the abstract stack `y`::`x`::ε. If `add` were the next instruction to transform, the abstract stack would become (`x` + `y`)::ε.

Transforming instructions `store` , `putfield` and `invokevirtual` follows the same principle. However, for semantics preservation issues, we must take care of their memory effect. Their execution might modify the value of local variables or object fields appearing in the expressions of the abstract stack, whose value would be erroneously modified by side effect. We tackle this subtlety by storing in temporary variables (of the form $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$) each stack element whose value might be modified. In the case of `store x`, it is enough only remembering the old value of `x`. In the case of `putfield f`, all expressions in *as* accessing an `f` field are remembered: *Fsave*(`pc`, `f`, $\mathtt{e}_1$::$\mathtt{e}_2$::. . .::$\mathtt{e}_n$) generates an assignment $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := e_i$ for all e_i that reads at least once the field `f`. Finally, in the case of `invokevirtual`, we store the value of each expression accessing the heap,

```
1  function BC2BIR(P, m) =
2  AS_in[m, 0] := nil
3  for (pc = 0, pc ≤ length(m), pc++) do
4   // Compute the entry abstract stack
5   if (pc ∈ jmpTgt^P_m) then
6     if (not C_UR(pc)) then fail end
7     AS_in[m, pc] := newStackJmp(pc, AS_in[m, pc])
8   end
9
10  // Decompile instruction
11  (AS_out[m, pc], code) := BC2BIR_instr(pc, instrAt_P(m, pc), AS_in[m, pc])
12  IR[m, pc] := TAssign(succ(pc) ∩ jmpTgt^P_m, AS_out[m, pc])++code
13
14  // Fail on a non-empty stack backward jump
15  if (AS_out[m, pc] ≠ nil ∧ ∃pc' ∈ succ(pc).pc > pc') then fail end
16
17  // Pass on the output abstract stack
18  if (pc + 1 ∈ succ(pc) ∧ pc + 1 ∉ jmpTgt^P_m) then AS_in[m, pc + 1] := AS_out[m, pc]  end
19 end
```

Fig. 7. BC2BIR – BC method transformation. *length*(m) is the size of the code of method *m*, *succ*(pc) the set of successors of pc in m, *stackSize*(pc) the stack size at point pc and $jmpTgt^{\mathtt{P}}_{\mathtt{m}}$ the set of jump targets in m.

which could be modified by the callee execution: *Hsave*(pc, $\mathtt{e}_1$::$\mathtt{e}_2$::. . .::$\mathtt{e}_n$) generates an assignment $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := e_i$ for all e_i that reads a field.

Object creation and initialization require special attention as this is done by separate (and possibly distant) instructions. Symbolically executing `new C` at point pc pushes $UR^{\mathtt{C}}_{\mathtt{pc}}$ (representing the freshly allocated reference) on the stack and generates `mayinit C` for class initialization whenever it is required. Instruction `constructor C` will be transformed differently whether it corresponds to a constructor or a super constructor call. Both cases are distinguished thanks to the symbolic expression on which it is called. We generate a BIR folded constructor call at point pc if the symbolic expression is $UR^{\mathtt{C}}_{\mathtt{pc}'}$ (and a super constructor call otherwise). $UR^{\mathtt{C}}_{\mathtt{pc}}$ are used to keep track of alias information between uninitialized references, when substituting them for the local variable receiving the new object. This mechanism is similar to what is used by the BCV to check for object initialization.

Transforming the whole code of a BC method is done by BC2BIR which (i) computes the entry abstract stack used by BC2BIR$_{instr}$ to transform the instruction, (ii) performs the BIR generation and (iii) passes on the output abstract stack to the successor points. BC2BIR is given in Figure 7. It computes three arrays: IR[m] is the BIR version of the method m, $\mathtt{AS}_{\mathtt{in}}$[m] and $\mathtt{AS}_{\mathtt{out}}$[m] respectively contain the input and output symbolic stacks used by BC2BIR$_{instr}$.

