Science of Computer Programming 191 (2020) 102392

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

From CIL to Java bytecode: Semantics-based translation for static analysis leveraging

Pietro Ferrara [a,b,*], Agostino Cortesi [b], Fausto Spoto [c]

[a] JuliaSoft, Verona, Italy
[b] Università Ca' Foscari di Venezia, Italy
[c] Università di Verona, Italy

ARTICLE INFO

Article history:
Received 28 June 2019
Received in revised form 20 December 2019
Accepted 8 January 2020
Available online 31 January 2020

Keywords:
Static analysis
Abstract interpretation
Java bytecode
CIL

ABSTRACT

A formal translation of CIL (*i.e.*, .Net) bytecode into Java bytecode is introduced and proved sound with respect to the language semantics. The resulting code is then analyzed with Julia, an industrial static analyzer of Java bytecode. The overall process of translation and analysis is fast, scales to industrial programs, and introduces a negligible number of false alarms. The main contribution of this work is to leverage existing, mature, and sound analyzers for Java bytecode by applying them also to the wide range of .Net software systems. Experimental results show the actual effectiveness of this approach when applied to all the system libraries of the Microsoft .Net framework version 4.0.30319 (about 5 MLOCs).

1. Introduction

Static analysis infers, at compile-time, properties about the run-time behavior of computer programs. It allows one to verify, for instance, the absence of run-time errors or security breaches. Static analysis applies also to compiled code in assembly or bytecode format. This is particularly interesting for applications distributed on the Internet, or downloaded from public (and possibly unsafe) application repositories (*e.g.*, the Google Play store), when the source code is not available, but the user would like to statically check some safety or security properties.

The analysis of Java bytecode (from now on, JB) for the Java Virtual Machine has a long research tradition and many analyzers exist [1]. Some analyses build on formal mathematical roots, such as abstract interpretation [2–6]. Moreover, JB makes the design of static analysis easier by requiring bytecode to be type-checkable [7] and without unsafe operations such as free pointer operations. On the contrary, CIL bytecode, that is, the compiled bytecode used for the .Net platform (from now on, just CIL), has not received much attention from the static analysis community yet. Therefore, while there are several syntactic (that is, not based on formal methods) static analyzers for CIL, there are very few industrial semantic analyzers based on formal methods. Moreover, CIL can be used in an *unsafe* way, that is, by allowing free pointer operations, which makes its static analysis harder. However, these operations are very often used in very controlled contexts. Hence, in most cases, a static analyzer could possibly capture their actual behavior anyway.

Despite clear differences, JB and CIL share strong similarities: both are low-level object-oriented languages where objects are stored and shared in the heap. Hence, it is tempting to leverage mature existing static analyses and tools for JB by translating CIL into *equivalent* JB and by running the tools on the latter. Obviously, this introduces issues about the exact

* Corresponding author.
E-mail addresses: pietro.ferrara@unive.it (P. Ferrara), cortesi@unive.it (A. Cortesi).

https://doi.org/10.1016/j.scico.2020.102392

meaning of *equivalence* between CIL and its translation into JB. Moreover, the translation should not introduce code artifacts that confuse the analyzer and should work on industrial-size CIL applications, supporting as many unsafe pointer operations as possible.

1.1. Contribution

The main contribution of this work[1] is the introduction of a translation of CIL to JB that is both theoretically sound and effective in practice, so that an industrial static analyzer for JB can be applied to .Net (and in particular C#) programs. More languages compile into JB (*e.g.*, Java and Scala) and CIL (*e.g.*, VB.Net and F#), with distinct features and code structures. Here, we focus on Java and C#, that have similar structure and compile into comparable bytecode.

We start by formalizing the concrete semantics of a representative subset of CIL and JB, and the translation of CIL into JB. Then, we prove this translation sound, in the sense that the concrete semantics of the initial CIL program is equivalent to that of the translated JB program. This guarantees that, if we prove a property of the JB program, then such property holds also for the original CIL program. Our implementation supports the full set of CIL safe statements. Fig. 7 in Appendix B reports all the CIL statements as defined by ECMA 335 standard [9], and if and how we translate them. Then we present a deep experimental evaluation over industrial-size open-source popular programs, by applying the Julia static analyzer [4] to the translated JB.

Our experiments are focused on three main research hypotheses to prove the scalability, precision, and coverage of our approach. In particular, this article answers the following research questions:

Research Question 1 *(Scalability). Does the CIL to JB translation scale, that is, (i) can it deal with libraries of industrial size (100 KLOCs) in a few minutes, and (ii) is its computational time negligible w.r.t. that required by the overall analysis?*

Research Question 2 *(Precision). Does the CIL to JB translation introduce less than 10% of false alarms w.r.t. the overall number of alarms produced by the analyzer when considering high severity warnings?*

Research Question 3 *(Libraries). Despite supporting only a subset of CIL, does the CIL to JB translation succeed on at least 95% of the system libraries?*

About scalability, the overall goal is to obtain a tool that can be applied during the normal software development life-cycle, that is, it can perform the analysis at each build of the system. Therefore, we need a tool that analyzes hundreds of thousands of LOCs in few minutes.

About precision, the 10% bound was chosen as such limit is perceived as standard by the community when evaluating the effectiveness of static analyzers. False alarms are not real issues in the code, but they are due to some forms of approximation performed by the analysis. For instance, a recent paper [10] stated that Google adopts a static analyzer during code review if it "produces less than 10% effective false positives" (aka, alarms), since such amount of *noise* does not compromise the practical effectiveness of the tool when used by software developers. Note that our research question refers to the false alarms due to the translation, and that would not be produced on the same code written in Java by the Java analyzer. For instance, let us assume that an analyzer produces 10 false alarms on a Java program. Research Question 2 is satisfied if the analysis on the same program written in .NET (and translated to JB before being analyzed by the Java analyzer) produces at most 11 false alarms (the 10 produced on the Java program – that are due to the imprecision of the analysis – and 1 introduced by the translation). In our experience, these false alarms usually happen because the .NET compiler introduces some patterns (e.g., when checking the nullness of a variable) that are different from those produced by the Java compiler, and the Java static analyzer approximates them too coarsely.

With respect to question 3, it is crucial that a static analyzer understands the behavior of system libraries (*e.g.*, semantics of method calls and assumptions made on their parameters) or otherwise it could only rely on manual annotations or (possibly unsound) assumptions on their execution. However, system libraries need to access memory through unsafe pointer. Java allows such behaviors through native methods (written in languages other than Java and bound through the Java Native Interface), while .Net allows unsafe pointers directly in its same code. In these cases, our translation produces Java native methods. Through Research Question 3, we ensure that the effort of manually annotating .Net libraries is comparable to that needed for Java.

Our translation could also be applied to let Java and C# code interoperate, by compiling them both into Java bytecode. This requires, however, the translation of the libraries called by the applications, including the standard library of C# that contains relevant portions of unsafe and native code, which is out of the scope of this work.

[1] This paper is a revised and extended version of [8], distinguished paper award at ICSE-FormaliSE (2018). In particular, this paper adds detailed formal proofs of soundness in Section 4.3, and extended experimental results in Section 5.

1.2. Related work

Few attempts have been made in the past to translate CIL to JB. Grasshopper is probably the most popular one. However, it is not available anymore.[2] As far as we can see, it was abandoned about a decade ago, and we cannot make any comparison with our translation. A similar tool is CLR2JVM [11]: it translates CIL to an intermediate XML_{CLR} representation, that can be then translated into XML_{JVM}, and finally to JB. As far as we can see,[3] the tool should read .NET executables, but it failed to parse all the executable files of our experiments (see Section 5 for the complete list). This probably happened because CLR2JVM is not maintained any more (the last commit to the repository https://sourceforge.net/p/xmlvm/code/HEAD/tree/trunk/xmlvm/src/clr2jvm/ occurred more than six years ago), and it does not support the last CIL versions. Neither Grasshopper nor CLR2JVM has any documentation or discussion about how the translation is performed (in particular, how they handle instructions that are different between CIL and JB, such as direct references). Therefore, as far as we can see, our translation from CIL to JB is the only one that works on recent releases of CIL and JB, and is formalized and proved sound.

Other translations between low-level languages exist, justified by the need of applying verification tools that work on a specific language only. For instance, [12] defines a translation from Boogie into WhyML and proves its soundness, as we have done from CIL to JB. Similar translations work also at run time, in particular inside a just-in-time compiler, as in [13]. However, we did not find any literature on the translation of CIL into JB for industrial-size software.

Many other static analyzers for .Net exist, in particular for C# code. There are tools that verify compliance to some guideline, such as Fxcop [14] and Coverity Prevent [15]. Others, such as NDepend [16] and CodeMetrics [17], provide metrics about the code under analysis. ReSharper [18] applies syntactical code inspections, finds code smells and guarantees compliance against coding standards. As far as we can see, there exist only two main fundamental tools with scientific base: Spec# [19], an extension of C# with static checking of various kinds of manual specifications, and CodeContracts [20], an abstract interpretation-based static analyzer for CIL. In the Java world, the number of static analyzers based on syntactic reasoning (that is, not based on formal methods), such as Checkstyle [21], FindBugs [22] and PMD [23], is comparable to that for .Net. However, Java attracted much more attention from the scientific community, and more semantic analyzers have been introduced during the last decade, such as CodeSonar [24], ThreadSafe [25] and Julia [4]. Instead, there are quite fewer semantic static analyzers for .NET. In addition, few semantic analyzers, such as WALA [26], have been applied to various languages (*e.g.*, Java and JavaScript), but with ad-hoc translation of the source to the analyzed language.

Our approach lets us apply all the Java analyzers on .Net programs (almost) *for free*, that is, by translating CIL into JB and by using the analyzers as they are (we expect that few manual annotations are needed to improve the precision of the analysis, in particular when dealing with library calls). We have also studied performance and results with Julia. As far as we know, our work is the first translation of CIL into JB for static analysis that is proven to be *sound* and comes with evidence that this translation applies to *industrial-size software*, with results that are comparable in terms of precision and efficiency to those obtained on JB.

2. Background

This section provides some background on CIL and JB, a running example, and a discussion on the architecture of our approach. For an exhaustive definition of JB and CIL, see [7] and [9], respectively.

2.1. CIL and JB

Bytecode is a machine-independent low-level programming language, used as target of the compilation of high-level languages, that hence becomes machine-independent. Bytecode languages are interpreted by their corresponding virtual machine, specific to each execution architecture. Both .Net and Java compile into bytecode. However, they use distinct instructions and virtual machines. .Net compiles into CIL, while Java compiles into JB. These have strong similarities: both use an operand stack for temporary values and an array of local variables standing for source code variables; both are object-oriented, with instructions for object creation, field access and virtual method dispatch. Despite these undeniable similarities, CIL and JB differ for the way of performing parameter passing (CIL uses a specific array of variables for the formal parameters, while JB merges them into the array of local variables); they handle object creation differently (CIL creates and initializes the object at the same time, while these are distinct operations in JB); they allocate memory slots differently (in CIL each value uses a slot, while JB uses 1 or 2 slots per 32- or 64-bit values, respectively); finally, CIL uses pointers explicitly, also in type-unsafe ways, while JB has no notion of pointer. We focus our formalization on a minimal representative subset of JB and CIL, as defined in Fig. 1. That figure presents bytecode instructions for:

arithmetic: JB has type-specific operations, such as `iadd` and `ladd` to add two integer or long values, respectively. Instead, CIL has generic operations, such as `add` to add two numerical values of the same type;

[2] We were unable to access the website http://dev.mainsoft.com that, from past forum discussions (http://stackoverflow.com/questions/95163/differences-between-msil-and-java-bytecode), seems to be the website of the tool.

[3] http://xmlvm.org/documentation/.

JB	CIL	
`iadd` `ladd`	`add`	*(arith. op.)*

`iload i`
`lload i`
`aload i`
`istore i`
`lstore i`
`astore i`
	`ldloc i`
	`stloc i`
	`ldarg i`
		(local vars)

`invokevirtual` `invokestatic`	`call`	*(meth. call)*

`new T`
`getfield f`
`putfield f`
	`newobj T(···)`
	`ldfld f`
	`stfld f`
		(objects)

`if_icmpgt`	`bgt`	*(cond. branch)*
`dup` `dup2`	`dup`	*(stack)*

	`ldloca i`
	`stind`
	`ldind`
		(pointers)

Fig. 1. JB and CIL minimal bytecode languages.

local variables access: JB has a single array of variables for both local variables and method arguments, and reads and writes values from this array through `xload` and `xstore`, where `x` is `i` for integer values, `l` for long values and `a` for references, respectively. In this array, 64 bits values use two subsequent slots. CIL, instead, uses two arrays: one for method's arguments (`ldarg i` loads the value of the i-th argument) and one for local variables (`ldloc i` and `stloc i` read and write the i-th local variable, respectively). In addition, it uses one slot both for 32- and 64-bit variables;

method call: JB has several kinds of method call instructions, such as `invokevirtual` and `invokestatic`. Instead, CIL has a unique `call` instruction;

object manipulation: in JB, instructions `new`, `getfield`, and `putfield` allocate a new object, read, and write its fields, respectively. CIL has similar instructions `newobj`, that also calls the constructor, `ldfld` and `stfld`;

conditional branch: JB has type-specific conditional branch instructions such as `if_icmpgt` (to branch if the greater than operator returns true on the topmost two integer values of the operand stack); CIL has generic instructions such as `bgt`;

stack: CIL duplicates the top value of the stack through the `dup` instruction. JB does the same with `dup` for 32 bits values and `dup2` for 64 bits values;

pointers: CIL contains some instructions to load the address of a local variable (`ldloca i`), and to store and load a value into the memory cell pointed by a reference (`stind` and `ldind`, respectively). Instead, JB has no direct pointer manipulation.

In the rest of this article, $\mathsf{St}_{\mathsf{CIL}}$ and $\mathsf{St}_{\mathsf{JB}}$ denote CIL and JB instructions or statements, respectively. A method (both in CIL and JB) is represented by (i) a sequence of (possibly conditional and branching) statements, and (ii) the number and static types of arguments and local variables.

2.2. Running example

Fig. 2 shows the running example that Sections 3 and 4 use to clarify the formalization. The C# code in Fig. 2a defines a class `Wrap` that wraps an integer value, and a static method `WrapsCollection` that, given an integer `n`, returns a collection of n wrappers containing values from 0 to $n-1$. Fig. 2b presents the (simplified) CIL obtained from its compilation: as usual with CIL, code is unstructured (*e.g.*, there are branches at lines 7 and 20), and each source code statement could be translated into many bytecode statements (*e.g.*, line 7 in Fig. 2a compiles into lines 3 and 4 in Fig. 2b). Fig. 2c presents the results of our translation of CIL into JB. The next sections explain the steps of the translation. First of all, notice some of the differences highlighted in Section 2.1. Namely, the type-generic CIL statement `stloc.1` at line 6 of Fig. 2b is translated into the type-specific `istore_2` at line 7 of Fig. 2c. Similarly, `newobj` (line 11) is translated into multiple JB statements (line 12-16). The running example contains some instructions that are not part of the minimal language defined in Section 2.1, and in particular (i) `ldc` and `iconst_*` that load constant (integer) values in CIL and JB, respectively, (ii) `blt` and `if_icmplt` for conditional branching when an (integer) value is strictly less than another, (iii) `invokespecial` to invoke a specific method in JB, and (iv) `ret` and `areturn` to return a (reference) value.

```
1  public class Wrap
2  {
3    int f;
4    Wrap(int f) { this.f = f; }
5    static ICollection<Wrap>WrapsCollection(int n)
6    {
7      ICollection<Wrap> result = new List<Wrap>();
8      for (int i = 0; i < n; i++)
9        result.Add(new Wrap(i));
10     return result;
11   }
12 }
```

(a) The C# source of the running example.

```
1  static ICollection<Wrap>
2    WrapsCollection(int n) {
3    newobj List<Wrap>::.ctor()
4    stloc.0
5    ldc.0
6    stloc.1
7    br #18
8
9    ldloc.0
10   ldloc.1
11   newobj Wrap::.ctor(int32)
12   call ICollection<Wrap>::Add
13   ldloc.1
14   ldc.1
15   add
16   stloc.1
17
18   ldloc.1
19   ldarg.0
20   blt #9
21
22   ldloc.0
23   ret
24 }
```

(b) Compiling (a) into CIL.

```
1  static WrapsCollection(I)LICollection {
2    new List
3    dup
4    invokespecial List.<init>
5    astore_1
6    iconst_0
7    istore_2
8    goto 23
9
10   aload_1
11   iload_2
12   istore_3
13   new Wrap
14   dup
15   iload_3
16   invokespecial Wrap.<init>
17   invokevirtual <ICollection.Add>
18   iload_2
19   iconst_1
20   iadd
21   istore_2
22
23   iload_2
24   iload_0
25   if_icmplt 10
26
27   aload_1
28   areturn
29 }
```

(c) Translating (b) into JB.

Fig. 2. The C# code, CIL, and JB of the running example.

```
1  void init(ref int i)
2  {
3    i++;
4  }
5
6  void run()
7  {
8    int i=0;
9    init(ref i);
10 }
```

(a) C# code

```
1  void init (WrapRef i) {
2    i.value = i.value + 1;
3  }
4
5  int run() {
6    int i = 0;
7    WrapRef wrap = new WrapRef();
8    wrap.value = i;
9    init (wrap);
10   i = wrap.value;
11 }
```

(b) Java code

```
1  void init(ref A& A)
2  {
3    ldarg.0
4    ldarg.0
5    ldind.i4
6    ldc.i4.1
7    add
8    stind.i4
9  }
10
11 int run()
12 {
13   ldc.i4.0
14   stloc.0
15   ldloca.s 0
16   call void Temporary.Foo
17     ::' init '(int32&)
18 }
```

(c) CIL

Fig. 3. CIL code using `ref` parameters.