Most of the time, the control flow is linear (from pc to only pc + 1). In this case, we only perform the BC2BIR$_{instr}$ generation (Lines 11 and 12) and the abstract stack resulting from BC2BIR$_{instr}$ is transmitted as it is (Line 18). The case of control flow joins must be handled more carefully. In a program passing the BCV, we know that at every join point, the size of the stack is the same regardless of the predecessor point. Still, the content of the abstract stack might change (when e.g. the two branches of a conditional compute two different expressions). But stack elements are expressions used in the generated instructions and hence must not depend on the control flow path. We illustrate this point with the example of Figure 8. This function returns 1 or -1,

```
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^1_5]
```
(c) Symbolic stack

```
int f(x);
  0 : nop;
  1 : if x 4;
  2 : nop;
  3 : T^1_5 := -1;  goto 5;
  4 : nop;  T^1_5 := 1;
  5 : vreturn T^1_5;
```
(d) BIR function

Fig. 8. Example of bytecode transformation – jumps on non-empty stacks

depending on whether the argument `x` is zero or not. We focus on program point 5, whose predecessors are points 3 and 4. The abstract stack after executing the instruction `goto 5` is -1 (point 3 in Figure 8(c)), while it becomes 1 after program point 4. At point 5, depending on the control flow path, the abstract stack is thus not unique.

The idea is here to store, before reaching a join point, every stack element in a temporary variable and to use, at the join point, a *normalized* stack made of all these variables. A naming convention ensures that (i) identifiers are independent of the control flow and (ii) each variable denote the same stack element: we use the identifier $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ to store the ith element of the stack for a join point at `pc`. All $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ are initialized when transforming a BC instruction preceeding a join point. In Figure 8(d), at points 3 and 4, we respectively store -1 and 1 in $\mathtt{T}^{\mathtt{1}}_{\mathtt{5}}$, the top element of the entry stack at point 5.

In the algorithm, this is done at Line 12: we prepend to the code generated by BC2BIR$_{instr}$ the assignments of all abstract stack elements to the $\mathtt{T}^{\mathtt{i}}_{\mathtt{jp}}$, for all join points `jp` successor of `pc`. These assignments are generated by $\mathtt{T}Assign(\mathtt{S}, as)$, where S is a set of program points. The restriction Line 15 ensures these assignments are conflict-free by making the transformation fail on non-empty stack backjumps. The function $newStackJmp(\mathtt{jp}, as)$ (Line 7) computes the normalized stack at join point `jp`. It returns a stack of $\mathtt{T}^{\mathtt{i}}_{\mathtt{jp}}$ except that $UR^{\mathtt{C}}_{\mathtt{pc}}$ are preserved. We need here the following constraint $C_{UR}(\mathtt{jp})$ on $\mathtt{AS}_{\mathtt{out}}$, that we check before computing the entry abstract stack (Line 6): $\forall \mathtt{i}.\ (\exists \mathtt{pc}' \in pred_m(\mathtt{jp}).\ \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = UR^{\mathtt{C}}_{\mathtt{pc}_{\mathtt{0}}}) \Rightarrow (\forall \mathtt{pc}' \in pred_m(\mathtt{jp}).\ \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = UR^{\mathtt{C}}_{\mathtt{pc}_{\mathtt{0}}})$. It means that before a join point `jp`, if the stack contains any $UR^{\mathtt{C}}_{\mathtt{pc}}$ at position i, then it is the case for all predecessors of $\mathtt{jp} \in jmpTgt^{\mathtt{P}}_{\mathtt{m}}$.

Relative BCV-Completeness. Every case undescribed in Figures 6 and 7 yields *Fail*. Most of them are ruled out by the BCV (e.g. stack height mismatch, or uninitialised reference field assignment) but few cases remain. First, this version of the algorithm fails on non-empty stack backjumps, but they are addressed in [5]. Finally, the present transformation puts restrictions on the manipulation of uninitialised locations in the operand stack and the local variables. Transforming `store x` requires that the top expression `e` is not $UR^{\mathtt{C}}_{\mathtt{pc}}$ because no valid BIR instruction would match, as constructors are folded. For the same reason, we fail to transform bytecode that does not satisfy C_{UR}: this constraint allows us not to store $UR^{\mathtt{C}}_{\mathtt{pc}}$ stack elements. Unfortunately these patterns are not ruled

out by the JVM specification and we may reject programs that pass the BCV. However this is not a limitation in practice because such patterns are not used by standard compilers. Our transformation tool has been tested on the 609209 methods of the Eclipse distribution without encountering such cases [5].