2.2.1. Example with direct references

Safe C# code adopts direct pointers only for `out` and `ref` method parameters. These parameters can be assigned (and read as well in case of `ref`) inside the method, and "any change to the parameter in the called method is reflected in the calling method".[4] In our translation, we build wrapper objects to soundly represent their semantics. Consider for instance the C# code in Fig. 3a. Method `init` receives a `ref` parameter and it increments it by one. This is compiled (Fig. 3c) into a method that reads the value pointed by the direct reference (line 5), and writes it (line 8). Our goal is to translate this code into the Java code in Fig. 3b: we simulate the direct reference by constructing a wrapper object (line 7), assigning the value of the local variable to field `value` of the wrapper (line 8), and then propagating back the results of the call to `init` (line 9) by assigning the value of the local variable with the one stored in the wrapper (line 10). In addition, the `ref` parameter is replaced by the type of the wrapper object.

[4] https://msdn.microsoft.com/en-us/library/14akc2c7.aspx.

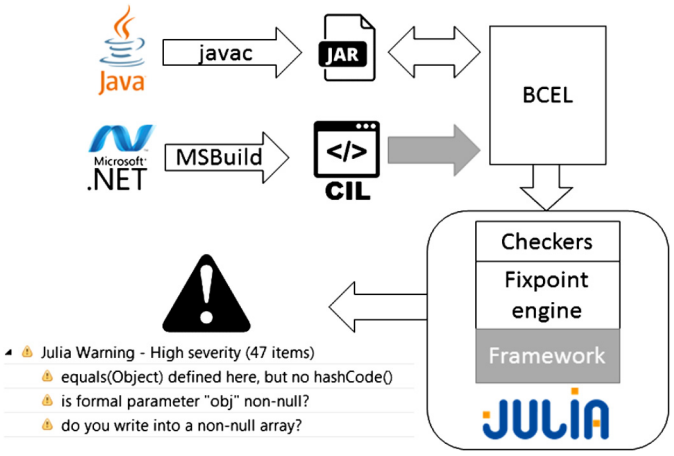

Fig. 4. Architecture of the analysis.

2.3. Julia

Julia [4] is a static analyzer for JB, based on abstract interpretation [2]. It transforms JB to basic blocks of code, that it analyzes through a fixpoint algorithm. Analyses are constraint-based or denotational. Currently, Julia features around 70 checkers, including nullness, termination, synchronization and taint analysis. Since Julia works on JB, in principle it analyzes any programming language that compiles into that bytecode. In particular, in this article we apply it to the analysis of the compilation of CIL into JB. Note that Julia verifies that the analyzed bytecode is well-formed, that is, it mainly performs sanity checks to avoid the static analyzer to fail while analyzing the bytecode. Therefore, all the translated bytecode is verified by Julia before being analyzed. Julia relies on Java annotations in order to allow a user to specify the semantic model of some components (e.g., if a method is an entry point in a particular runtime environment, or an API call returns user input). https://static.juliasoft.com/docs/latest/annotations.html reports the full list of Julia's annotations. While these annotations add semantic information about the application and the libraries, they do not affect the translation from CIL to JB.

Fig. 4 is a high level view of our analysis, where the gray components have been implemented and modified to support .NET analyses. Java code is compiled by `javac` into a `jar` file, then parsed through the Byte Code Engineering Library [27] (BCEL). Julia receives this latter format, applies its analysis (by using many components such as the checkers, that define the analyses relying on various abstract domains, a fixpoint engine, and the framework specifying the semantics of some specific components of the programming language), and outputs a list of warnings. The added component in our approach is the translation of CIL into BCEL format (grey arrow in the upper part of Fig. 4). Since a program in BCEL format can be dumped into a jar file, we can dump a .dll file in this format.

We instructed Julia by adding manual annotations to the main components of the .NET run-time environments. All together, this amounts to about 1.500 annotations on the main library APIs. For instance, we annotated field System.Decimal.One to specify that it is externally initialized with Julia's annotation Injected, and method System.Web.HttpRequest.Params with UntrustedUserInput to specify this returns a user input (that is, it is a source for Julia's Injection analysis).

3. Concrete semantics

3.1. Notation

We briefly recall some standard notation adopted by the further formalization in this section and the next one.

Common mathematical elements. First of all, we adopted standard set notations, that is: (i) $a \in A$ denotes that element a is contained in set A, (ii) $A \times B$ denotes the Cartesian product of sets A and B (that is, the set of all ordered pairs (a, b) such that $a \in A$ and $b \in V$, (iii) $\cup$ and $\cap$ denote set union and intersection, respectively, and (iv) $|A|$ denotes the cardinality (that is, the number of elements) of set A.

About functions, we denote by (i) $F : D \to C$ the set of all functions relating elements in a domain D to a codomain C, (ii) $[d \mapsto c]$ the function that relates element d to c, (iii) $f[d \mapsto c]$ the function that behaves like f except for element d that is related to c, and (iv) $f(d)$ the application of function f to the element of the domain d (e.g., $f(d) = c$ if $f = [d \mapsto c]$).

We represent lists and stacks as functions mapping natural indexes to elements. In particular, a stack s (or list) of elements in T is a function in $\mathbb{N} \to T$ such that $\exists i \in \mathbb{N} : \forall i_1 \leq i : i_1 \in dom(s) \wedge \forall i_2 > i : i_2 \notin dom(s)$; $dom(s)$ denotes the domain of the given function (that is, the indexes on which stack s is defined in this particular case). We will refer to i as the height of s ($height(s)$). Given a stack s and an element e, $s :: e$ denotes a stack whose top element (that is, the one with

$$\frac{typeOf(v_1) = typeOf(v_2)}{\langle \texttt{add}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: (v_1 + v_2), \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{add}) \qquad \frac{}{\langle \texttt{ldloc i}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{l}(\mathtt{i}), \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{ldloc})$$

$$\frac{}{\langle \texttt{stloc i}, (s :: \mathsf{v}, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}[\mathtt{i} \mapsto \mathsf{v}], \mathsf{a}, \mathsf{h})} \ (\texttt{stloc}) \qquad \frac{}{\langle \texttt{ldarg i}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{a}(\mathtt{i}), \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{ldarg})$$

$$\frac{\begin{array}{c}\mathit{isStatic}(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i})) = \mathsf{false} \wedge t \neq \mathtt{null} \wedge \\ \langle \mathit{body}(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i}), (t, v_1, \cdots, v_i)), ([], \emptyset, [0 \mapsto t, j \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')\end{array}}{\langle \texttt{call m}(\mathtt{arg}_1, \cdots, \mathtt{arg}_\mathtt{i}), (s :: t :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}')} \ (\texttt{call})$$

$$\frac{\begin{array}{c}\mathit{isStatic}(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i})) = \mathsf{true} \wedge \\ \langle \mathit{body}(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i}), (v_1, \cdots, v_i)), ([], \emptyset, [j-1 \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')\end{array}}{\langle \texttt{call m}(\mathtt{arg}_1, \cdots, \mathtt{arg}_\mathtt{i}), (s :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}')} \ (\texttt{call static})$$

$$\frac{\mathit{fresh}(\mathtt{T}, \mathsf{h}) = (r, \mathsf{h}_1) \wedge \langle \mathit{body}(\mathtt{ctor}(\mathtt{arg}_1, \cdots, \mathtt{arg}_\mathtt{i}), (v_1, \cdots, v_i)), ([], \emptyset, [0 \mapsto r, j \mapsto v_j : j \in [1..i]], \mathsf{h}_1)\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')}{\langle \texttt{newobj T}(\mathtt{a}_1, \cdots, \mathtt{a}_\mathtt{i}), (s :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: r, \mathsf{l}, \mathsf{a}, \mathsf{h}')} \ (\texttt{newobj})$$

$$\frac{o \neq \mathtt{null}}{\langle \texttt{ldfld f}, (s :: o, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{h}(o)(\mathtt{f}), \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{ldfld}) \qquad \frac{o \neq \mathtt{null} \quad s' = \mathsf{h}(o)[\mathtt{f} \mapsto v]}{\langle \texttt{stfld f}, (s :: o :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}[o \mapsto s'])} \ (\texttt{stfld})$$

$$\frac{typeOf(v_1) = typeOf(v_2) \wedge v_1 > v_2}{\langle \texttt{bgt l}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} \langle \mathtt{l}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle} \ (\texttt{bgt true}) \qquad \frac{typeOf(v_1) = typeOf(v_2) \wedge v_1 \leq v_2}{\langle \texttt{bgt l}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{bgt false})$$

$$\frac{}{\langle \texttt{ldloca i}, (s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: r_i, \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{ldloca}) \qquad \frac{}{\langle \texttt{stind}, (s :: r_i :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}[r_i \mapsto v])} \ (\texttt{stind})$$

$$\frac{}{\langle \texttt{dup}, (s :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: v :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{dup}) \qquad \frac{}{\langle \texttt{ldind}, (s :: r_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{h}(r_i), \mathsf{l}, \mathsf{a}, \mathsf{h})} \ (\texttt{ldind})$$

Fig. 5. Concrete CIL semantics.

the highest index) is e followed by the stack s. In addition, $< \ldots >$ denotes a list of elements (e.g., $< a, b >$ denotes the list represented by the function $[0 \mapsto a, 1 \mapsto b]$).

Semantics. When formalizing concrete states and semantics, we denoted by Σ the set of concrete states of execution for JB (Σ_{JB}) and CIL (Σ_{CIL}). The small step semantics of JB and CIL is denoted by $\rightarrow_{\mathsf{JB}}$ and $\rightarrow_{\mathsf{CIL}}$, respectively, where, for instance, $\langle \mathsf{st}, \sigma_1\rangle \rightarrow_{\mathsf{JB}} \sigma_2$ represents that the execution of JB statement st with an entry state σ_1 results in the exit state σ_2. The small step semantics is defined through inference rules. For instance,

$$\frac{n' = n + 1}{\langle \mathsf{increment}, n\rangle \rightarrow \langle n'\rangle}$$

formalizes that when we execute statement increment on a value n we obtain a value n' defined as $n' = n + 1$.

Object-oriented components. We denote the sets of reference and numerical values by Ref and Num, respectively, and values by $\mathsf{Val} = \mathsf{Ref} \cup \mathsf{Num}$. As usual for object-oriented programming languages, an object is a map from field names to values, and a heap is a map from references to objects. Formally, $\mathsf{Heap} : \mathsf{Ref} \rightarrow \mathsf{Field} \rightarrow \mathsf{Val}$, where Field is the set containing all field names. $\mathit{fresh}(T, \mathsf{h}) = (r, \mathsf{h}')$ allocates an object of type T in heap h and returns (i) the reference r of the freshly allocated object, and (ii) the heap h′ resulting from the allocation of memory on h.

For simplicity, we consider only integer (Int) and long (Long) numerical types ($\mathsf{NumTypes} = \{\mathsf{Int}, \mathsf{Long}\}$), and references (Ref). Given a value v, $\mathit{typeOf}(v)$ returns its type (Int, Long, or Ref). Since JB instructions often prepend a prefix to distinguish instructions dealing with different types (*e.g.*, `iadd` and `ladd`), we define a support function *JVMprefix* that, given a type t, returns the prefix of the given type (*i.e.*, `i` if $t = \mathsf{Int}$, `l` if $t = \mathsf{Long}$, and `a` if $t = \mathsf{Ref}$). We define by WRef an object type with a unique field value.

Given a method signature `m` and a list of arguments L (with the receiver in the first argument if `m` is not static), $\mathit{body}(\mathtt{m}, L) : \mathbb{N} \rightarrow \mathsf{St}$ returns the body of the method resolving the call, that is, a sequence of statements (represented by a function mapping indexes to statements). Similarly, each statement belongs to a method; hence $\mathit{getBody}(\mathtt{st}) = b$ is the body of the method where `st` occurs. Finally, $\mathit{isStatic}(\mathtt{m})$ means that `m` is static.

3.2. CIL

We define the concrete semantics of the CIL fragment of Section 2.1.

Concrete State. A local state in CIL is composed by a stack of values or reference to local variables $\mathsf{Stack} : \mathbb{N} \rightarrow \mathsf{Val} \cup \mathsf{Ref}_{\mathsf{Loc}}$ (where $r_i \in \mathsf{Ref}_{\mathsf{Loc}}$ represents the cell's reference of the i-th local variable), an array of local variables $\mathsf{Loc} : \mathbb{N} \rightarrow \mathsf{Val}$, and an array of method arguments $\mathsf{Arg} : \mathbb{N} \rightarrow \mathsf{Val}$. A concrete CIL state consists of a local state and a heap, that is, $\Sigma_{\mathsf{CIL}} = \mathsf{Stack} \times \mathsf{Loc} \times \mathsf{Arg} \times \mathsf{Heap}$.

Concrete Semantics. Fig. 5 shows the concrete CIL semantics $\langle \mathtt{st}, \sigma\rangle \rightarrow_{\mathsf{CIL}} \sigma'$. For a statement `st` and an entry state σ, it yields the state σ' resulting from the execution of `st` over σ; or a program label `l`, meaning that the next instruction to

$$\frac{typeOf(v) \neq \mathsf{Long}}{\langle \texttt{dup}, (s :: v, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: v :: v, \mathsf{l}, \mathsf{h})} \ (\texttt{dup})$$

$$\frac{typeOf(v_1) \neq \mathsf{Long}}{\langle \texttt{dup2}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: v_1 :: v_2 :: v_1 :: v_2, \mathsf{l}, \mathsf{h})} \ (\texttt{dup2 32}) \qquad \frac{typeOf(v) = \mathsf{Long}}{\langle \texttt{dup2}, (s :: v, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: v :: v, \mathsf{l}, \mathsf{h})} \ (\texttt{dup2 64})$$

$$\frac{typeOf(v_1) = \mathsf{Int} \wedge typeOf(v_2) = \mathsf{Int}}{\langle \texttt{iadd}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: (v_1 + v_2), \mathsf{l}, \mathsf{h})} \ (\texttt{iadd}) \qquad \frac{typeOf(v_1) = \mathsf{Long} \wedge typeOf(v_2) = \mathsf{Long}}{\langle \texttt{ladd}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: (v_1 + v_2), \mathsf{l}, \mathsf{h})} \ (\texttt{ladd})$$

$$\frac{x = JVMprefix(typeOf(\mathsf{l}(\mathtt{i})))}{\langle x\texttt{load i}, (s, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: \mathsf{l}(\mathtt{i}), \mathsf{l}, \mathsf{h})} \ (x\texttt{load}) \qquad \frac{x = JVMprefix(typeOf(\mathsf{v}))}{\langle x\texttt{store i}, (s :: \mathsf{v}, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}[\mathtt{i} \mapsto \mathsf{v}], \mathsf{h})} \ (x\texttt{store})$$

$$\frac{\begin{array}{c} isStatic(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i})) = \mathsf{false} \wedge t \neq \mathtt{null} \wedge \\ \langle body(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i}), (t, v_1, \cdots, v_i)), ([], [0 \mapsto t, j \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (\mathsf{s}', \mathsf{l}', \mathsf{h}') \end{array}}{\langle \texttt{invokevirtual m}(\mathtt{arg}_1, \cdots, \mathtt{arg}_\mathtt{i}), (s :: t :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h}')} \ (\texttt{invokevirtual})$$

$$\frac{\begin{array}{c} isStatic(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i})) = \mathsf{true} \wedge \\ \langle body(\mathtt{m}(\mathtt{arg}_0, \cdots, \mathtt{arg}_\mathtt{i}), (v_1, \cdots, v_i)), ([], [j - 1 \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (\mathsf{s}', \mathsf{l}', \mathsf{h}') \end{array}}{\langle \texttt{invokestatic m}(\mathtt{arg}_1, \cdots, \mathtt{arg}_\mathtt{i}), (s :: v_1 :: \cdots :: v_i, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h}')} \ (\texttt{invokestatic})$$

$$\frac{fresh(\mathtt{T}, \mathsf{h}) = (r, \mathsf{h}')}{\langle \texttt{new T}, (s, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: r, \mathsf{l}, \mathsf{h}')} \ (\texttt{new}) \qquad \frac{o \neq \mathtt{null}}{\langle \texttt{getfield f}, (s :: o, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s :: \mathsf{h}(o)(\mathtt{f}), \mathsf{l}, \mathsf{h})} \ (\texttt{getfield})$$

$$\frac{o \neq \mathtt{null} \quad s' = \mathsf{h}(o)[\mathtt{f} \mapsto v]}{\langle \texttt{putfield f}, (s :: o :: v, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h}[o \mapsto s'])} \ (\texttt{putfield}) \qquad \frac{typeOf(v_1) = \mathsf{Int} \wedge typeOf(v_2) = \mathsf{Int} \wedge v_1 > v_2}{\langle \texttt{if\_icmpgt l}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} \langle \mathtt{l}, (s, \mathsf{l}, \mathsf{h})\rangle} \ \begin{pmatrix}\texttt{if\_icmpgt} \\ \texttt{true}\end{pmatrix}$$

$$\frac{typeOf(v_1) = \mathsf{Int} \wedge typeOf(v_2) = \mathsf{Int} \wedge v_1 \leq v_2}{\langle \texttt{if\_icmpgt l}, (s :: v_1 :: v_2, \mathsf{l}, \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (s, \mathsf{l}, \mathsf{h})} \ \begin{pmatrix}\texttt{if\_icmpgt} \\ \texttt{false}\end{pmatrix}$$

Fig. 6. Concrete JB semantics.

execute is that at `l`. Otherwise, the next instruction to execute is implicitly assumed to be the subsequent one, sequentially (if any).

For the most part, the concrete semantics is a straightforward formalization of the run-time semantics defined by the CIL ECMA Standard [9]. For instance, rule `add` pops the two topmost values of the operand stack and replaces them with their addition. However, its semantics is defined iff the two values have the same type. Instead, `ldloc i` pushes to the operand stack the value of the i-th local variables, while `stloc i` stores the top of the operand stack into the i-th local variable. Statements working with objects, such as `ldfld` and `stfld`, read from and write into the heap, if their receiver is not `null`. Rules `call` and `call static` create a frame (*i.e.*, an array of arguments, an empty array of local variables and an empty operand stack), execute the callee and leave its returned value on the stack, if any. For simplicity, the formalization assumes that there is no returned value.[5] Finally, `ldloca i` loads the reference to the i-th local variable to the stack (represented by r_i), `stind` stores the given value to the given reference, and `ldind` loads the value pointed by the given reference.

3.2.1. Running example

Consider the running example in Fig. 2b and apply the concrete semantics when `n` is 1, that is, when the entry state consists of an empty operand stack and of an array of local variables, while the value of the arguments is $[0 \mapsto 1]$. Assume that `newobj` allocates the object at address #1. Then after the first block (lines 3-7) the address of the object is stored in local variable 0, local variable 1 (representing variable `i` of the source program) holds 0, and address #1 in the heap holds an object of type `List⟨Collection⟩`. Formally, the concrete state at line 7 is $(\emptyset, [0 \mapsto \#1, 1 \mapsto 0], [0 \mapsto 1], [\#1 \mapsto <>])$, where $<>$ stands for the empty list. Then the body of the `for` loop (lines 9-16) is executed once and creates a new `Wrap` object (assume at address #2) wrapping 0, adds it to the list at address #1 and increments counter `i` (*i.e.*, local variable 0) by 1. Hence the execution of the concrete semantics of the body of the loop leads to the concrete state $\sigma = (\emptyset, [0 \mapsto \#1, 1 \mapsto 1], [0 \mapsto 1], [\#1 \mapsto < \#2 >, \#2 \mapsto [f \mapsto 0]])$ at line 16. The condition at line 20 will then route the program to line 22 (since argument 0 and local variable 1 hold 1). In conclusion, the concrete semantics will reach statement `ret` at line 23 with a state σ_{CIL} equal to σ, but where the operand stack contains reference #1.

3.3. JB

We define the concrete semantics of the JB fragment of Section 2.1.

Concrete State. A local state in JB consists of a stack of values $\mathsf{Stack} : \mathbb{N} \rightarrow \mathsf{Val}$ and an array of local variables $\mathsf{Loc} : \mathbb{N} \rightarrow \mathsf{Val}$. A concrete JB state consists of a local state and a heap, that is, $\Sigma_{\mathsf{JB}} = \mathsf{Stack} \times \mathsf{Loc} \times \mathsf{Heap}$.

[5] This would have required to define two distinct rules (the one with returned values where the final stack contains the returned value followed by the previous stack in addition to the one already formalized). However, the implementation fully supports this case as well, and we omit it here only to improve the readability of the formalization and formal proofs.

Concrete Semantics. Fig. 6 reports the concrete JB semantics $\rightarrow_{\mathsf{JB}}$. For a statement `st` and an entry state σ, it yields the state σ' resulting from the execution of `st` over σ. For the most part, the behavior of this semantics is identical to that for CIL. The main differences are that JB instructions work on specific types (*e.g.*, while CIL `add` statement adds two values of the same type, JB `iadd` and `ladd` statements add the values iff they are both `int` or `long`, respectively), and `new` only allocates a new object, while CIL `newobj` statements also calls a constructor.

3.3.1. Running example

The application of the JB concrete semantics is similar to that for CIL, but there are two minor differences: (i) there is only one array of local variables representing both CIL arguments and local variables (*e.g.*, CIL local variable 0 is represented by JB local variable 1, since the first local variable holds the argument of the method); and (ii) there is an *instrumentation* local variable[6] at index 3. Therefore, after we apply the JB concrete semantics from the entry state that maps the argument to 1, we obtain the concrete state $([0 \mapsto \#1], [0 \mapsto 1, 1 \mapsto \#1, 2 \mapsto 1, 3 \mapsto 0], [\#1 \mapsto <\#2>, \#2 \mapsto [f \mapsto 0]])$.

4. From CIL to JB

4.1. Concrete states

Function $\mathbb{T}_\sigma[\![\,]\!] : \Sigma_{\mathsf{CIL}} \rightarrow \Sigma_{\mathsf{JB}}$ translates CIL concrete states into JB concrete states (where l represents the number of elements in the stack of s): $\mathbb{T}_\sigma[\![(\mathsf{s}, \mathsf{l}, \mathsf{a}, \mathsf{h})]\!] = (\mathsf{s}', \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}'_l)$.

This function (i) replaces direct references with wrapper objects, and (ii) merges the array of local variables and arguments, adjusting variable indexes for 64 bits values. Formally:

$$i \in \mathit{dom}(\mathsf{s}), l = \mathit{height}(\mathsf{s}), \mathsf{s}' = \left[i \mapsto \begin{cases} \mathsf{s}(i) & \text{if } \mathsf{s}(i) \in \mathsf{Val} \\ r_i & \text{if } \mathsf{s}(i) \in \mathsf{Ref}_{\mathsf{Loc}} \end{cases} \right]$$

$$\text{where } \mathsf{h}'_{-1} = \mathsf{h}, \text{ and } (\mathsf{h}'_i, \mathsf{r}_i) = \mathit{allocWrp}(\mathsf{h}'_{i-1}, i), \mathit{allocWrp}(\mathsf{h}, j) = \begin{cases} (\mathsf{h}, \texttt{null}) \text{ if } \mathsf{s}(j) \in \mathsf{Val} \\ (\mathsf{h}'[r \mapsto \mathsf{h}(r)[\texttt{value} \mapsto \mathsf{l}(j)]]) \\ \quad \text{where } (r, \mathsf{h}') = \mathit{fresh}(\mathsf{WRef}, \mathsf{h}) \end{cases} \text{if } \mathsf{s}(i) \in \mathsf{Ref}_{\mathsf{Loc}}$$

Intuitively, each direct reference in the operand stack (that is, $\mathsf{s}(i) \in \mathsf{Ref}_{\mathsf{Loc}}$) is replaced by another reference pointing to a wrapper object freshly allocated and containing in its field the value pointed by the original direct reference.

Then, for an array of values b and an index i, the following function counts the 64 bits types among the first i:

$$64^i_b = |\{j : 0 \leq j < i \text{ and } b[j] \text{ is a 64 bit value}\}|$$

Then the array $\mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a})$ is defined as follows:

$$\forall 0 \leq i < |\mathsf{a}| : \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a})[i + 64^i_{\mathsf{a}}] = \mathsf{a}[i]$$

$$\forall 0 \leq i < |\mathsf{l}| : \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a})[|\mathsf{a}| + 64^{|\mathsf{a}|}_{\mathsf{a}} + i + 64^i_{\mathsf{l}}] = \mathsf{l}[i]$$

Definition 1. Consider a function f that maps elements of a pair of arrays (a, b) into elements of an array c. Let $f(a[i]) = c[i']$ and $f(b[i]) = c[i'']$. The function f is an *identity embedding* if the following conditions hold:

1) $\forall 0 \leq i < |a| : a[i] = c[i']$ and $\forall j < |b| : b[j] = c[j'']$
2) $\forall 0 \leq i, j < |a| : i \leq j \Rightarrow i' \leq j'$ and $\forall 0 \leq h, k < |b| : h \leq k \Rightarrow h'' \leq k''$
3) $\{i' : 0 \leq i \leq |a|\} \cap \{j'' : 0 \leq j \leq |b|\} = \emptyset$

Lemma 1. *The function cnvrtLoc is an identity embedding.*

Proof. It is sufficient to observe that, by construction, the function concatenates the two arrays by shifting indexes when a 64 bits value occurs, hence preserving the values and the ordering of the elements' indexes. □

4.1.1. Running example

Consider the CIL exit state computed in Section 3.2, that is, $\sigma_{\mathsf{CIL}} = ([0 \mapsto \#1], [0 \mapsto \#1, 1 \mapsto 1], [0 \mapsto 1], [\#1 \mapsto <\#2>, \#2 \mapsto [f \mapsto 0]])$. The CIL to JB translation computes $\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = ([0 \mapsto \#1], [0 \mapsto 1, 1 \mapsto \#1, 2 \mapsto 1], [\#1 \mapsto <\#2>, \#2 \mapsto [f \mapsto 0]])$ (it merges CIL arguments and local variables). This state is almost identical to the exit state of the JB concrete semantics applied to the running example (Section 3.3) except for the instrumentation local variable at index 3.

[6] Section 4.2 will introduce instrumentation variables and why they are needed.

$$\mathbb{T}[\![\texttt{dup}, \overline{\texttt{s}} :: \texttt{t}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \begin{cases} \texttt{dup} & \text{if } t \neq \text{Long} \\ \texttt{dup2} & \text{if } t = \text{Long} \end{cases}$$

$$\mathbb{T}[\![\texttt{add}, \overline{\texttt{s}} :: \texttt{t}_1 :: \texttt{t}_2, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \begin{cases} \texttt{iadd} & \text{if } t_1 = t_2 = \text{Int} \\ \texttt{ladd} & \text{if } t_1 = t_2 = \text{Long} \end{cases}$$

$$\mathbb{T}[\![\texttt{ldloc i}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{xload}\ j \text{ where } j = |\overline{\texttt{a}}| + 64^{|\overline{\texttt{a}}|}_{\overline{\texttt{a}}} + i + 64^{i}_{\overline{\texttt{l}}} \wedge x = \mathit{JVMprefix}(\mathit{typeOf}(\overline{\texttt{l}}(\texttt{i})))$$

$$\mathbb{T}[\![\texttt{stloc i}, \overline{\texttt{s}} :: \texttt{t}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{xstore}\ j \text{ where } j = |\overline{\texttt{a}}| + 64^{|\overline{\texttt{a}}|}_{\overline{\texttt{a}}} + i + 64^{i}_{\overline{\texttt{l}}} \wedge x = \mathit{JVMprefix}(\mathit{typeOf}(\overline{\texttt{l}}(\texttt{i})))$$

$$\mathbb{T}[\![\texttt{ldarg i}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{xload}\ j \text{ where } j = i + 64^{i}_{a} \wedge x = \mathit{JVMprefix}(\mathit{typeOf}(\overline{\texttt{a}}(\texttt{i})))$$

$$\mathbb{T}[\![\texttt{call m}(\texttt{arg}_1, \cdots, \texttt{arg}_\texttt{i}), \overline{\texttt{s}} :: t_1 :: \cdots :: t_i, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}} :: p_1 :: \cdots :: p_i]\!] =$$
$$\texttt{invoke ; aload}\ p^1_{idx_1} \texttt{ ; getfield value ; } x_{idx_1}\texttt{store}\ p^2_{idx_1} \texttt{ ; } \cdots$$
$$\cdots\texttt{aload}\ p^1_{idx_j} \texttt{ ; getfield value ; } x_{idx_j}\texttt{store}\ p^2_{idx_j} \texttt{ ;}$$
$$\text{where } \texttt{invoke} = \begin{cases} \texttt{invokestatic m}(\texttt{arg}_1, \cdots, \texttt{arg}_\texttt{i}) & \text{if } \mathit{isStatic}(\texttt{m}(\texttt{arg}_1, \cdots, \texttt{arg}_\texttt{i})) \\ \texttt{invokevirtual m}(\texttt{arg}_1, \cdots, \texttt{arg}_\texttt{i}) & \text{otherwise} \end{cases}$$
$$\{idx_1, \cdots, idx_j\} = \{k : \texttt{arg}_\texttt{k} \in \mathsf{Ref}_{\mathsf{Loc}}\}$$
$$\forall k \in [1..j] : x_{idx_k} = \mathit{JVMprefix}(\mathit{typeOf}(\overline{\texttt{l}}(p^2_{idx_k}))) \wedge \forall r \in [1..i] : p_i = (p^1_i, p^2_j)$$

$$\mathbb{T}[\![\texttt{newobj T}(\texttt{a}_1, \cdots, \texttt{a}_\texttt{i}), \overline{\texttt{s}} :: t_1 :: \cdots :: t_i, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] =$$
$$x_i\texttt{store}\ idx_i \texttt{ ; } \cdots \texttt{ ; } x_1\texttt{store}\ idx_1 \texttt{ ; new T ; dup ;}$$
$$x_1\texttt{load}\ idx_1 \texttt{ ; } \cdots \texttt{ ; } x_i\texttt{load}\ idx_i \texttt{ ; invokevirtual } <\texttt{init}>(\texttt{arg}_1, \cdots, \texttt{arg}_\texttt{i})$$
$$\text{where } \forall j \in [1..i] : x_j = \mathit{JVMprefix}(a_j) \wedge idx_j = \mathit{freshIdx}(\texttt{newobj T}(\texttt{a}_1, \cdots, \texttt{a}_\texttt{i}), j)$$

$$\mathbb{T}[\![\texttt{ldfld f}, \overline{\texttt{s}} :: \texttt{t}_\texttt{o}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{getfield f}$$

$$\mathbb{T}[\![\texttt{stfld f}, \overline{\texttt{s}} :: \texttt{t}_\texttt{o} :: \texttt{t}_\texttt{v}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{putfield f}$$

$$\mathbb{T}[\![\texttt{bgt k}, \overline{\texttt{s}} :: \texttt{t}_1 :: \texttt{t}_2, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{if\_icmpgt k}' \text{ where } \texttt{k}' = \mathit{statementIdx}(\mathit{getBody}(\texttt{bgt k})(\texttt{k})) \text{ if } t_1 = t_2 = \text{Int}$$

$$\mathbb{T}[\![\texttt{ldloca i}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \mathbb{T}[\![\texttt{newobj WrapRef() ; dup2 ; stloc j; ldloc i; stfld value}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!]$$
$$\text{where } j = \mathit{freshIdx}(\texttt{ldloca i}, 0)$$

$$\mathbb{T}[\![\texttt{stind}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \mathbb{T}[\![\texttt{stfld value}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!]$$

$$\mathbb{T}[\![\texttt{ldind}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \mathbb{T}[\![\texttt{ldfld value}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!]$$

Fig. 7. Translation of CIL statements into JB.

4.2. Statements

Fig. 7 formalizes the translation $\mathbb{T}[\![\texttt{st}_{\mathsf{CIL}}, \texttt{K}]\!] = \texttt{st}_{\mathsf{JB}}$ of a single CIL statement into a sequence of JB statements. The components with a linear accent (e.g., $\overline{\texttt{s}}$) denotes static information about computational values. In particular, our translation relies on static type information about locals (denoted by $\overline{\texttt{l}}$), arguments ($\overline{\texttt{a}}$) and stack elements ($\overline{\texttt{s}}$), computed at $\texttt{st}_{\mathsf{CIL}}$ by a standard algorithm [9]. In particular, types and height of the stack are fixed and known for each bytecode. In addition, the fourth component $\overline{\texttt{w}}$ is a stack of element in $\bot \cup (\mathbb{N} \times \mathbb{N})$ that, for each element in the stack, tells (i) $\bot$ if it is not a direct reference, or (ii) (i, j) where i is the index of the local variables pointed by the direct reference,[7] and j is the index of the local instrumentation variable containing a pointer to the wrapper simulating the reference. Instrumentation local variables are appended at the end of the array of JB local variables (that is, after the arguments and the local variables used in the program). They are needed in order to cache some values that are later needed to represent the semantics of the CIL program in its JB translation.

Few CIL statements (namely, `ldfld` and `stfld`) have a one-to-one translation into a JB statement (`getfield` and `putfield`). The statements reading and writing local variables and arguments (`ldarg`, `ldloc`, and `stloc`) are translated into their JB counterpart (`xload`, `xstore`, respectively), taking into account the type of the value at the top of the stack, and adjusting the index of the variable, taking into account arguments and 64 bit variables. Some CIL statements (`dup`) get translated into different JB statements on the basis of contextual information such as the type of values in the operand stack (`dup` and `dup2`). Other CIL statements can be translated only if the type of the values in the operand stack is numeric: (i) `add` can be translated into `ladd` and `iadd`, and (ii) `bgt` to `is_icmpgt`; if they are applied to references (as in generic CIL code), then the code is considered unsafe and is not translated.

`call` requires to (i) translate the method call to the corresponding static or dynamic invocation statement in JB, and (ii) to propagate the side effects on direct pointers passed to the method as `out`/`ref` parameters to the local variables of the callee. The translation of `newobj` is tricky because of the different patterns used in CIL and JB for object creation.[8] While CIL creates and initializes the object (*i.e.*, calls its constructor) with a single instruction, JB splits these operations and requires the newly created object to occur below the arguments on the stack, before calling the constructor. Hence, the translation relies on a function *freshIdx* to store and load the values of the constructor arguments through instrumentation local variables. In particular, given a CIL method m, the number and types of arguments and local variables of the method are known (Section II.15.4 of [9]). Therefore, function *cnvrtLoc* tells which local variables the translated JB method already uses. Then, for each argument of each `newobj` statement in m, it is possible to allocate a fresh local variable to store and load its value. In this way, the translation allocates a new object and puts its address below the constructor arguments.

[7] Since the language we introduced in Fig. 1 supports only `ldloca` to get a direct pointer, we need to track only this information in the formalization.

[8] For sake of simplicity, we assume the constructor does not have `out`/`ref` parameters. In the implementation, they are treated as for the `call` statements.