4 Correctness

The BC2BIR algorithm satisfies a precise semantics preservation property that we formalize in this section: the BIR program BC2BIR(P) simulates the initial BC program P and both have similar execution traces. This similarity cannot be a simple equality, because some variables have been introduced by the transformation and the object allocation order is modified by BC2BIR— both heaps do not keep equal along both program executions. We define in Section 4.1 what semantic relations make us able to precisely relate BC and BIR executions. Section 4.2 formally states the semantic preservation of BC2BIR. For space reason, we only provide a proof sketch. The complete proof is given in the accompanying report [5]. We lighten the notations from now and until the end of this section by writing a BC program P, its BIR version $P' = \text{BC2BIR}(P)$.

4.1 Semantic Relations

Heap Isomorphism. The transformation does not preserve the object allocation order. However, the two heaps stay isomorphic: there exists a partial bijection between them. For example, in P (Figure 1(c)), the `B` object is allocated before the `A` object is passed as an argument to the `B` constructor. In P' (Figure 1(d)), constructors are folded and object creation is not an expression, the `A` object must thus be created (and initialized) before passing $\mathtt{t}_1$ (containing its reference) as an argument to the `B` constructor.

Heaps are not equal along the execution of the two programs: after program point `5` in P, the heap contains two objects that are not yet in the heap of P'. However, after program points `7`, each use in P' of the `A` object is synchronized with a use in P of the reference pointing to the `A` object (both objects are initialized, so both references can be used). The same reasoning can be applied just after points `8` about the `B` objects. A bijection thus exists between references of both heaps. It relates references to allocated objects as soon as their initialization has begun. Along the executions of BC and BIR programs, it is extended accordingly on each constructor call starting the initialization of a new object. In Figure 1, given an initial partial bijection on the heaps domains, it is first extended at points `7` and then again at points `8`.

Semantic Relations. This heap isomorphism has to be taken into account when relating semantic domains and program executions. Thus, the semantic relations over values, heaps, environments, configurations and observable events (see Table 1) are parametrized by a bijection β defined on the heap domains.

When relating values, the interesting case is for references. Only references related by β are in the relation. The semantic relation on heaps is as follows. First, objects related by β are exactly those existing in both heaps and on which a constructor has been called. Secondly, the related objects must have the same initialization status (hence the