Instructions dealing with direct pointers (namely, `ldloca`, `stind`, and `ldind` in our minimal language) are translated through equivalent CIL instructions dealing with wrapper objects (and their field `value`). Therefore, `stind` and `ldind` are simply translated through equivalent write and read of field `value`, respectively. `ldloca` instead requires to allocate a wrapper object `newobj`, stores a reference to the wrapper (`stloc`) in an instrumentation variable obtained through *freshIdx*, stores the value pointed by the direct reference in the local variable to its field `value` (`ldloc` and `stfld`), and leaves a reference to the wrapper in the stack (`dup`).

Each CIL statement is translated into one or more JB statements, hence offsets are not preserved. Thus, function *statementIdx* : $\mathsf{St} \to \mathbb{N}$ yields the JB offset of the first statement in the translation of the given CIL statement. In addition, since direct references are replaced by wrapper objects, when a method parameter has a direct reference type `&T` (and this happens when it is a `ref` or `out` parameter in safe C#), this is replaced by a wrapper object WRef.

4.2.1. Running example

Consider the running example in Fig. 2. Most CIL statements are translated into a single JB statement (*e.g.*, lines 18-20 and 22-23 of Fig. 2b are translated into lines 23-25 and 27-28 of Fig. 2c), with the noticeable exception of the CIL `newobj` statements at lines 3 and 11, translated into lines 2-4 and 12-16, respectively. The former passes no argument to the constructor; the latter (that instantiates a `Wrap`) calls a constructor with an argument, hence requiring an instrumentation variable at index 3.

4.2.2. Direct references

As sketched in Section 2.2.1, we model the semantics of pointers in safe C# code through wrapper objects. In particular, `ldind` (line 5 of Fig. 3c) is translated into the field access `i.value` (right side of the assignment at line 2 of Fig. 3b), while `stind` (line 8 of Fig. 3c) is translated into the assignment of `i.value` (left side of the assignment at line 2 of Fig. 3b). In addition, `ldloca` (line 15 of Fig. 3c) leads to the construction and assignment of a wrapper object (lines 7 and 8 of Fig. 3b), while after the method call the value contained in the wrapper object is written into the local variable (line 10 of Fig. 3b).

4.3. Correctness

This section proves that the translation from CIL to JB statements is correct. Namely, given a concrete CIL state σ_{CIL} and applying the operational semantics for a statement `st`, one obtains a state that, when translated into JB, is exactly the state resulting from the translation of σ_{CIL} into JB and the application of the JB semantics to it:

$$\forall \mathtt{st} \in \mathsf{St}_{\mathsf{CIL}}, \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}} : \langle \mathtt{st}, \sigma_{\mathsf{CIL}} \rangle \to_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}} \text{ and } \langle \mathbb{T}[\![\mathtt{st}, \mathtt{K}]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \to_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$$
$$\Downarrow$$
$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$$

where $\sigma_1 =_\bullet \sigma_2$ means that the two states are equal up to instrumentation variables introduced by the translation process. Formally, let $\sigma_1 = (\mathtt{s}_1, \mathtt{l}_1^1 :: \ldots :: \mathtt{l}_1^n, \mathtt{h}_1)$ and $\sigma_2 = (\mathtt{s}_2, \mathtt{l}_2^1 :: \ldots :: \mathtt{l}_2^n, :: \mathtt{l}_2^{n+1} :: \ldots :: \mathtt{l}_2^{n+k}, \mathtt{h}_2)$, then $\sigma_1 =_\bullet \sigma_2$ iff $\mathtt{s}_1 = \mathtt{s}_2$, and $\forall \mathtt{i} \leq \mathtt{n} : \mathtt{l}_1^i = \mathtt{l}_2^i$, and $\mathtt{h}_1 = \mathtt{h}_2$. Note that instrumentation variables are present only in the JB state, hence in the right hand-side of the equality.

4.3.1. Formal proof of soundness

Lemma 2. *The translation of* `ldloc` *is correct.*

Proof. Let us consider the case of integer arguments (the other case can be treated analogously). Let us prove that the translation of `ldloc` for integer values is correct, *i.e.*, that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \mathtt{ldloc\ i}, \sigma_{\mathsf{CIL}} \rangle \to_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\mathtt{ldloc\ i}, (\overline{\mathtt{s}}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \to_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s, l, a, h)$ be arbitrary. By the `ldloc`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\mathtt{s} :: \mathtt{l[i]}, \mathtt{l}, \mathtt{a}, \mathtt{h})$. By rule (3) of Fig. 7, $\mathbb{T}[\![\mathtt{ldloc\ i}, \overline{\mathtt{s}}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}}]\!] = \mathtt{iload}\ j$ where $j = |\overline{\mathtt{a}}| + 64^{|\overline{\mathtt{a}}|}_{\overline{\mathtt{a}}} + i + 64^{i}_{\overline{\mathtt{l}}}$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s}, \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s}', \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}').$$

By the `iload`−JB rule:

$$\langle \mathtt{iload\ j}, (\mathtt{s}', \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') \rangle \to_{\mathsf{JB}} (\mathtt{s}' :: \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a})[\mathtt{j}], \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\ ]\!]$ we have

$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{l[i]}, \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s} :: \mathtt{l[i]}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}).$$

By definition of *cnvrtLoc* we have $\mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a})[\mathtt{j}] = \mathtt{l[i]}$, which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 3. *The translation of* `ldarg` *is correct.*

Proof. Let us consider the case of integer arguments (the other case can be treated analogously). Let us prove that the translation of `ldarg` for integer values is correct, *i.e.*, that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \texttt{ldarg i}, \sigma_{\mathsf{CIL}} \rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\texttt{ldarg i}, (\overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s, l, a, h)$ be arbitrary. By the `ldarg`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\texttt{s} :: \texttt{a[i]}, \texttt{l}, \texttt{a}, \texttt{h})$. By rule (5) of Fig. 7, $\mathbb{T}[\![\texttt{ldarg i}, \overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{iload } j$ where $j = i + 64^{|\overline{\texttt{a}}|}_{\overline{\texttt{a}}} + i + 64^{i}_{\overline{\texttt{l}}}$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\texttt{s}, \texttt{l}, \texttt{a}, \texttt{h})]\!] = (\texttt{s}', \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}').$$

By the `iload`−JB rule:

$$\langle \texttt{iload j}, (\texttt{s}', \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}') \rangle \rightarrow_{\mathsf{JB}} (\texttt{s}' :: \mathit{cnvrtLoc}(\texttt{l}, \texttt{a})[\texttt{j}], \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}') = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\,]\!]$ we have

$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\texttt{s} :: \texttt{l[i]}, \texttt{l}, \texttt{a}, \texttt{h})]\!] = (\texttt{s} :: \texttt{l[i]}, \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}).$$

By definition of *cnvrtLoc* we have $\mathit{cnvrtLoc}(\texttt{l}, \texttt{a})[\texttt{j}] = \texttt{l[i]}$, which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 4. *The translation of* `stloc` *is correct.*

Proof. Let us consider the case of integer arguments (the other case can be treated analogously). Let us prove that the translation of `stloc` for integer values is correct, *i.e.*, that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \texttt{stloc i}, \sigma_{\mathsf{CIL}} \rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\texttt{stloc i}, (\overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s :: v, l, a, h)$ be an arbitrary state where the stack has an integer value v at the top. By the `stloc`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\texttt{s}, \texttt{l}[\texttt{i} \mapsto \texttt{v}], \texttt{a}, \texttt{h})$. By rule (4) of Fig. 7, $\mathbb{T}[\![\texttt{stloc i}, \overline{\texttt{s}} :: \texttt{t}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{istore } j$ where $j = |\overline{\texttt{a}}| + 64^{|\overline{\texttt{a}}|}_{\overline{\texttt{a}}} + i + 64^{i}_{\overline{\texttt{l}}}$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\texttt{s} :: \texttt{v}, \texttt{l}, \texttt{a}, \texttt{h})]\!] = (\texttt{s}' :: v, \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}')$$

By the `istore`−JB rule:

$$\langle \texttt{istore j}, (\texttt{s}' :: v, \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}') \rangle \rightarrow_{\mathsf{JB}} (\texttt{s}', \mathit{cnvrtLoc}(\texttt{l}, \texttt{a})[\texttt{j} \mapsto \texttt{v}], \texttt{h}') = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\,]\!]$ we have

$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\texttt{s}, \texttt{l}[\texttt{i} \mapsto \texttt{v}], \texttt{a}, \texttt{h})]\!] = (\texttt{s}, \mathit{cnvrtLoc}(\texttt{l}[\texttt{i} \mapsto \texttt{v}], \texttt{a}), \texttt{h}')$$

By definition of *cnvrtLoc* we have that $\mathit{cnvrtLoc}(\texttt{l}[\texttt{i} \mapsto \texttt{v}], \texttt{a}) = \mathit{cnvrtLoc}(\texttt{l}, \texttt{a})[\texttt{j} \mapsto \texttt{v}]$ which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 5. *The translation of* `add` *is correct.*

Proof. Let us consider the case of integer arguments (the other case can be treated analogously). Let us prove that the translation of `add` for integer values is correct, *i.e.*, that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \texttt{add}, \sigma_{\mathsf{CIL}} \rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\texttt{add}, (\overline{\texttt{s}}, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s :: v_1 :: v_2, l, a, h)$ be an arbitrary state where the stack has two integer values v_1 and v_2 at the top. By the `add`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\texttt{s} :: (\texttt{v}_1 + \texttt{v}_2), \texttt{l}, \texttt{a}, \texttt{h})$. By rule (2) of Fig. 7, $\mathbb{T}[\![\texttt{stloc i}, \overline{\texttt{s}} :: \texttt{t}_1 :: \texttt{t}_2, \overline{\texttt{l}}, \overline{\texttt{a}}, \overline{\texttt{w}}]\!] = \texttt{iadd } j$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\texttt{s} :: \texttt{v}_1 :: \texttt{v}_2, \texttt{l}, \texttt{a}, \texttt{h})]\!] = (\texttt{s}' :: v_1 :: v_2, \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}')$$

By the `iadd`−JB rule:

$$\langle \texttt{iadd j}, (\texttt{s}' :: v_1 :: v_2, \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}') \rangle \rightarrow_{\mathsf{JB}} (\texttt{s}' :: (\texttt{v}_1 + \texttt{v}_2), \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}') = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\,]\!]$ we have

$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\texttt{s} :: (\texttt{v}_1 + \texttt{v}_2), \texttt{l}, \texttt{a}, \texttt{h})]\!] = (\texttt{s}' :: (\texttt{v}_1 + \texttt{v}_2), \mathit{cnvrtLoc}(\texttt{l}, \texttt{a}), \texttt{h}')$$

which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 6. *The translation of* `dup` *is correct.*

Proof. Let us consider the case of integer arguments (the other case can be treated analogously). Let us prove that the translation of `dup` for integer values is correct, *i.e.*, that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \mathtt{dup}, \sigma_{\mathsf{CIL}} \rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\mathtt{add}, (\overline{\mathtt{s}}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s :: v, l, a, h)$ be an arbitrary state where the stack has an integer value v at the top. By the `dup`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\mathtt{s} :: \mathtt{v} :: \mathtt{v}, \mathtt{l}, \mathtt{a}, \mathtt{h})$. By rule (1) of Fig. 7, $\mathbb{T}[\![\mathtt{dup}, \overline{\mathtt{s}} :: \mathtt{t}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}}]\!] = \mathtt{dup}\ j$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{v}, \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s}' :: v, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}').$$

By the `dup`−JB rule:

$$\langle \mathtt{dup}, (\mathtt{s}' :: v, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') \rangle \rightarrow_{\mathsf{JB}} (\mathtt{s}' :: \mathtt{v} :: \mathtt{v}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\,]\!]$ we have

$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{v}, \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s}' :: \mathtt{v}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}')$$

which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 7. *The translation of* `ldfld` *is correct.*

Proof. Let us consider the case of a numerical field (the case of a reference field can be treated analogously). We want to prove that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \mathtt{ldfld\ f}, \sigma_{\mathsf{CIL}} \rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\mathtt{ldfld\ f}, (\overline{\mathtt{s}}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s :: o, l, a, h)$ be an arbitrary where the stack contains a reference o at the top. By the `ldfld`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\mathtt{s} :: \mathtt{h}(\mathtt{o})(\mathtt{f}), \mathtt{l}, \mathtt{a}, \mathtt{h})$. By rule (8) of Fig. 7, $\mathbb{T}[\![\mathtt{ldfld\ f}, \overline{\mathtt{s}} :: \mathtt{t}_\mathtt{o}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}}]\!] = \mathtt{getfield\ f}$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{o}, \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s}' :: \mathtt{r}_\mathtt{o}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}')$$

where r_o is defined as a reference to a wrapper JB object representing the CIL object. By the `getfield`−JB rule:

$$\langle \mathtt{getfield\ f}, (\mathtt{s}' :: \mathtt{r}_\mathtt{o}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') \rangle \rightarrow_{\mathsf{JB}} (\mathtt{s}' :: \mathtt{h}'(\mathtt{r}_\mathtt{o})(\mathtt{f}), \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\,]\!]$ we have

$$\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{h}(\mathtt{o})(\mathtt{f}), \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s}' :: \mathtt{h}(\mathtt{r}_\mathtt{o})(\mathtt{f}), \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}')$$

which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 8. *The translation of* `stfld` *is correct.*

Proof. Let us consider the case of a numerical field (the case of a reference field can be treated analogously). We want to prove that $\forall \sigma_{\mathsf{CIL}} \in \Sigma_{\mathsf{CIL}}$, if $\langle \mathtt{stfld\ f}, \sigma_{\mathsf{CIL}} \rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ and $\langle \mathbb{T}[\![\mathtt{stfld\ f}, (\overline{\mathtt{s}}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] \rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$ then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. Let $\sigma_{\mathsf{CIL}} = (s :: o :: v, l, a, h)$ be an arbitrary where the stack contains a reference o and a value v at the top. By the `stfld`−CIL rule we have $\sigma'_{\mathsf{CIL}} = (\mathtt{s}, \mathtt{l}, \mathtt{a}, \mathtt{h}[\mathtt{o} \mapsto \mathtt{h}(\mathtt{o})[\mathtt{f} \mapsto \mathtt{v}]])$. By rule (9) of Fig. 7, $\mathbb{T}[\![\mathtt{stfld\ f}, \overline{\mathtt{s}} :: \mathtt{t}_\mathtt{o} :: \mathtt{t}_\mathtt{v}, \overline{\mathtt{l}}, \overline{\mathtt{a}}, \overline{\mathtt{w}}]\!] = \mathtt{putfield\ f}$. By definition of $\mathbb{T}_\sigma[\![\,]\!]$:

$$\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = \mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{o} :: \mathtt{v}, \mathtt{l}, \mathtt{a}, \mathtt{h})]\!] = (\mathtt{s}' :: \mathtt{r}_\mathtt{o} :: \mathtt{v}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}')$$

where r_o is defined as a reference to a wrapper JB object representing the CIL object. By the `putfield`−JB rule:

$$\langle \mathtt{putfield\ f}, (\mathtt{s}' :: \mathtt{r}_\mathtt{o} :: \mathtt{v}, \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}') \rangle \rightarrow_{\mathsf{JB}} (\mathtt{s}' :: \mathtt{h}'(\mathtt{r}_\mathtt{o})(\mathtt{f}), \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}'[\mathtt{r}_\mathtt{o} \mapsto \mathtt{h}'(\mathtt{r}_\mathtt{o})[\mathtt{f} \mapsto \mathtt{v}]]) = \sigma'_{\mathsf{JB}}.$$

By definition of $\mathbb{T}_\sigma[\![\,]\!]$ we have

$$\begin{aligned} \mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] &= \mathbb{T}_\sigma[\![(\mathtt{s}, \mathtt{l}, \mathtt{a}, \mathtt{h}[\mathtt{o} \mapsto \mathtt{h}(\mathtt{o})[\mathtt{f} \mapsto \mathtt{v}]])]\!] \\ &= (\mathtt{s}', \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}'[\mathtt{r}_\mathtt{o} \mapsto \mathtt{h}(\mathtt{o})[\mathtt{f} \mapsto \mathtt{v}]]) \\ &= (\mathtt{s}', \mathit{cnvrtLoc}(\mathtt{l}, \mathtt{a}), \mathtt{h}'[\mathtt{r}_\mathtt{o} \mapsto \mathtt{h}'(\mathtt{r}_\mathtt{o})[\mathtt{f} \mapsto \mathtt{v}]]) \end{aligned}$$

which implies $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 9. *The translation of* `call` *is correct.*

Proof. Let us consider the case of static call (the other case can be treated analogously). Moreover, let us denote by $\ell_{[i,j]}$ the sequence $\ell_i, \ldots, \ell_j$, and by $\mathtt{s}_{(\mathtt{i},\mathtt{j})}$ the sequence of types $[\mathtt{k}_1, \ldots, \mathtt{k}_i, \mathtt{t}_1, \ldots, \mathtt{t}_j]$. We show that: $\forall \sigma_{\mathsf{CIL}} = ([u_{[1,n]}, v_{[1,i]}], \mathsf{l}, \mathsf{a}, \mathsf{h})$, if $\langle \mathtt{call\ m(arg}_{[1,i]}), \sigma_{\mathsf{CIL}}\rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}} = ([u_{[1,n]}], \mathsf{l}, \mathsf{a}, \mathsf{h}')$ and $\langle \mathbb{T}[\![\mathtt{call\ m(arg}_{[1,i]}), (\mathtt{s}_{(\mathtt{n},\mathtt{i})}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}_{[1,i]})]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!]\rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$, then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$.

Rule $\langle \mathtt{call\ m(arg}_{[1,i]}), \sigma_{\mathsf{CIL}}\rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}}$ requires, to be applied, that the condition

$$\langle \mathit{body}(\mathtt{m(arg}_{[0,i]}), (v_{[1,i]})), ([\,], \emptyset, [j-1 \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')$$

is satisfied.

Observe that for $\mathtt{a} = [j-1 \mapsto v_j : j \in [1..i]]$, we get $\mathbb{T}_\sigma[\![([\,], \emptyset, \mathsf{a}, \mathsf{h})]\!] = ([\,], \mathsf{l}'', \mathsf{h})$, where for each $j \in [1..i]$, $\mathsf{l}''[j + 64^j_{\mathsf{a}}] = v_j$. Moreover, $\mathbb{T}[\![\mathit{body}(\mathtt{m(arg}_{[0,i]})), (\mathtt{t}_1, \ldots, \mathtt{t}_i), \emptyset, \mathsf{a}, \mathsf{h}]\!] = \mathit{body}(\mathtt{m(arg}_{[0,i]}))$. Therefore, by inductive hypothesis, we get that $\langle \mathit{body}(\mathtt{m(arg}_{[0,i]})(v_{[1,i]})), ([\,], [j-1 \mapsto v_j : j \in [1..i]], \mathsf{h})\rangle \rightarrow_{\mathsf{JB}} (\mathsf{s}'', \mathsf{l}'', \mathsf{h}'')$ is such that $\mathbb{T}_\sigma[\![(\mathsf{s}', \mathsf{l}', \mathsf{a}', \mathsf{h}')]\!] =_\bullet (\mathsf{s}'', \mathsf{l}'', \mathsf{h}'')$, and in particular $\mathsf{h}' = \mathsf{h}''$. It is sufficient now to recall that by Fig. 7 (static call) $\mathbb{T}[\![\mathtt{call\ m(arg}_{[1,i]}), \mathtt{s}_{(\mathtt{n},\mathtt{i})}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}_{[\mathtt{n},\mathtt{i}]}]\!]$ is obtained by applying `invokestatic` $\mathtt{m(arg}_{[1,i]})$ followed by the update of all the local variables passed by reference to the called method, and that by the `invokestatic` rule of Fig. 6, $\sigma'_{\mathsf{JB}} = ([\mathtt{u}_{[1,n]}, \mathtt{l}'', \mathtt{h})$. Finally, by the definition of $\mathbb{T}_\sigma[\![\,]\!]$, we get $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 10. *The translation of* `bgt` *is correct.*

Proof. Consider $\mathbb{T}[\![\mathtt{bgt\ k}, \overline{\mathtt{s}} :: \mathtt{t}_1 :: \mathtt{t}_2, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!]$ in the case $\mathtt{t}_1 = \mathtt{t}_2 = \mathtt{int}$, and assume $\sigma_{\mathsf{CIL}} = ([s_{[1,n]} :: v_1, v_2], \mathsf{l}, \mathsf{a}, \mathsf{h})$ with $v_2 > v_2$ (the other case is similar), yielding to $\langle \mathtt{bgt\ l}, \sigma_{\mathsf{CIL}}\rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}} = \langle \mathtt{l}, \sigma_{\mathsf{CIL}}\rangle$.

We show that if $\langle \mathbb{T}[\![\mathtt{bgt\ k}, \overline{\mathtt{s}} :: \mathtt{t}_1 :: \mathtt{t}_2, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!]\rangle \rightarrow_{\mathsf{JB}} \sigma'_{\mathsf{JB}}$, then $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!] =_\bullet \sigma'_{\mathsf{JB}}$. By the corresponding rule in Fig. 7, $\mathbb{T}[\![\mathtt{bgt\ k}, \overline{\mathtt{s}} :: \mathtt{t}_1 :: \mathtt{t}_2, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!] = \mathtt{if\_icmpgt}\, k'$ where

$k' = \mathit{statementIdx}(\mathit{getBody}(\mathtt{bgt\ k})(k))$.

By the semantics of `if_icmpgt` we have that $\langle \mathtt{if\_icmpgt}\, k', ([s_{[1,n]} :: v_1 :: v_2], \mathit{convertLocals}(\mathtt{l}, \mathtt{a}), \mathtt{h}) \rightarrow_{\mathsf{JB}} \langle k', (s_{[1,n]}, \mathit{convertLocals}(\mathtt{l}, \mathtt{a}), \mathtt{h})\rangle$. As $\mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!] = ([s_{[1,n]} :: v_1 :: v_2], \mathit{convertLocals}(\mathtt{l}, \mathtt{a}), \mathtt{h})$, and by definition of *statementIdx*(), we get that $\mathbb{T}_\sigma[\![\langle \mathtt{l}, \sigma_{\mathsf{CIL}}\rangle]\!] = \langle k', \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!]\rangle = \sigma'_{\mathsf{JB}}$, and thus $\mathbb{T}_\sigma[\![\langle \mathtt{l}, \sigma_{\mathsf{CIL}}\rangle]\!] =_\bullet \sigma'_{\mathsf{JB}}$. □

Lemma 11. *The translation of* `newobj` *is correct.*

Proof. Assume $\sigma_{\mathsf{CIL}} = ([s_{[1,n]} :: v_{[1,i]}], \mathsf{l}, \mathsf{a}, \mathsf{h})$. By definition, $\langle \mathtt{newobjT(arg}_{[1,i]}), \sigma_{\mathsf{CIL}}\rangle \rightarrow_{\mathsf{CIL}} \sigma'_{\mathsf{CIL}} = (s :: r, (l, a, h'))$, where r and h' satisfy the constraints of the corresponding rule of Fig. 5. In particular, *fresh* $\mathtt{T}\ \mathsf{h} = (r, \mathsf{h}')$ allocates the memory for an object of type $\mathtt{T}$ on heap h and returns (i) the reference r of the freshly allocated object, and (ii) the heap h' resulting from the allocation of memory on h.

Let $\sigma'_{\mathsf{JB}} = \langle \mathbb{T}[\![\mathtt{newobj\ T(a}_{[1,i]}), \overline{\mathtt{s}} :: \mathtt{t}_{[1,i]}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!], \mathbb{T}_\sigma[\![\sigma_{\mathsf{CIL}}]\!]\rangle$, and compare $\mathbb{T}_\sigma[\![\sigma'_{\mathsf{CIL}}]\!]$ and σ'_{JB} componentwise. We may observe that in both cases the store is equal to $s :: r$, as the new elements added to the store during the translation in order to implement the object initialization are finally removed when applying the `invokevirtual` call, whose correctness is granted by structural inductive hypothesis. Moreover, the single array in JB for both local variables and method arguments is updated properly by storing and loading the values of the constructor arguments in the expected ordering. Finally, the heap h' results in both cases from the allocation of corresponding memory on h. □

Lemma 12. *The translation of* `stind` *is correct.*

Proof. By Fig. 7, we have that

$\mathbb{T}[\![\mathtt{stind}, \overline{\mathsf{s}}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!] = \mathbb{T}[\![\mathtt{stfld\ value}, \overline{\mathsf{s}}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!] = \mathtt{putfield\ value}$

By Fig. 5, we have that $\langle \mathtt{stind}, (s :: r_i :: v, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s, \mathsf{l}, \mathsf{a}, \mathsf{h}[r_i \mapsto v])$. By definition of the translation of concrete states (assuming that the only direct reference in the stack is r_i), we have that $\mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{r_i} :: \mathtt{v}, \mathsf{l}, \mathsf{a}, \mathsf{h})]\!] = (s :: r' :: v, \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r' \mapsto [\mathtt{value} \mapsto l(i)]])$ where r' is a freshly allocated reference pointing to a wrapper. Then by Fig. 6, we have that $\langle \mathtt{putfield\ value}, (s :: r' :: v, \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r \mapsto [\mathtt{value} \mapsto l(i)]])\rangle \rightarrow_{\mathsf{JB}} (s, \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r' \mapsto [\mathtt{value} \mapsto v]])$. Finally, we obtain that $\mathbb{T}_\sigma[\![(\mathtt{s}, \mathsf{l}, \mathsf{a}, \mathsf{h}[\mathtt{r_i} \mapsto \mathtt{v}])]\!] =_\bullet (s, \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r' \mapsto [\mathtt{value} \mapsto v]])$ proving the soundness of the translation of `stind`. □

Lemma 13. *The translation of* `ldind` *is correct.*

Proof. By Fig. 7, we have that

$\mathbb{T}[\![\mathtt{ldind}, \overline{\mathsf{s}}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!] = \mathbb{T}[\![\mathtt{ldfld\ value}, \overline{\mathsf{s}}, \overline{\mathsf{l}}, \overline{\mathsf{a}}, \overline{\mathsf{w}}]\!] = \mathtt{getfield\ value}$

By Fig. 5, we have that $\langle \mathtt{ldind}, (s :: r_i, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: \mathsf{h}(r_i), \mathsf{l}, \mathsf{a}, \mathsf{h})$. By definition of the translation of concrete states (assuming that the only direct reference in the stack is r_i), we have that $\mathbb{T}_\sigma[\![(\mathtt{s} :: \mathtt{r_i}, \mathsf{l}, \mathsf{a}, \mathsf{h})]\!] = (s :: r', \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r' \mapsto [\mathtt{value} \mapsto l(i)]])$ where r' is a freshly allocated reference pointing to a wrapper. Then by Fig. 6, we have that

```
1  if_icmpeq 4
2  iconst_0
3  goto 5
4  iconst_1
5  nop
```

(a) `ceq` on integers.

```
1  lcmp
2  iconst_0
3  if_icmpeq 6
4  iconst_0
5  goto 7
6  iconst_1
7  nop
```

(b) `ceq` on longs.

```
1  ldloc.0
2  ldc.i4.0
3  callvirt !0 List<A>::get(int32)
4  stloc.1
```

(c) Getting a list element in CIL.

```
1  aload_0
2  iconst_0
3  invoke List.get:(I)LObject;
4  checkcast A
5  astore_1
```

(d) Getting a list element in JB.

Fig. 8. Examples of numerical comparisons and generics.

$\langle$`getfield value`, $(s :: r', \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}),\ \mathsf{h}[r' \mapsto [\texttt{value} \mapsto l(i)]])\rangle \rightarrow_{\mathsf{JB}} (s :: l(i)]], \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r' \mapsto [\texttt{value} \mapsto v]])$. Finally, we obtain that $\mathbb{T}_\sigma [\![(\mathsf{s} :: \mathtt{l(i)}, \mathsf{l}, \mathsf{a}, \mathsf{h})]\!] =_\bullet (s, \mathit{cnvrtLoc}(\mathsf{l}, \mathsf{a}), \mathsf{h}[r' \mapsto [\texttt{value} \mapsto l(i)]])$ proving the soundness of the translation of `ldind`. □

Lemma 14. *The translation of* `ldloca` *is correct.*

Proof. By Fig. 5, we have that $\langle$`ldloca i`, $(s, \mathsf{l}, \mathsf{a}, \mathsf{h})\rangle \rightarrow_{\mathsf{CIL}} (s :: r_i, \mathsf{l}, \mathsf{a}, \mathsf{h})$ where r_i is the direct reference pointing to the i-th local variable. By Fig. 7, we have that `ldloca i` is translated into a sequence of statements that (i) creates a wrapper object containing the value of the i-th local variable, (ii) stores its reference into an instrumentation variable, and (iii) leaves its reference on the operand stack as well. Then, by definition of the concrete semantics of JB, we obtain a final state σ'_{JB} appending to the initial stack a reference to the wrapper object whose value is the one of the i-th local variable. Therefore, $\sigma'_{\mathsf{JB}} =_\bullet \mathbb{T}_\sigma [\![(\mathsf{s} :: \mathtt{r_i}, \mathsf{l}, \mathsf{a}, \mathsf{h})]\!]$ since $=_\bullet$ ignores the instrumentation variables. This proves the soundness of the translation of `ldloca`. □

Theorem 1. *The translation of the presented set of* CIL *instructions into* JB *code is correct.*

Proof. We prove that the translation of each statement from CIL to JB depicted in Fig. 7 satisfies the correctness property introduced in Section 4.3. In fact, Lemma 2, Lemma 3, Lemma 4, Lemma 5, Lemma 6, Lemma 7, and Lemma 8 prove the correctness of the translation of `ldloc`, `ldlarg`, `stloc`, `add`, `dup`, `ldfld`, and `stfld`, respectively. The correctness of a new object creation is proved by Lemma 11. The correctness proof of static and dynamic calls translation has been given in Lemma 9, and that of `stind`, `ldind`, `ldloca` was proved by Lemmas 12, 13 and 14, respectively. Finally, the correctness of the comparison statements translation is shown in Lemma 10. □

4.3.2. Running example

Section 3.2 showed that, starting from $\sigma_{\mathsf{CIL}} = (\emptyset, \emptyset, [0 \mapsto 1], \emptyset)$, the concrete semantics on the program in Fig. 2b ends up in $\sigma'_{\mathsf{CIL}} = ([0 \mapsto \#1], [0 \mapsto \#1, 1 \mapsto 1], [0 \mapsto 1], [\#1 \mapsto <\#2>, \#2 \mapsto [f \mapsto 0]])$. Then Section 3.3 showed that, starting from the corresponding $\mathbb{T}_\sigma [\![\sigma_{\mathsf{CIL}}]\!]$ state, the JB concrete semantics leads to $\sigma'_{\mathsf{JB}} = ([0 \mapsto \#1], [0 \mapsto 1, 1 \mapsto \#1, 2 \mapsto 1, 3 \mapsto 0], [\#1 \mapsto <\#2>, \#2 \mapsto [f \mapsto 0]]$. Hence, by definition of $\mathbb{T}_\sigma [\![\ ]\!]$, we have $\mathbb{T}_\sigma [\![\sigma'_{\mathsf{CIL}}]\!] = \bullet\sigma'_{\mathsf{JB}}$ since the two stacks and the two heaps are equal, the values of three local variables of the JB state correspond to the values of the argument and the two local variables of the CIL state, respectively, and $=_\bullet$ projects away the fourth variable of the JB state σ'.

4.4. Other instructions

In this section, we informally discuss how our approach deals with CIL instructions that are slightly different from other instructions in JB. We decided to handle these instructions informally since their translation is mostly straightforward. However, our implementation, whose experimental results will be discussed in Section 5, fully supports them. It is intended for readers that are expert of JB, CIL and more advanced C# features, such as generic type erasure in JB, or delegates in C#.

Numerical and reference comparison. CIL compares numerical or reference values in two ways: through conditional branches (*e.g.*, `beq` branches when the topmost two values on the stack are equal) and comparisons (*e.g.*, `ceq` pushes 1 iff the topmost two values on the stack are equal, and 0 otherwise). As usual, these instructions are type-independent and apply to numerical (int, float, long, ...) as well as reference values. JB uses a different approach, since its instructions are type-dependent. If the topmost two values on the stack are integers, it uses a conditional branch instruction (`if_icmpeq`) similar to that of CIL (`beq`). However, JB has no comparison instruction on integers and we need to simulate it through a sequence of JB instructions relying on constants and branch. For instance, a `ceq` statement on integers is simulated as in Fig. 8a. Instead, if the topmost two values on the stack are long, JB uses a comparison statement `lcmp` that pushes to the stack 1, 0, or -1 iff the first value is less than, equal to, or greater than the second, respectively. Hence, we simulate CIL conditional branch and comparison instructions through `lcmp`, integer constants and a conditional branch on integers. For instance, `beq` is translated into the sequence `lcmp; ifne #i;` where `i` is the target JB instruction of `beq`. Instead, the treatment of comparisons over long is similar to int. Namely, `ceq` is translated into the code in Fig. 8b. Moreover, conditional branch and comparison work also on references. Equality and inequality statements are treated as for integers, since

```
1  delegate void Del(string message);
2  void DelegateMethod(string message) {...}
3  void go() {
4          Del handler = DelegateMethod;
5          handler("Hello World");}
```

(a) C# code.

```
1  void go () {
2          ldarg.0
3          ldftn  A::DelegateMethod(string)
4          newobj A/Del::.ctor(object, int)
5          ldstr  "Hello World"
6          call  void A/Del::Invoke(string)}
```

(b) CIL

```
1  void go() {
2          ldc  "DelegateMethod(LString;)V"
3          invokestatic
   Reflection.GetMethod:(LString;)LMethod;
4          ldc  "Hello World"
5          invokevirtual  A/Del.Invoke:(LString;)}
```

(c) JB.

Fig. 9. An example of CIL delegate.

JB defines an `if_acmpeq` statement. Other CIL operators (*e.g.*, `bgt`) can be applied to arbitrary references, as long as one of them is `null`. For instance, it might branch if a reference is strictly greater than `null` (that is, if it is not `null`) and we translate these cases accordingly.

Generic types. CIL keeps information about generic types, while JB erases it into `Object`. For instance, imagine that we have a local variable `list` of type `List⟨A⟩`. At source code level, a method call like `A a = list.get(0)` in Java or `A a = list[0]` in C# is legal since the elements of the list have type `A` in both languages. At bytecode level, getting an element from the list effectively returns an object of type `A` in CIL (see Fig. 8c), while it returns an object whose static type is `Object` in JB and casts it dynamically to `A` through a `checkcast` (Fig. 8d). Hence, our translation of a CIL method call with generic return type `T` adds a `checkcast` instruction to `T` after the call.

Primitive types (*e.g.*, `int` and `long`) can be passed as generic types in CIL but not in JB. Hence, when using a primitive type for the generic parameter or return value of a CIL method call, we box and unbox the primitive value into a Java wrapper class such as `java.lang.Integer`.