Relation	Definition
$v_1 \overset{\mathrm{v}}{\sim}_\beta v_2$ $v_1, v_2 \in \mathit{Value}$	$\dfrac{}{\mathit{Null} \overset{\mathrm{v}}{\sim}_\beta \mathit{Null}}$ $\dfrac{n \in \mathbb{Z}}{(\mathit{Num}\ n) \overset{\mathrm{v}}{\sim}_\beta (\mathit{Num}\ n)}$ $\dfrac{\beta(r_1) = r_2}{(\mathit{Ref}\ r_1) \overset{\mathrm{v}}{\sim}_\beta (\mathit{Ref}\ r_2)}$
$h_1 \overset{\mathrm{H}}{\sim}_\beta h_2$ $h_1, h_2 \in \mathit{Heap}$	– $\mathit{dom}(\beta) = \{r \in \mathit{dom}(h_1) \mid \forall \mathtt{C}, \mathtt{pc},\ \mathit{tag}_{h_1}(r) \neq \widetilde{\mathtt{C}_{\mathtt{pc}}}\}$ – $\mathit{rng}(\beta) = \mathit{dom}(h_2)$ – $\forall r \in \mathit{dom}(h_1)$, let $o_t = h_1(r)$ and $o'_{t'} = h_2(\beta(r))$ then $(i)\ t = t' \quad (ii)\ \forall f,\ o_t(f) \overset{\mathrm{v}}{\sim}_\beta o'_t(f)$
$l_1 \overset{\mathrm{E}}{\sim}_\beta l_2$ $(l_1, l_2) \in \mathit{Env}_{\mathrm{BC}} \times \mathit{Env}_{\mathrm{BIR}}$	$\mathit{dom}(l_1) = \mathit{var}_{\mathrm{BC}} \cap \mathit{dom}(l_2)$ and $\forall x \in \mathit{dom}(l_1),\ l_1(x) \overset{\mathrm{v}}{\sim}_\beta l_2(x)$
$c_1 \overset{\mathrm{c}}{\sim}_\beta c_2$ $(c_1, c_2) \in \mathit{State}_{\mathrm{BC}} \times \mathit{State}_{\mathrm{BIR}}$	$\dfrac{h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt}{\langle h, m, \mathtt{pc}, l, s\rangle \overset{\mathrm{c}}{\sim}_\beta \langle ht, m, (\mathtt{pc}, \mathit{instrAt}_{P'}(m, pc)), lt\rangle}$ $\dfrac{h \overset{\mathrm{H}}{\sim}_\beta ht \quad rv \overset{\mathrm{v}}{\sim}_\beta rv'}{\langle h, rv\rangle \overset{\mathrm{c}}{\sim}_\beta \langle ht, rv'\rangle}$ $\dfrac{h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt}{\langle \Omega^k, m, pc, h, l\rangle \overset{\mathrm{c}}{\sim}_\beta \langle \Omega^k, m, pc, ht, lt\rangle}$
$\lambda_1 \overset{!}{\sim}_\beta \lambda_2$ with $\lambda_1, \lambda_2 \in \mathit{Evt}$	$\dfrac{}{\tau \overset{!}{\sim}_\beta \tau}$ $\dfrac{}{\mathit{mayinit}(\mathtt{C}) \overset{!}{\sim}_\beta \mathit{mayinit}(\mathtt{C})}$ $\dfrac{\beta(r_1) = r_2 \quad v_1 \overset{\mathrm{v}}{\sim}_\beta v_2}{r_1.f \leftarrow v_1 \overset{!}{\sim}_\beta r_2.f \leftarrow v_2}$ $\dfrac{x \in \mathit{var}_{\mathrm{BC}} \quad v_1 \overset{\mathrm{v}}{\sim}_\beta v_2}{x \leftarrow v_1 \overset{!}{\sim}_\beta x \leftarrow v_2}$ $\dfrac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\mathrm{v}}{\sim}_\beta v'_i}{r_1 \leftarrow C.\mathit{init}(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2 \leftarrow C.\mathit{init}(v'_1, \ldots, v'_n)}$ $\dfrac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\mathrm{v}}{\sim}_\beta v'_i}{r_1.C.\mathit{init}(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2.C.\mathit{init}(v'_1, \ldots, v'_n)}$

Table 1. Semantic relations

same class) and their fields must have related values. Here we write $\mathit{tag}_h(r)$ for the tag t such that $h(r) = o_t$. A BIR environment is related to a BC environment if and only if both local variables have related values. Temporary variables are, as expected, not taken into account. Execution states are related through their heaps and environments, the stack is not considered here. Program points are not related to a simple one-to-one relation: the whole block generated from a given BC instruction must be executed before falling back into the relation. Hence, a BC state is matched at the beginning of the BIR block of the same program point: the function $\mathit{instrAt}_{P'}(m, pc)$ gives the BIR program point $(\mathtt{pc}, \ell)$ with ℓ the complete instruction list at $\mathtt{pc}$. We only relate error states of the same kind of error. Finally, two observable events are related if they are of the same kind, and the values they involve are related. To relate execution traces, we pointwise extend $\overset{!}{\sim}_\beta$. We now assume that $\mathtt{IR}, \mathtt{AS}_{\mathtt{in}}$ and $\mathtt{AS}_{\mathtt{out}}$ are the code and abstract stack arrays computed by BC2BIR, and so until the end of the section.