Delegates. Lambda expressions have only been introduced in Java 8, while C# has been using *delegates* since its very beginning. C# implements delegates through CIL instructions that load a pointer to a method (`ldftn`) and execute it, sometime by using inner classes. Namely, C# accesses a pointer to the method through `ldftn` and calls the `Invoke` method of the delegate class. Consider for instance Fig. 9. The C# code in Fig. 9a uses a delegate to call a method. In Fig. 9b, this is compiled into a `ldftn` statement at line 4 followed by a call to `Invoke` at line 7. We translate this by using reflection and string constants, since at the time our approach was developed Julia did not support yet Java 8 and thus `invokedynamic` statements. Namely, the signature of the method pointed by `ldftn` is represented by a string, passed to an instrumentation library call in class `Reflection`, that calls this method by reflection (Fig. 9c). However, many static analyzers (including Julia) are unsound for reflection. Hence, our translation marks all signatures accessed in this way as entry points (that is, methods that might be directly called from outside the application and therefore are analyzed under the most generic assumptions). This might cause a loss of precision, since contextual information on delegates is lost, but preserves soundness.

Async and await. In C#, an `async` method returns a `Task` object that allows the caller to execute the code of the method asynchronously. On the other hand, statement `await` waits until the execution of the asynchronous method ends and extracts the results of the computation. This pattern is compiled into method pointers and reflection at CIL bytecode level, in the same way delegates are treated. Therefore, we apply the same solution for delegates that we described in Section 8.

Exceptions. While the throw of exceptions is identical in JB (athrow) and CIL (throw), exceptions handlers are different. In particular, JB has an exception table for each method that, for each try block, defines the code portion guarded by the block and the catch block that manages the exception. finally blocks are translated in JB using these constructs. Instead, CIL supports natively finally blocks. When translating them, we apply the same semantics of Java when compiling finally blocks to bytecode, that is, we add to the JB exception table a block guarding both try and catch, and pointing to the code of finally.

5. Experimental results

Our experiments are aimed at answering the research questions in Section 1.1, hence assessing the practical interest of our translation for static analysis purposes.

5.1. Experimental setup

We implemented our translation from CIL to JB through (i) a C# program that translates a CIL program to an intermediate XML representation (representing Java bytecode), and (ii) a Java program that produces a `jar` file from an XML representation. We had to split the implementation in this way since the library to read CIL bytecode (`Mono.Cecil`) is written in .NET, while the library writing `jar` bytecode (`BCEL`) is written in Java. The first part of the translation (performing the real CIL to JB translation) runs in parallel on different classes through the System.Threading.Tasks library (part of the standard .Net framework). We ran our experiments on an Intel Core i5-6600 CPU at 3.30 GHz machine with 16 GB of RAM, 64-bit Windows 7 Professional, and Java SE Runtime Environment version 1.8.0_111-b14.

The CIL to JB translator has been incorporated inside the Julia static analyzer. Such tool is exposed as a cloud service at http://portal.juliasoft.com, where one can register and get credits to analyze about 10 KLOCs. The quickstart manual points to the Visual Studio plugin that allows one to run the analyses from this IDE.

Table 1
The 5 most starred GitHub C# projects.

Program	LOC	Overall analysis time	Translation time	# fail
Shadowsocks	17,175	1'28"	4"	0
CodeHub	51,387	1'54"	8"	0
Wox	19,911	1'03"	2"	0
Dapper	26,613	1'04"	7"	0
ShareX	144,904	3'13"	19"	21
*l Total	259,990	8'42"	40"	21

We ran two distinct experiments. First of all, in order to study the precision of our approach, we analyzed with Julia's basic checkers[9] the five most popular GitHub repositories (as of April 9th, 2019) written in C# and tagged as C# repositories.[10] All the analyses were run on the Julia cloud services version 2.7.0. In particular, we inspected all the warnings with Critical and Major severity.[11] Table 1 reports the libraries we analyzed, the number of lines of code (Column **LOC**),[12] the time (**Overall Analysis Time**) consumed by the overall analysis (including both the translation from CIL to JB, the time (**Translation Time**) consumed by the translation of CIL to JB (note that this is comprised in the preceding column), and the time of Julia's analyses), and the number of unsafe methods (Column **# fail**) that our approach failed to translate. Table 2 reports the detailed experimental results, and in particular for each application the number of alarms (divided into Category - Bug or Efficiency -, Severity - Critical or Major -, and warning type[13]) produced by the analysis (Column # a), and the number of false alarms caused by loss of information in the translation from CIL to JB (Column # f, that is a subset of Column # a).

Then, in order to assess the efficiency and library coverage of our approach, we also analyzed all system libraries of the Microsoft .Net framework version 4.0.30319. They contain unsafe code (such as cryptographic code in mscorlib.dll) and might not be compiled from C#, but possibly from VB.Net or other programming languages. Table 3 reports the number of methods of the library that were correctly translated (**# met.**), the number and percentage of methods where the translation fails because of unsafe code (**# fail** and **% fail**). These translations took all together 23'39" on a Windows 10 machine with a 2-core Intel® Core™ i7-7500U CPU and 16 GB of RAM memory.

5.2. Research question 1: efficiency

As discussed in the previous section, the translation of all the .NET libraries (almost 500K methods for a total of about 5 MLOCs) took about 24 minutes, that is, about 4 methods per millisecond. The translation took at most 238 MB of RAM memory. In addition, on the 5 GitHub projects (more than 250 KLOCs), the overall analysis time took about 9 minutes, and only 40 seconds were consumed by the translation of CIL to JB (that is, about 8% of the overall analysis time). Therefore, we conclude that both the computational time and the memory consumption are negligible *w.r.t.* the overall analysis time. This shows that our approach respects Research Question 1: it deals with industrial-size software in a few minutes and with a negligible translation time.

5.3. Research question 2: precision

We manually investigated all the warnings with severity Critical and Major issued by Julia on the top 5 Github projects. Table 2 reports, for each project, the detailed experimental results. The static analysis might generate false alarms as well because of inherent approximations applied by the static analysis engine. For instance, Julia approximates together the analysis of distinct branches of if statements after the statement itself. Imagine that the condition is a Boolean flag, and if this is true, then the program initializes a variable to a non-null value. Later, the program tests again the same Boolean guard, and if true it dereferences the local variable previously initialized. In this case, Julia's BasicNullness analysis produces a false alarm since it is not able to infer that the value of the Boolean flag guarantees that the variable is non-null. Such analysis could be improved with trace partitioning [28], but the analysis probably would not scale up. We do not count these as false alarms, since we want to evaluate the imprecision due to the translation, and not inherent to Julia's analysis engine.

All together, the analyses produced 2,008 Critical and Major warnings. Among these, 88 (that is, 4%) are false alarms due to our translation from CIL to JB. In particular:

[9] https://juliasoft.com/resources/checkers/ contains a comprehensive list of Julia's checkers.
[10] We consider the number of received stars as measure of popularity of a repository. We discarded some projects *tagged* as C# that actually mostly contain native code (corefx, coreclr, aspnetcore), that did not compile in Visual Studio (roslyn, powershell), or that are particularly small (wavefunction, below 1 KLOC).
[11] Julia's warnings have 4 distinct levels of severity: Critical, Major, Minor and Info.
[12] LOC are computed with LocMetrics http://www.locmetrics.com/.
[13] Table A.4 in the Appendix reports the exhaustive lists of Julia warnings.

Table 2
Experimental results on the 5 most starred GitHub C# projects.

Category	Severity	Warning	ShadowSocks		CodeHub		Wox		Dapper		ShareX		Total		
			#a	#f	#a	#f	#a	#f	#a	#f	#a	#f	#a	#f	% f
		ActualNull	3	0	15	0	1	0	2	0	14	2	35	2	6%
		AssigningInsteadOfComparing							1	0			1	0	0%
		CallOnNull	5	1	1	1			5	4	6	2	17	8	47%
		CastIntegralComputationIntoFloatingPoint									5	0	5	0	0%
		CompareToWithDefaultEquals	1	0									1	0	0%
		EqualsOnDisjointTypes					3	3					3	3	100%
Bug	**Critical**	FloatComparison									5	0	5	0	0%
		FormalNull	1	0									1	0	0%
		GetFieldFromNull	1	1									1	1	100%
		ReturnValueShouldBeUsed	1	0							1	0	2	0	0%
		UnsafeLazyInitialisation	6	0			2	0	3	0	5	0	16	0	0%
		XXEAttack									4	0	4	0	0%
		Total Critical Bug	18	2	16	1	6	3	11	4	40	4	91	14	15%
		ArrayStore							4	0	1	1	5	1	20%
		AssignmentoToUnreadParameter			1	0					10	0	11	0	0%
		AssignmentoToUnusedParameter			3	0			2	0	1	0	6	0	0%
		CastIntComputationIntoLong	7	0							3	0	10	0	0%
		ClasscastOfFormal	14	0	4	4	5	0	10	0	32	2	65	6	9%
		ImpossibleInstanceOf			3	3			2	0	7	5	12	8	67%
		InadequateCallInProduction									1	0	1	0	0%
		MissingNullnessCheckOfReturnedValue	1	0	1	0							2	0	0%
		NonShortCircuitAND			4	4	3	3	4	4	3	3	14	14	100%
		NonShortCircuitOR									2	0	2	0	0%
Bug	**Major**	ResourceNotClosedAtEndOfMethod	8	0	248	0	3	0	2	0	55	0	316	0	0%
		SuspiciousInheritanceOfEquals	9	0	32	0	4	4	2	0	28	28	75	32	43%
		TestIsPredetermined	18	3			2	0	2	0	111	0	133	3	2%
		Uncalled	2	0	13	2	23	2	33	0	28	0	99	4	4%
		UnexpectedInstanceof									1	1	1	1	100%
		UnreachableInstruction	12	3			2	0	1	0	72	0	87	3	3%
		UnsafeBase64Encoding	3	0			1	0			2	0	6	0	0%
		VariableCanOnlyBeNull			1	0							1	0	0%
		WeakHashingAlgorithmWarning			1	0					7	0	8	0	0%
		Total Major Bug	74	6	311	13	43	9	62	4	364	40	854	72	8%
		BlockingCallInsideSynchronization	1	0									1	0	0%
		Equals									2	0	2	0	0%
		FieldIsOnlyUsedInConstructors					2	0	3	0	6	0	11	0	0%
		FieldShouldBeReplacedByLocals	43	0	5	0	3	0			978	0	1029	0	0%
Efficiency	**Major**	TautologicalInstanceof									2	0	2	0	0%
		UselessCall					2	0			2	0	4	0	0%
		UselessConstruction	1	0									1	0	0%
		UselessTest	5	0							4	2	9	2	22%
		UseLogInstead	4	0									4	0	0%
		Total Major Inefficiency	54	0	5	0	7	0	3	0	994	2	1063	2	0%
		Total Critical and Major	146	8	332	14	56	12	76	8	1398	46	2008	88	4%

- the nullness checker produced 11 false alarms (ActualNull, CallOnNull, and GetFieldFromNull) because of nullness checks introduced by the C# compiler that were not considered as synthetic (that is, code generated by the compiler) by Julia analyses;
- 19 false alarms (ArrayStore, ClasscastOfFormal, ImpossibleInstanceOf, and UnexpectedInstanceOf) were due to missing type information in JB (in particular, since CIL contains information about generic types, while JB does not support it) and type casts introduced by the C# compiler and not considered synthetic by Julia analyses;
- 14 false alarms are NonShortCircuitAND warnings (that were all false alarms). These are due to instrumentation checks added by the C# compiler on conditions on nullable variables;
- 32 false alarms are SuspiciousInheritanceOfEquals warnings due to the fact that C# `structs` do not need to explicitly implement `Equals` (even if this is often done for performance reasons); and
- the remaining 12 false alarms (TestIsPredetermined, Uncalled, UnreachableInstruction, and UselessTest) are due to lack of knowledge in Julia of the C# frameworks.

Julia analyses have been further refined in order to take into account synthetic code that caused false alarms as well as knowledge about some C# frameworks through ad-hoc framework specification [29]. However, we leave these refinements

Table 3
Experimental results on libraries.

Library	# met.	# fail	% fail	Library	# met.	# fail	% fail
Accessibility	55	0	0,0%	System.Dynamic	518	4	0,8%
AspNetMMCExt	462	0	0,0%	System.EnterpriseServices	1.456	2	0,1%
CustomMarshalers	113	0	0,0%	System.EnterpriseServices.Wrapper	61	105	63,3%
ISymWrapper	82	73	47,1%	System.IdentityModel	5.150	15	0,3%
Microsoft.Activities.Build	109	0	0,0%	System.IdentityModel.Selectors	426	1	0,2%
Microsoft.Build.Conversion.v4.0	84	0	0,0%	System.IdentityModel.Services	822	0	0,0%
Microsoft.Build	6.753	14	0,2%	System.IO.Compression	228	0	0,0%
Microsoft.Build.Engine	3.098	1	0,0%	System.IO.Compression.FileSystem	45	0	0,0%
Microsoft.Build.Framework	717	0	0,0%	System.IO.Log	758	6	0,8%
Microsoft.Build.Tasks.v4.0	5.293	9	0,2%	System.Management	1.672	0	0,0%
Microsoft.Build.Utilities.v4.0	1.190	2	0,2%	System.Management.Instrumentation	728	0	0,0%
Microsoft.CSharp	2.341	0	0,0%	System.Messaging	986	0	0,0%
Microsoft.Data.Entity.Build.Tasks	64	0	0,0%	System.Net	861	10	1,1%
Microsoft.Internal.Tasks.Dataflow	1.217	1	0,1%	System.Net.Http	1.142	0	0,0%
Microsoft.JScript	3.943	0	0,0%	System.Net.Http.WebRequest	29	0	0,0%
Microsoft.Transactions.Bridge	2.796	4	0,1%	System.Numerics	584	3	0,5%
Microsoft.Transactions.Bridge.Dtc	840	63	7,0%	System.Numerics.Vectors	1	0	0,0%
Microsoft.VisualBasic.Activities.Compiler	152	40	20,8%	System.Printing	2.024	235	10,4%
Microsoft.VisualBasic.Compatibility.Data	604	3	0,5%	System.Reflection.context	797	0	0,0%
Microsoft.VisualBasic.Compatibility	3.994	8	0,2%	System.Runtime.Caching	485	0	0,0%
Microsoft.VisualBasic	2.446	64	2,5%	System.Runtime.DurableInstancing	746	0	0,0%
Microsoft.VisualC	11	0	0,0%	System.Runtime.Remoting	1.438	3	0,2%
Microsoft.VisualC.STLCLR	538	39	6,8%	System.Runtime.Serialization	5.592	72	1,3%
mscorlib	27.932	880	3,1%	System.Runtime.Serialization.Formatters.Soap	343	0	0,0%
PresentationBuildTasks	2.355	0	0,0%	System.Security	1.375	39	2,8%
PresentationCore	22.425	394	1,7%	System.ServiceModel.Activation	916	0	0,0%
PresentationFramework-SystemCore	21	0	0,0%	System.ServiceModel.Activities	3.092	1	0,0%
PresentationFramework-SystemData	7	0	0,0%	System.ServiceModel.Channels	859	0	0,0%
PresentationFramework-SystemDrawing	16	0	0,0%	System.ServiceModel.Discovery	2.168	0	0,0%
PresentationFramework-SystemXml	14	0	0,0%	System.ServiceModel	34.687	20	0,1%
PresentationFramework-SystemXmlLinq	6	0	0,0%	System.ServiceModel.Internals	1.212	33	2,7%
PresentationFramework.Aero	273	0	0,0%	System.ServiceModel.Routing	602	0	0,0%
PresentationFramework.AeroLite	89	0	0,0%	System.ServiceModel.ServiceMoniker40	4	0	0,0%
PresentationFramework.Classic	133	0	0,0%	System.ServiceModel.WasHosting	71	0	0,0%
PresentationFramework	36.971	31	0,1%	System.ServiceModel.Web	1.485	0	0,0%
PresentationFramework.Luna	272	0	0,0%	System.ServiceProcess	294	13	4,2%
PresentationFramework.Royale	202	0	0,0%	System.Speech	3.100	0	0,0%
PresentationUI	1.434	0	0,0%	System.Transactions	1.565	0	0,0%
ReachFramework	4.143	4	0,1%	System.Web.ApplicationServices	183	0	0,0%
SMDiagnostics	204	0	0,0%	System.Web.DataVisualization.Design	286	0	0,0%
sysglobl	291	0	0,0%	System.Web.DataVisualization	5.746	1	0,0%
System.Activities.Core.Presentation	2.147	0	0,0%	System.Web	28.462	25	0,1%
System.Activities	9.854	1	0,0%	System.Web.DynamicData.Design	44	0	0,0%
System.Activities.DurableInstancing	490	4	0,8%	System.Web.DynamicData	1.605	0	0,0%
System.Activities.Presentation	9.088	0	0,0%	System.Web.Entity.Design	513	0	0,0%
System.AddIn.Contract	197	0	0,0%	System.Web.Entity	589	0	0,0%
System.AddIn	659	0	0,0%	System.Web.Extensions.Design	1.700	0	0,0%
System.ComponentModel.Composition	1.831	0	0,0%	System.Web.Extensions	4.245	0	0,0%
system.componentmodel.composition.registration	184	0	0,0%	System.Web.Mobile	4.683	0	0,0%
System.ComponentModel.DataAnnotations	529	0	0,0%	System.Web.RegularExpressions	216	0	0,0%
System.Configuration	2.049	0	0,0%	System.Web.Services	2.696	0	0,0%
System.Configuration.Install	166	0	0,0%	System.Windows.Controls.Ribbon	3.128	0	0,0%
System.Core	7.935	114	1,4%	System.Windows.Forms.DataVisualization.Design	159	0	0,0%
System.Data.DataSetExtensions	148	0	0,0%	System.Windows.Forms.DataVisualization	5.798	1	0,0%
System.Data	13.670	44	0,3%	System.Windows.Forms	28.573	30	0,1%
System.Data.Entity.Design	1.322	0	0,0%	System.Windows.Input.Manipulations	378	0	0,0%
System.Data.Entity	18.849	4	0,0%	System.Windows.Presentation	36	0	0,0%
System.Data.Linq	4.224	0	0,0%	System.Workflow.Activities	4.957	0	0,0%
System.Data.OracleClient	2.178	0	0,0%	System.Workflow.ComponentModel	5.268	0	0,0%
System.Data.Services.Client	2.001	1	0,0%	System.Workflow.Runtime	2.148	1	0,0%
System.Data.Services.Design	657	0	0,0%	System.WorkflowServices	1.803	0	0,0%
System.Data.Services	3.334	1	0,0%	System.Xaml	3.801	0	0,0%
System.Data.SqlXml	4.527	16	0,4%	System.Xaml.Hosting	111	0	0,0%
System.Deployment	2.475	2	0,1%	System.XML	12.582	163	1,3%
System.Design	13.513	2	0,0%	System.Xml.Linq	925	0	0,0%
System.Device	224	0	0,0%	System.Xml.Serialization	0	0	0,0%
System.Diagnostics.Tracing	51	0	0,0%	UIAutomationClient	1.005	3	0,3%

(*continued on next page*)

Table 3 (*continued*)

Library	# met.	# fail	% fail	Library	# met.	# fail	% fail
System.DirectoryServices.AccountManagement	1.322	0	0,0%	UIAutomationClientsideProviders	2.323	53	2,2%
System.DirectoryServices	2.072	4	0,2%	UIAutomationProvider	145	0	0,0%
System.DirectoryServices.Protocols	873	2	0,2%	UIAutomationTypes	977	2	0,2%
System	17.748	238	1,3%	WindowsBase	6.843	9	0,1%
System.Drawing.Design	351	0	0,0%	WindowsFormsIntegration	430	0	0,0%
System.Drawing	4.015	8	0,2%	XamlBuildTask	612	0	0,0%
				XsdBuildTask	199	0	0,0%
Total	473.864	2.921	0,6%				

out of the experiments of this article since our goal is to evaluate if the CIL to JB translation introduces imprecision, without ad hoc refinements of the Julia core analyses.