4.2 Soundness Result

The previously defined observational semantics and semantic relations allows achieving a very fine-grained correctness criterion for the transformation BC2BIR. It says that P' simulates the initial program P: starting from two related initial configurations, if the execution of P terminates in a given (normal or error) state, then P' terminates in a

related state, and both execution traces are related, when forgetting temporary variables assignments in the BIR trace (we write Λ_{proj} for such a projection of Λ). More formally:

Theorem 1 (Semantic preservation)
Let $m \in \mathbb{M}$ be a method of P (and P') and $n \in \mathbb{N}$. Let $c = \langle h, m, 0, l, \varepsilon \rangle \in State_{\mathrm{BC}}$ and $ct = \langle h, m, (0, instrAt_{P'}(m, 0)), l \rangle \in State_{\mathrm{BIR}}$. Then two properties hold:

Normal return *If $c \overset{\Lambda}{\Rightarrow}_n \langle h', v \rangle$ then there exist unique ht', v', Λ' and β such that $ct \overset{\Lambda'}{\Rightarrow}_n \langle ht', v' \rangle$ with $\langle h', v \rangle \overset{\mathrm{c}}{\sim}_\beta \langle ht', v' \rangle$ and $\Lambda \overset{!}{\sim}_\beta \Lambda'_{proj}$.*

Error *If $c \overset{\Lambda}{\Rightarrow}_n \langle \Omega^k, m, pc', l', h' \rangle$ then there exist unique ht', lt', Λ' and β s.t $ct \overset{\Lambda'}{\Rightarrow}_n \langle \Omega^k, m, pc', lt', ht' \rangle$ with $\langle \Omega^k, m, pc', l', h' \rangle \overset{\mathrm{c}}{\sim}_\beta \langle \Omega^k, m, pc', lt', ht' \rangle$ and $\Lambda \overset{!}{\sim}_\beta \Lambda'_{proj}$.*

Executions that get stuck do not need to be considered, since corresponding programs would not pass the BCV. Theorem 1 only partially deals with infinite computations: we e.g. do not show the preservation of executions when they diverge inside a method call. All reachable states (intra and inter-procedurally) could be matched giving small-step operational semantics to both languages. This would require parametrizing events by the method from which they arise, and extending the relation on configurations to all frames in the call stack.

We now provide a proof sketch of the theorem, giving an insight on the technical arguments used in the complete proof, which is given in [5]. We prove this theorem using a strong induction on the call depth n. The inductive reasoning is made possible by considering not only computations from initial states to (normal and error) return states, but also intermediate computation states. The crucial point is that BC intermediate states require dealing with the stack, to which BIR expressions must be related. Semantically, this is captured by a correctness criterion on the abstract stack used by the transformation. It intuitively means that expressions are correctly decompiled:

Definition 1 (Stack correctness: $\approx_{h,ht,lt,\beta}$) *Given $h, ht \in Heap$ such that $h \overset{\mathrm{H}}{\sim}_\beta ht$ and $lt \in Env_{\mathrm{BIR}}$, an abstract stack $as \in AbstrStack$ is said to be correct with regards to a run-time stack $s \in Stack$ if and only if $s \approx_{h,ht,lt,\beta} as$:*

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

where $ht, lt \vDash e \Downarrow v'$ means that expression e evaluates to v' in ht and lt.

The last definition rule says that the symbol $UR^{\mathsf{C}}_{\mathsf{pc}}$ correctly approximates a reference r of tag $\widetilde{\mathsf{C}}_{\mathsf{pc}}$. The alias information tracked by $UR^{\mathsf{C}}_{\mathsf{pc}}$ is made consistent if we additionally demand that all references appearing in the stack with the same status tag are equal to r (second condition of this last rule). This strong property is enforced by the restrictions imposed by the BCV on uninitialized references in the operand stack.

We are now able to state the general proposition on intermediate execution states. In order to clarify the induction hypothesis, we parametrize the proposition by the call depth and the name of the executed method:

Proposition 1 ($\mathcal{P}(n,m)$ **–** BC2BIR **n call-depth preservation)**
Let $m \in \mathbb{M}$ be a method of P (and P') and $n \in \mathbb{N}$. Let β be a partial bijection on Ref. Let $c = \langle h, m, pc, l, s\rangle \in State_{\mathrm{BC}}$ and $ct = \langle ht, m, (pc, instrAt_{P'}(m, pc)), lt\rangle \in State_{\mathrm{BIR}}$ such that $c \overset{\mathrm{c}}{\sim}_\beta ct$ and $s \approx_{h,ht,lt,\beta} \mathtt{AS_{in}[m,pc]}$. Then, for all $c' \in State_{\mathrm{BC}}$, whenever $c \overset{\Lambda}{\Rightarrow}_n c'$, there exist unique ct' and Λ' and a unique β' extending β such that $ct \overset{\Lambda'}{\Rightarrow}_n ct'$ with $c' \overset{\mathrm{c}}{\sim}_{\beta'} ct'$ and $\Lambda \overset{!}{\sim}_{\beta'} \Lambda'_{proj}$.

In the base case $\mathcal{P}(0,m)$, we reason by induction on the number of BC steps. A step $\langle h, m, pc, l, s\rangle \xrightarrow{\Lambda}_0 \langle h', m, pc', l', s'\rangle$ is matched by: $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\Lambda_1}{\Rightarrow}_0 \langle ht, m, (pc, \mathtt{code}), lt_0\rangle \overset{\Lambda_2}{\Rightarrow}_0 \langle ht', m, (pc', instrAt_{P'}(m, pc')), lt'\rangle$ where the intermediate state $\langle ht, m, (pc, \mathtt{code}), lt_0\rangle$ is obtained by executing the potential additional assignments prepended to the instructions $\mathtt{code}$ generated by BC2BIR$_{instr}$. We obtain the second part of the matching computation thanks to a correctness lemma about BC2BIR$_{instr}$ (proved in [5]):

Lemma 1 (BC2BIR$_{instr}$ **0 call-depth one-step preservation)**
Suppose $\langle h, m, pc, l, s\rangle \xrightarrow{\Lambda}_0 \langle h', m, pc', l', s'\rangle$. Let ht, lt, as, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$, $s \approx_{h,ht,lt,\beta} as$ and $\mathrm{BC2BIR}_{instr}(pc, instrAt_P(m, pc), as) = (\mathtt{code}, as')$. There exist unique ht', lt' and Λ' such that $\langle ht, m, (pc, \mathtt{code}), lt\rangle \overset{\Lambda'}{\Rightarrow}_0 \langle ht', m, (pc', instrsAt_{P'}(m, pc')), lt'\rangle$ with $h' \overset{\mathrm{H}}{\sim}_\beta ht'$, $l' \overset{\mathrm{E}}{\sim}_\beta lt'$, $\Lambda \overset{!}{\sim}_\beta \Lambda'_{proj}$ and $s' \approx_{h',ht',lt',\beta} as'$.

It is similar to $\mathcal{P}(n,m)$, but only deals with one-step BC transitions and does not require extending the bijection (instructions at a zero call depth do not initialize any object). Moreover, considering an *arbitrary* correct entry abstract stack allows us applying the lemma with more modularity.

Lemma 1 cannot be directly applied for proving the $\overset{\Lambda_2}{\Rightarrow}_0$ step, because the entry abstract stack $\mathtt{AS_{in}[m,pc]}$ is sometimes normalized and because of the additional assignments prepended to $\mathtt{code}$. For the hypotheses of Lemma 1 to be satisfied, we thus have to show that $s \approx_{h,ht,lt_0,\beta} \mathtt{AS_{in}[m,pc]}$. Two cases are distinguished. If $\mathtt{pc} \notin jmpTgt^{\mathtt{P}}_{\mathtt{m}}$, the stack is not normalized, but additional assignments could break the stack correctness. However, as we forbid backwards jumps on non-empty stacks, all $\mathtt{T}^{\mathtt{j}}_{\mathtt{pcj}}$ (where $\mathtt{pcj} \in succ(\mathtt{pc})$) assigned by T*Assign* cannot be used in the stack. Now, if $\mathtt{pc} \in jmpTgt^{\mathtt{P}}_{\mathtt{m}}$, then the stack is normalized. Assignments generated by T*Assign* do not alterate the stack correctness: if $\mathtt{pcj}$ is a join point successing $\mathtt{pc}$, $\mathtt{T}^{\mathtt{k}}_{\mathtt{pcj}}$ is assigned, but all the $\mathtt{T}^{\mathtt{k}'}_{\mathtt{pc}}$ that appear in the normalized stack are distinct from $\mathtt{T}^{\mathtt{k}}_{\mathtt{pcj}}$ ($\mathtt{pc}<\mathtt{pcj}$ if the stack at $\mathtt{pcj}$ is non-empty). Hence $s \approx_{h,ht,lt_0,\beta} \mathtt{AS_{in}[m,pc]}$.