Among the true alarms, there are two cases that produced quite a lot of warnings in specific projects:

- 978 FieldShouldBeReplacedByLocals alarms were issued on ShareX. These refer to fields introduced by ASP.NET in order to represent Web UI elements that are first initialized and then used only in one method; and
- 248 ResourceNotClosedAtEndOfMethod alarms were issued on CodeHub (and other 55 on ShareX) on instances of UI components. These components effectively are disposable objects, but often the developer does not take care of their disposal since such objects do not consume many resources.

Therefore, while these warnings are true alarms, one might prefer to switch them off (*e.g.*, by not running these analyses or by adding framework specifications to avoid them) and focus on more critical alarms.

However, these experiments show that our approach respects Research Question 2, that is, the CIL to JB translation produced less than 10% false alarms because of the translation.

5.4. Research Question 3: libraries

Only ShareX, among the 5 top GitHub C# projects that we analyzed, contains unsafe methods. In particular, its 21 unsafe methods belong to 5 distinct classes: (i) the Item property of ShareX.HelpersLib.ConvolutionMatrix returns a pointer to a double (that is, an element inside the matrix), (ii) ShareX.HelpersLib.ConvolutionMatrixManager contains seven methods (namely, Apply, EdgeDetect, Emboss, GaussianBlur, MeanRemoval, Sharpen, and Smooth) that deal with instances of such matrixes in an unsafe way, (iii) ShareX.HelpersLib.UnsafeBitmap contains 11 unsafe methods (namely, getter of property Pointer, the 2 implementations of ClearPixel, Compare, the 2 implementations of GetPixel, IsTransparent, and the 4 implementations of SetPixel), (iv) ShareX.ImageEffectsLib.GaussianBlur and (v) ShareX.ImageEffectsLib.MatrixConvolution both implement an Apply unsafe method.

About the libraries, Table 3 shows that the translation succeeds to translate 99.4% of their methods. Among the most representative libraries (that is, with at least $1,000$ methods), mscorlib (that contains the basic classes and implementation of the .NET framework and is therefore expected to contain several unsafe methods) exposed the highest percentage (that is, 3.1%) of unsafe methods.

Therefore, we conclude that our approach respects Research Question 3, since we were able to always translate at least 95% of method even in the libraries containing the most unsafe code, such as mscorlib.

6. Conclusion

This article formalizes a correct translation from CIL to JB, for static analysis. To assess its feasibility and interest, the translation has been implemented and connected to the Julia analyzer. Experiments show positive results for efficiency, precision, and libraries' coverage. As future work, we plan to (i) improve Julia precision in the corner cases highlighted by our experiments, (ii) investigate new .Net properties of interest (e.g., deprecated cryptographic APIs, or pairing begin and end invoke in C# delegates), and (iii) rely on `invokedynamic` when translating delegates (see Section 8). In the longer term, our translation could let Java and C# interoperate, by compiling both into JB. This will require the full translation of the C# framework, including unsafe code.

Appendix A. Julia warnings

Table A.4 reports the complete list of warnings that could be produced by Julia (version 2.7.0), where column **Warning** reports the type of the warning, **Checker** the checker that produces the warning, **Description** the description of the warning,

Table A.4
Warnings of Julia 2.7.0.

Warning	Checker	Description	CWE
AbsOfRandom	AbsOfRandom	The absolute value of a random number might actually be negative	682
ApproximateE	Approximation	An approximate value of E is used instead of a constant in the libraries	197
ApproximatePI	Approximation	An approximate value of PI is used instead of a constant in the libraries	197
CastIntComputationIntoLong	Approximation	The result of an integer computation that might overflow is cast into long, with possible loss of precision	190
CastIntegralComputationIntoFloatingPoint	Approximation	The result of an integral computation that might lose precision is cast into a floating point value	197
FloatComparison	Approximation	A comparison between non-integral values might be unreliable	197
AuthenticationSetToAnonymous	Authentication	The LDAP authentication is set to anonymous, thus compromising security	287
HostNameInCondition	Authentication	A host name is used in a condition	287
UnauthenticatedWebAPI	Authentication	A Web API method is not annotated for authentication	287
AssigningInsteadOfComparing	BadEq	The assignment into a Boolean value is used in a condition, while == would be expected	481
Equality	BadEq	Two objects are compared with == but equals() seems more appropriate instead	595
EqualsOnArrays	BadEq	Two arrays are compared with equals()	595
EqualsOnDisjointTypes	BadEq	Two objects are compared by equals() but they have always distinct types	596
Equals	BadEq	Two objects are compared with equals() but == seems more appropriate instead	480
ImpossibleEquality	BadEq	Two objects are compared by == but the comparison will always fail	595
ImpossibleEquals	BadEq	Two objects are compared by equals() but the comparison will always fail	596
InefficientStringEmptynessTest	BadEq	A string is compared to the empty string instead of using isEmpty()	597
CaseOverride	BadExtension	A method has a name identical to another in a superclass, up to capitalisation	628
FieldShadowed	BadExtension	A class defines a field with the same name as another in a superclass	485
ParametersOverride	BadExtension	A method has the same signature as another in a superclass, up to the package of some class	686
RedundantImplements	BadExtension	An implements clause is already present and could be removed	398
AddressInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @AddressTrusted	74
AddressInjection	BasicInjection	Tainted data might flow into the creation of an Internet address	74
CodeInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @CodeTrusted	94
CodeInjection	BasicInjection	Tainted data might flow into a script execution routine	94
CommandInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @CommandTrusted	78
CommandInjection	BasicInjection	Tainted data might flow into a command execution routine	78
ControlInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @ControlTrusted	74
ControlInjection	BasicInjection	Tainted data might flow into a control modifying method	74
DOSInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @DenialTrusted	74
DOSInjection	BasicInjection	Tainted data might flow into a method that makes the computer sleep or wait	74
DeviceInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @AddressTrusted	74
DeviceInjection	BasicInjection	Tainted data might flow into the creation of an Internet address	74
EvalInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @EvalTrusted	95
EvalInjection	BasicInjection	Tainted data might flow into code that dynamically evaluates an expression	95
GenericInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @Trusted	74
GenericInjection	BasicInjection	Tainted data might flow into a trusted parameter	74
HttpResponseInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @HttpResponseTrusted	113
HttpResponseSplitting	BasicInjection	Tainted data might flow into an HTTP response	113
LDAPAttributeInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @AttributeTrusted	90
LDAPAttributeInjection	BasicInjection	Tainted data might flow into an LDAP attribute	90
LDAPFilterInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @FilterTrusted	90
LDAPFilterInjection	BasicInjection	Tainted data might flow into the filter of an LDAP search	90
LogForging	BasicInjection	Tainted data might flow into a log	117
LogInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @LogTrusted	117
MessageInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @MessageTrusted	319
MessageInjection	BasicInjection	Tainted data might flow into a message sent by the device	319
PathInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @PathTrusted	22
PathInjection	BasicInjection	Tainted data might flow into a file path creation method	22
ReflectionInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @ReflectionTrusted	470
ReflectionInjection	BasicInjection	Tainted data might flow into a reflection method	470
ResourceInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @ResourceTrusted	74
ResourceInjection	BasicInjection	Tainted data might flow into a variable annotated as @ResourceTrusted	74

(*continued on next page*)

Table A.4 (*continued*)

Warning	Checker	Description	CWE
SessionInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @SessionTrusted	501
SessionInjection	BasicInjection	Tainted data might flow into a session	501
SqlInjectionIntoField	BasicInjection	The request of a servlet might flow into a field annotated as @SqlTrusted, unsanitized	89
SqlInjection	BasicInjection	Tainted data might flow into an sql query, unsanitized	89
TrustBoundaryViolationIntoField	BasicInjection	Tainted data flows into a field annotated as @BoundaryTrusted	501
TrustBoundaryViolation	BasicInjection	Tainted data might flow into a bundle of information that should not contain tainted pieces of information	501
URLInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @UrlTrusted	74
URLInjection	BasicInjection	Tainted data might flow into a URL creation	74
XPathInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @XPathTrusted	643
XPathInjection	BasicInjection	Tainted data might flow into an xpath creation method	643
XSSInjectionIntoField	BasicInjection	Tainted data flows into a field annotated as @CrossSiteTrusted	79
XSSInjection	BasicInjection	Tainted data might flow into a script execution routine	79
ActualNull	BasicNullness	An actual parameter passed to a method might be null	476
ArrayLengthOfNull	BasicNullness	The length of a possibly null array is computed	476
ArrayLoadFromNull	BasicNullness	An element of a possibly null array is read	476
ArrayStoreIntoNull	BasicNullness	An element of a possibly null array is written	476
CallOnNull	BasicNullness	A method call might occur on a null receiver	476
FormalNull	BasicNullness	A formal parameter of a method or constructor might hold null	476
GetFieldFromNull	BasicNullness	A field is read from a possibly null receiver	476
MethodShouldNotReturnNull	BasicNullness	A method returns null but is normally assumed to return a non-null value	227
MissingNullnessCheckOfReturnedValue	BasicNullness	The return value of a method is checked against null, but not here	252
PutFieldIntoNull	BasicNullness	A field is written into a possibly null receiver	476
ReturningNullForArray	BasicNullness	A method returns null instead of an empty array	476
ReturningNullForBoolean	BasicNullness	A method returns null instead of a java.lang.Boolean	476
ReturningNullForOptional	BasicNullness	A method returns null instead of a java.util.Optional	476
SynchronizationOnNull	BasicNullness	A synchronization might occur on null	476
ThrowOfNull	BasicNullness	A throw command might throw null	476
VariableCanOnlyBeNull	BasicNullness	A variable that can only hold null is dereferenced	456
CallSuper	CallSuper	A call to super() is missing	573
CallToToStringOnArray	CallsOnArray	toString() is called over an array	440
ArrayStore	Classcast	The value written into an array cannot be assigned to the type of the elements of the array	704
ClasscastGeneric	Classcast	A classcast might be incorrect at runtime	704
ClasscastOfField	Classcast	A classcast of a field might be incorrect at runtime	704
ClasscastOfFormal	Classcast	A classcast of a formal parameter might be incorrect at runtime	704
ClasscastOfMethodReturn	Classcast	A classcast of the return value of a method might be incorrect at runtime	704
UselessClasscast	Classcast	A classcast is useless and can be removed from the code	398
CloneForNonCloneable	Clone	Method clone() is defined in a non-cloneable class	491
NonFinalCloneMethod	Clone	Method clone() is not final, which allows object-hijack	491
SubclassesMayBeCloned	Clone	A subclass of a non-cloneable class may be cloned	491
AsymmetricalCompareTo	CompareTo	compareTo() is not symmetrical	596
CompareToForNonObject	CompareTo	compareTo() is defined for a non-java.lang.Object argument but the comparable class is raw	227
CompareToInNonComparable	CompareTo	compareTo() is defined in a class that is not an instance of java.lang.Comparable	227
CompareToInconsistentWithEquals	CompareTo	compareTo() is defined inconsistently from equals()	596
CompareToWithDefaultEquals	CompareTo	compareTo() is defined but equals() is inherited from Object	596
BlockingCallInsideSynchronization	Concurrency	A blocking call occurs inside a synchronization block, hence increasing monitor contention and reducing performance	833
ExpensiveSynchronizationOnStatic	Concurrency	A synchronized statement on a static guard should lock an instance guard instead	413
ImpossibleClientSideLocking	Concurrency	Synchronisation occurs on a concurrent map that does not allow client-side locking	413
SynchronisationOnInternedString	Concurrency	Synchronisation occurs on an interned string	412
SynchronousCallToThreadBody	Concurrency	The body of a thread is called synchronously	572
UnsafeLazyInitialisation	Concurrency	A static field is lazily initialized in an incorrect way	609
UselessSynchronization	Concurrency	A synchronized statement is useless	585
UselessVolatileModifier	Concurrency	A field should not be declared volatile	662
VolatileArrayField	Concurrency	An array field has been declared volatile	567
VolatileContainerField	Concurrency	A container field has been declared volatile	567
InsecureCookie	Cookie	An insecure cookie has been used	614
PossibleInsecureCookieCreation	Cookie	An insecure cookie might have been created	614
InsecureKeyDerivationFunction	Cryptography	An insecure key derivation function is used	326
RiskyCryptographicAlgorithm	Cryptography	An unsafe cryptographic algorithm is used	327
UnsafeBase64Encoding	Cryptography	The Base64 encoding is used, but it is nowadays easy to read	327

Table A.4 (*continued*)

Warning	Checker	Description	CWE
WeakHashingAlgorithm	Cryptography	A weak, possibly reversible cryptographic algorithm is used	328
ClassNeverInstantiated	Deadcode	A class is never instantiated	561
Uncalled	Deadcode	A method or constructor is not called	561
UnreachableInstruction	Deadcode	An instruction will never be executed	561
EqualsNotAgainstObject	EqualsHashCode	equals() is defined against a non-object class	227
NoEquals	EqualsHashCode	The equals() method seems needed	581
NoHashCode	EqualsHashCode	The hashCode() method seems needed	581
SuspiciousInheritanceOfEquals	EqualsHashCode	equals() is inherited but extra fields have been added	187
BroadThrowsClause	ExceptionHandlers	The throws clause of a method or constructor declares a very generic exception type	397
EmptyExceptionHandler	ExceptionHandlers	An exception handler has an empty body	390
GenericExceptionHandler	ExceptionHandlers	An exception handler is used for a very generic exception type	396
InappropriateExceptionHandler	ExceptionHandlers	An exception class is caught that should be rather prevented	395
FieldNeverRead	FieldAccess	A field is never read in the code	772
FieldNeverUsed	FieldAccess	A field is never read nor written in the code	398
FieldNeverWritten	FieldAccess	A field is never written in the code	456
MissingSynchronized	GuardedBy	A synchronized statement is needed to access a field	567
UnguardedField	GuardedBy	A field is accessed without the expected lock being held	567
UnguardedMethodOrConstructor	GuardedBy	A public or protected method or constructor might be called without the expected lock being held	820
UnguardedParameter	GuardedBy	A parameter of a method or constructor is accessed without the expected lock being held	567
FieldIsOnlyUsedInConstructors	ImproperField	A field is only used in a constructor and could hence be replaced by a local variable	772
FieldIsOnlyUsedInStaticInitialiser	ImproperField	A field is only used inside a static initializer and could hence be replaced by a local variable	772
FieldShouldBeReplacedByLocals	ImproperField	A field should be replaced by local variables inside the methods that use it	772
MutableEnum	ImproperField	An enumeration can be muted	607
UselessFieldUpdate	ImproperField	A field is updated but the written value is never used later	563
InefficientBoxUnbox	InefficientConstruction	A box/unbox sequence can be simplified and made more efficient	227
InefficientConstructionForGetClass	InefficientConstruction	A class is only instantiated to get its class tag, instead of using its name and the .class pseudo-field	227
InefficientConstruction	InefficientConstruction	The construction of an object might be replaced by a literal or by a call to a factory method	400
PassingEmptyArray	InefficientConstruction	An empty array is passed to a method instead of an array of the proper size	227
InfiniteRecursion	InfiniteRecursion	A method call looks infinitely recursive	674
AmbiguousCallFromInnerClass	InnerClasses	A method call from an inner class is ambiguous	227
InnerClassShouldBeStatic	InnerClasses	An inner class should be made static	492
LDAPPoisoning	Ldap	An LDAP poisoning attack seems possible	349
LeakThroughCallbackField	Leak	Data might be leaked by being stored into a field of an operating system callback	664
LeakThroughCallback	Leak	Data might be leaked through an operating system callback	664
LeakThroughField	Leak	Data might be leaked by being stored into a field	664
LeakThroughInnerClass	Leak	Data might be leaked because of a non-static inner class	664
ActualInnerNull	Nullness	An actual parameter passed to a method is an array or collection possibly containing null	476
ActualNull	Nullness	An actual parameter passed to a method might be null	476
ArrayLengthOfNull	Nullness	The length of a possibly null array is computed	476
ArrayLoadFromNull	Nullness	An element of a possibly null array is read	476
ArrayStoreIntoNull	Nullness	An element of a possibly null array is written	476
CallOnNull	Nullness	A method call might occur on a null receiver	476
FieldInnerNull	Nullness	A field holds an array or collection possibly containing null	476
FieldNull	Nullness	A field might hold null	476
FormalInnerNull	Nullness	A formal parameter of a method or constructor is an array or collection possibly containing null	476
FormalNull	Nullness	A formal parameter of a method or constructor might hold null	476
GetFieldFromNull	Nullness	A field is read from a possibly null receiver	476
MethodReturnsInnerNull	Nullness	A method returns an array or collection possibly containing null	476
MethodReturnsNull	Nullness	A method might return null	476
PutFieldIntoNull	Nullness	A field is written into a possibly null receiver	476
SynchronizationOnNull	Nullness	A synchronization might occur on null	476
ThrowOfNull	Nullness	A throw command might throw null	476
UselessNullnessTestOfField	Nullness	A comparison of a field against null is always true or always false	398
UselessNullnessTestOfFormal	Nullness	A comparison of a formal parameter against null is always true or always false	398
UselessNullnessTestOfMethodReturn	Nullness	A comparison of the return value of a method against null is always true or always false	253