Applying Lemma 1 gives us that $h' \overset{\mathrm{H}}{\sim}_\beta ht'$, $l' \overset{\mathrm{E}}{\sim}_\beta lt'$ and $\Lambda \overset{!}{\sim}_\beta \Lambda_{2proj}$. Furthermore, Λ_1 is only made of temporary variable assignment events, hence Λ_{1proj} is empty, and $\Lambda \overset{!}{\sim}_\beta (\Lambda_1.\Lambda_2)_{proj}$. Because of prepended assignments, we have to show that the transmitted abstract stack $\mathtt{AS_{in}[m,pc']}$ satisfies $s' \approx_{h',ht',lt',\beta} \mathtt{AS_{in}[m,pc']}$. There are two cases. If $\mathtt{pc'}$ is not a join point, then the transmitted abstract stack is simply $\mathtt{AS_{out}[m,pc]}$, resulting from BC2BIR$_{instr}$. We therefore use the conclusion of Lemma 1. Now, if $\mathtt{pc'} \in jmpTgt^{\mathtt{P}}_{\mathtt{m}}$, the output abstract stack is $newStackJmp(\mathtt{pc'}, \mathtt{AS_{in}[m,pc']})$. All of the

$\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}'}$ have been assigned, but we must show that they have not been modified by executing the BIR instructions `code`. As defined in Figure 6, the only assigned temporary variables are of the form $\mathtt{t}^{\mathtt{k}}_{\mathtt{pc}'}$. Our naming convention ensures $\forall \mathtt{k}.\, \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}'} \neq \mathtt{t}^{\mathtt{k}}_{\mathtt{pc}'}$. Thus, $s' \approx_{h',ht',lt',\beta} \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}']$, which concludes the proof of $\mathcal{P}(0, m)$.

Concerning the induction case $\mathcal{P}(n+1, m)$, the idea is to isolate one of the method calls, and to split the computation into three parts. Indeed, we know that there exist n_1, n_2 and n_3 such that a transition $c \Rightarrow_{n+1} c'$ can be decomposed into $c \Rightarrow_{n_1} c_1 \rightarrow_{n_2} c_2 \Rightarrow_{n_3} c'$, with $n_2 \neq 0$ and $n + 1 = n_1 + n_2 + n_3$. The first and third parts are easily treated applying the induction hypothesis. The method call $c_1 \rightarrow_{n_2} c_2$ is handled in a way similar to the base case. We prove an instruction-wise correctness intermediate lemma, under the induction hypothesis $\forall m'\, \mathcal{P}(n, m')$. The induction hypothesis is also applied on the execution of the callee, whose call depth is strictly lower.

5 Related Work

Many Java bytecode optimization and analysis tools work on an IR of bytecode that make its analysis much simpler. Soot [14] 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. The stack elimination is performed in two steps, when generating Jimple code from Baf code (see [15] for details). First, naive 3-address code is produced (one variable is associated to each element position of the stack). Then, numerous redundancies of variables are eliminated using a simple aggregation of single def-use pairs. Variables representing stack locations lead to type conflicts when their type is infered, so that they must be desambiguated using additional variables. Our transformation, relying on a symbolic execution, avoids this problem by only merging variables of distinct scopes. Auxiliary analyses (e.g. copy propagation) could further reduce the number of variables, but BC2BIR generates very few superfluous variables in practice [5].

The transformation technique used in BC2BIR is similar to what Whaley [16] uses for the high level IR of the Jalapeño Optimizing Compiler [3] (now part of the Jikes virtual machine [10]). The language provides explicit check operators for common runtime exceptions (`null_check`, `bound_check`...), so that they can be easily moved or eliminated by optimizations. We use a similar technique to enforce the preservation of the exception throwing order. We additionally use the `mayinit` instruction to ensure the preservation of the class initialization order, that could otherwise be broken because of folded constructors and side-effect free expressions. Our work 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 (see [5]). All these previous works have been mainly concerned with the construction of effective and powerful tools but, as far as we know, no 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 contexts of Java Bytecode analysis. The technique was already used in one of the first Sun Just-In-Time compilers [4] for direct translation of bytecode to machine instructions. Xi and Xia propose a dependent type system for array bound check elimination [18]. They use symbolic expressions to type operand stacks with *singleton* types in order to recover relations between lengths of arrays and index expressions. Besson *et al.* [2], and independently Wildmoser *et al.* [17], propose an extended interval analysis using symbolic decompilation that verifies that programs are free of out-of-bound array accesses. 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.* [1] also use a symbolic manipulation for the relational analysis of a simple bytecode language and prove it is as precise as a similar analysis at source level.

Among the numerous works on program transformation correctness proofs, the closest are those dealing with formal verification of the Java compiler algorithms (from Java source to Java bytecode) [12, 13, 7]. The present work studies a different transformation from bytecode to a higher intermediate level and handle difficulties (symbolic operand stack, non preservation of allocation order) that were not present in these previous works.

6 Conclusions and Future Work

This paper provides a semantically sound, provably correct transformation of bytecode into an IR 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 the accompanying technical report [5] we demonstrate on several examples of safety properties how some BIR static analysis verdicts can be translated back to the initial BC program. It would be interesting to study whether the translation of analysis results could be simplified by expressing BC2BIR in the form of annotations, as proposed by Matsuno and Ohori in [9] for the Static Single Assignment form. By the nature of the transformation, and because of the differences between BC and BIR, expressing BC2BIR in this setting would require several adaptations. The transformation is 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 accepting full Java bytecode. Our benchmarks show the expected efficiency is obtained in practice.

Several other 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. Second, it would be interesting to study if the transformation scheme would fit a more multi-language support such as CIL, the output format of several compilers (`VB.NET`, `C#`...). On one hand, this would require to adapt the formalization to the low-level memory operations available in this language. On the other hand, we could lift the constraints on the use of uninitialized objects by MSIL input programs, since constructor calls are folded in CIL. Finally,

we believe the current transformation would be a valuable layer on top of Bicolano, a formal JVM semantics formalized in Coq and developed during the European MOBIUS project. The Coq extraction mechanism would allow extracting certified and efficient Caml code from the Coq formalization of the algorithm.

Acknowledgments. We thank the anonymous reviewers for their thorough comments.

References

1. G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of proof obligations for hybrid verification methods. In *Proc. of SEFM 2008*, pages 127–136. IEEE Computer Society, 2008.
2. 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.
3. 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.
4. 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.
5. D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for Java bytecode. Research Report 7021, INRIA, 2009. `http://www.irisa.fr/celtique/ext/bir/rr7021.pdf`.
6. Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM TOPLAS*, 21(6):1196–1250, 1999.
7. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
8. 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.
9. Yutaka Matsuno and Atsushi Ohori. A type system equivalent to static single assignment. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 249–260. ACM, 2006.
10. The Jikes RVM Project. Jikes rvm - home page. `http://jikesrvm.org`.
11. E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.
12. R. F. Stark, E. Borger, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*. Springer-Verlag New York, Inc., 2001.
13. M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. of CADE-18*, pages 63–77. Springer-Verlag, 2002.
14. 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.
15. Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations, 1998.
16. J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.
17. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of BYTECODE 2005, Electronic Notes in Computer Science*, 2005.
18. 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.