(*continued on next page*)

Table A.4 (*continued*)

Warning	Checker	Description	CWE
UselessNullnessTest	Nullness	A comparison of a value against null is always true or always false	398
HardcodedPassword	Passwords	A hardcoded password is used	259
PasswordInPropertyFile	Passwords	A password is retrieved from a property file	522
InadequateCallInProduction	Production	A method should not be called in production code	477
UseLogInstead	Production	A method should be replaced with a logging code in production code	477
InsecureRandom	Random	An insecure random number generator is used instead of a secure one	330
SuboptimalRandomNumber	Random	A random number generator is recreated just before its use	332
HardcodedFileName	Resources	A file name is provided as a hardcoded string	547
MissingSerialVersionField	Serialization	Missing or incorrect serialVersionUID in serializable class	913
NonSerializableElementsOfField	Serialization	A non-transient field of a serializable class might hold a map or collection whose elements might be non-serializable	913
NonSerializableField	Serialization	A non-transient field of a serializable class might hold a non-serializable value	913
NonSerializableOuterClass	Serialization	An inner non-static serializable class has a non-serializable outer class	913
UnexpectedSerialVersionField	Serialization	A serialVersionUID field is defined where it is not expected	913
ANDAgainstConstant	ShortCircuit	An & operation operates on a Boolean constant	480
InefficientSameValueAND	ShortCircuit	&& should be used instead of & for better efficiency	480
InefficientSameValueOR	ShortCircuit	|| should be used instead of | for better efficiency	480
NonShortCircuitAND	ShortCircuit	There is a suspicious use of & instead of &&	768
NonShortCircuitOR	ShortCircuit	There is a suspicious use of | instead of ||	768
ORAgainstConstant	ShortCircuit	An | operation operates on a Boolean constant	480
SideEffectInAssertion	SideEffects	An assertion checks a condition with side-effects	665
SetStaticInNonStatic	StaticFieldAccess	A static field has been modified from a non-static method	398
UnusedClass	UnusedClass	A class is not used	398
ReturnValueShouldBeUsed	UnusedReturnValue	The returned value of a non-void method is thrown away but should instead be checked or used	252
UselessCallToAPureMethod	UnusedReturnValue	A call to a pure method is performed and the returned value is missing or discarded	227
AssignmentToUnreadParameter	UselessAssignment	A parameter is assigned before being read	563
AssignmentToUnusedParameter	UselessAssignment	A parameter is assigned but the written value is never used later	563
AssignmentToUnusedVariable	UselessAssignment	A variable is assigned but the written value is never used later	563
TautologicalAssignment	UselessAssignment	A field is assigned to itself	665
UselessAssignmentToDefaultValue	UselessAssignment	A field is assigned in a constructor or finaliser to its default value	665
UselessCallForIntegralValue	UselessCall	A call is useless when its argument is an integral value	227
UselessCall	UselessCall	A method call seems useless	398
UselessConstruction	UselessConstruction	An object is created and not assigned, although its construction has only side-effects on that object	392, 771
ImpossibleInstanceof	UselessInstanceof	An instanceof test is always true or always false	570
TautologicalInstanceof	UselessInstanceof	An instanceof test is always true or always false	571
UnexpectedInstanceof	UselessInstanceof	An instanceof test should not be used	227
TestIsPredetermined	UselessTest	A test is always true or always false and can hence be removed	571
UselessTest	UselessTest	The result of a test is not used and the test can hence be removed	398
XXEAttack	Xxe	A method call might perform an unrestricted XML external entity reference	611
EmptyJarEntry	Zip	An empty jar entry is added to a jar file	909
EmptyZipEntry	Zip	An empty zip entry is added to a zip file	909

and **CWE** the CWE identifier[14] of the weakness reported by the warning. The list of warnings of the latest release can be retrieved at https://juliasoft.com/resources/warnings/.

Appendix B. Translation of CIL statements

Table B.5 reports the translation of all CIL statements as defined by the ECMA-335 [30]. When presenting this translation, we adopted the following shortcuts:

- < i > denotes the character representing a numerical type (l for long, f for float, d for double, i for integer, b for Boolean, c for char, s for short, a for references) computed by the static type inference on the top values of the stack before the instruction,
- < iboxed> denotes the boxed representation of a native type < i > (Boolean, Byte, Char, Double, Float, Integer, Long, or Short), and < boxedi > the native type of a boxed type,

[14] http://cwe.mitre.org/.

Table B.5
Translation of CIL statements.

CIL	Description	JB Translation
	BASIC INSTRUCTIONS (Section III.3 of [30])	
add[.ovf]	add numeric values	See Fig. 7
and	bitwise AND	< i > and
arglist	get argument list	Unsafe instruction
beq. < length >	branch on equal	if_< i > cmpeq
bge[.un]. < length >	branch on greater than or equal to	if_< i > cmpge
bgt[.un]. < length >	branch on greater than	if_< i > cmpgt
ble[.un]. < length >	branch on less than or equal to	if_< i > cmple
blt[.un]. < length >	branch on less than	if_< i > cmplt
bne.un < length >	branch on not equal or unordered	if_< i > cmpne
br. < length >	unconditional branch	goto
break	breakpoint instruction	nop
brfalse. < length >	branch on false null or zero	ifeq or ifnull
brtrue. < length >	branch on non-false or non-null	ifne or ifnonnull
call	call a method	See Fig. 7
calli	indirect method call	Unsafe instruction
ceq	compare equal	if_< i > cmpeq l1 iconst0 goto l2 l1 : iconst1 l2 : nop
cgt[.un]	compare greater than	[if_< i > cmpgt|if_acmpne] l1 iconst0 goto l2 l1 : iconst1 l2 : nop
ckfinite	check for a finite real number	nop
clt[.un]	compare less than unsigned or unordered	if_< i > cmplt l1 iconst0 goto l2 l1 : iconst1 l2 : nop
conv[.ovf]. < type > [.un]	data conversion	< i > 2 < typei >
cpblk	copy data from memory to memory	Unsafe instruction
div[.un]	divide values	< i > div
dup	duplicate the top value of the stack	See Fig. 7
endfilter	end exception handling filter clause	aload < excindex > swap ifne < handler > athrow
endfinally	end the finally or fault clause of an exception block	nop
initblk	initialize a block of memory to a value	Unsafe instruction
jmp	jump to method	Unsafe instruction
ldarg. < length >	load argument onto the stack	See Fig. 7
ldarga. < length >	load an argument address	See Fig. 7 (translation of ldloca replacing the correct index for arguments instead of variables)
ldc. < type >	load numeric constant	< i > const
ldftn	load method pointer	if followed by newobj T(a1...an) : pop ldc "T(a1...an)" invokestaticReflection.getMethod(LString;)Method; Otherwise, unsafe instruction

(*continued on next page*)

Table B.5 (*continued*)

CIL	Description	JB Translation
ldind. < type >	load value indirect onto the stack	See Fig. 7
ldloc	load local variable onto the stack	See Fig. 7
ldloca. < length >	load local variable address	See Fig. 7
ldnull	load a null pointer	aconstnull
leave. < length >	exit a protected region of code	goto
localloc	allocate space in the local dynamic memory pool	Unsafe instruction
mul[.ovf. < type >]	multiply values	< i > mul
neg	negate	< i > neg
nop	no operation	nop
not	bitwise complement	< i > const − 1 < i > xor
or	bitwise OR	< i > or
pop	remove the top element of the stack	pop
rem[.un]	compute remainder	< i > rem
ret	return from method	< i > return
shl	shift integer left	< i > shl
shr[.un]	shift integer right	< i > shr
starg. < length >	store a value in an argument slot	See Fig. 7 (translation of stloc replacing the correct index for arguments instead of variables)
stind. < type >	store value indirect from stack	See Fig. 7
stloc	pop value from stack to local variable	See Fig. 7
sub[ovf. < type >]	subtract numeric values	< i > sub
switch	table switch based on value	switch
xor	bitwise XOR	< i > xor
	OBJECT MODEL INSTRUCTIONS (Section III.4 of [30])	
box	convert a boxable value to its boxed form	invokestatic< iboxed > .box(< boxedi >)Z >
callvirt	call a method associated at runtime with an object	See Fig. 7, case call
castclass	cast an object to a class	checkcast
cpobj	copy a value from one address to another	If the value comes from ldflda < f > : putfield < f > If the value comes from ldsflda < f > : putstatic < f > Otherwise, unsafe instruction
initobj	initialize the value at an address	If the value comes from ldarg < i > of type t : pop newt astorej where j represent argument i as formalized in Fig. 7 for statement ldarg. Otherwise, unsafe instruction
isinst	test if an object is an instance of a class or interface	dup instanceof ifnel1 pop aconstnull gotol2 l1 : checkcast l2 : ...
ldelem[. < type >]	load an element of an array	< i > aload

Table B.5 (*continued*)

CIL	Description	JB Translation
ldelema	load address of an element of an array	If the value is used by ldobj aaload Otherwise unsafe instruction
ldfld	load field of an object	See Fig. 7
ldflda	load field address	If the value is used by ldobj getfield If the value is used by cpobj nop Otherwise unsafe instruction
ldlen	load the length of an array	arraylength
ldobj	copy a value from an address to the stack	If the value comes from ldloca, ldflda, *or* ldelema nop Otherwise, unsafe instruction
ldsfld	load static field of a class	getstatic
ldsflda	load static field address	If the value is used by ldobj getstatic If the value is used by cpobj nop Otherwise unsafe instruction.
ldstr	load a literal string	ldc

ldtoken	load the runtime representation of a metadata token	If the token < t > is a field reference:ldc "< t >"
		invokestaticReflection.getTypeHandle(LString;)LRuntimeTypeHandle;
		If the token < t > is a method reference:ldc "< t >"
		invokestaticReflection.getMethod(LString;)LMethod;
		Otherwise, unsafe instruction

ldvirtftn	load a virtual method pointer	pop ldc "< par >" invokestaticReflection.getMethod(LString;)Method;
mkrefany	push a typed reference on the stack	Unsafe instruction
newarr	create a zero -based one-dimensional array	anew_array or new_array
newobj	create a new object	See Fig. 7
refanytype	load the type out of a typed reference	Unsafe instruction
refanyval	load the address out of a typed reference	Unsafe instruction
rethrow	rethrow the current exception	aload < excindex > athrow
sizeof	load the size in bytes of a type	invokeReflection.sizeOf(LObject;)I
stelem[. < type >]	store an element of an array	< i > astore
stfld	store into a field of an object	See Fig. 7
stobj	store a value at an address	If the value comes from ldflda< f > of type t : putfield < f > Otherwise, unsafe instruction
stsfld	store a static field of a class	putstatic
throw	throw an exception	athrow
unbox[.any]	convert boxed value type to its raw form	invokestatic< boxedi > .unbox(< boxedi >)< ijvm >

- < ijvm > denotes the Java Virtual Machine representation of type < i > (I for integer, F for float, D for double, S for short, B for byte, J for long, C for char, and Z for Boolean),
- < typei > denotes the character < i > of the token < type > contained in CIL instructions,
- < f > denotes the name of a field,
- < type > denotes a CIL type,
- < t > denotes the name of a CIL metadata token,
- < length > denotes an integer value in CIL (e.g., index of an argument or target of a branching instruction),
- < par > denotes the signature of a method pointed by a ldftn CIL statement,
- < excindex > denotes the integer index of the local variable containing the name of a caught exception in JB, and
- < handler > denotes the offset where an exception handler begins.

References

[1] Wikipedia, List of tools for static code analysis, https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#Java. (Accessed 17 April 2019).
[2] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of POPL '77, ACM, 1977.
[3] F. Logozzo, Cibai: an abstract interpretation-based static analyzer for modular analysis and verification of Java classes, in: Proceedings of VMCAI '07, Springer, 2007.
[4] F. Spoto, The Julia static analyzer for Java, in: Proceedings of SAS '16, Springer, 2016.
[5] G. Costantini, P. Ferrara, A. Cortesi, A suite of abstract domains for static analysis of string values, Softw. Pract. Exp. 45 (2) (2015) 245–287.
[6] G. Barbon, A. Cortesi, P. Ferrara, E. Steffinlongo, DAPA: degradation-aware privacy analysis of Android apps, in: Proceedings of STM '16, 2016, pp. 32–46.
[7] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, The Java Virtual Machine Specification, Java SE 7 Edition, 1st edition, Addison-Wesley Professional, 2013.
[8] P. Ferrara, A. Cortesi, F. Spoto, CIL to Java-bytecode translation for static analysis leveraging, in: Proc. of the 6th Conference on Formal Methods in Software Engineering, FormaliSE 2018, Collocated with ICSE 2018, Gothenburg, Sweden, June 2, 2018, 2018, pp. 40–49.
[9] ECMA, Standard ECMA-335: Common Language Infrastructure (CLI), 2012.
[10] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, Lessons from building static analysis tools at Google, Commun. ACM 61 (4) (Mar. 2018).
[11] CLR to JVM, http://www.xmlvm.org/clr2jvm/. (Accessed 17 April 2019).
[12] M. Ameri, C.A. Furia, Why just boogie? Translating between intermediate verification languages, in: Proceedings of IFM '16, Springer, 2016.
[13] M. Bebenita, F. Brandner, M. Fähndrich, F. Logozzo, W. Schulte, N. Tillmann, H. Venter, SPUR: a trace-based JIT compiler for CIL, in: Proceedings of OOPSLA '10, ACM, 2010.
[14] Microsoft, FxCop, https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx. (Accessed 17 April 2019).
[15] Coverity, Coverity Prevent^{TM}, http://www.coverity.com/library/pdf/coverity_prevent.pdf. (Accessed 17 April 2019).
[16] Ndepend, http://www.ndepend.com. (Accessed 17 April 2019).
[17] Codemetrics, https://marketplace.visualstudio.com/items?itemName=kisstkondoros.vscode-codemetrics. (Accessed 17 April 2019).
[18] JetBrains, Resharper, https://www.jetbrains.com/resharper. (Accessed 17 April 2019).
[19] M. Barnett, K. Leino, W. Schulte, The spec# Programming system: an overview, in: Proceedings of CASSIS '04, 2004.
[20] F. Logozzo, M. Fähndrich, Static contract checking with abstract interpretation, in: Proceedings of FoVeOOS '10, Springer, 2010.
[21] Checkstyle, http://checkstyle.sourceforge.net. (Accessed 17 April 2019).
[22] Findbugs^{TM} – Find Bugs in Java Programs, http://findbugs.sourceforge.net/. (Accessed 17 April 2019).
[23] PMD, https://pmd.github.io/. (Accessed 17 April 2019).
[24] CodeSonar – Static Analysis SAST Software, https://www.grammatech.com/products/codesonar. (Accessed 17 April 2019).
[25] R. Atkey, D. Sannella, ThreadSafe: static analysis for Java concurrency, in: Electronic Comm. of the European Association of Software Science and Technology, vol. 72, 2015.
[26] WALA, http://wala.sourceforge.net/wiki/index.php/Main_Page. (Accessed 17 April 2019).
[27] T.A.S. Foundation, Apache Commons BCEL, https://commons.apache.org/proper/commons-bcel. (Accessed 17 April 2019).
[28] X. Rival, L. Mauborgne, The trace partitioning abstract domain, ACM Trans. Program. Lang. Syst. 29 (5) (2007) 26.
[29] L. Negrini, P. Ferrara, SARL: framework specification for static analysis, in: Proceedings of TAPAS '18, 2018.
[30] S. ECMA-335, Common Language Infrastructure (CLI), 4th edition, ECMA, 2006.