Abstract Interpretation of Java Bytecode in Sturdy

Stefan Marx
JGU Mainz
Germany

Sebastian Erdweg
JGU Mainz
Germany

Abstract

We develop a framework of definitional abstract interpreters for Java bytecode in Sturdy. Specifically, we provide a generic interpreter that abstractly executes Java bytecode but resorts to configurable analysis components for abstracting values and effects. From this, we can derive a concrete reference semantics for Java bytecode and sound abstract interpreters.

CCS Concepts

• **Software and its engineering → Automated static analysis**.

Keywords

abstract interpretation, static analysis, Java bytecode, control flow, data flow

ACM Reference Format:
Stefan Marx and Sebastian Erdweg. 2024. Abstract Interpretation of Java Bytecode in Sturdy. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '24), September 20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3678721.3686226

1 Introduction

Static analysis of Java bytecode has a long tradition in the programming-languages literature, and multiple state-of-the-art frameworks analysis frameworks target Java bytecode today. For example, Soot [10],[9], WALA [1], and OPAL [5] construct call graphs, perform pointer analyses, and support user-defined data-flow analyses. In this paper, we aim to develop a new analysis framework for Java bytecode following a different approach: definitional abstract interpretation.

A definitional abstract interpreter [3] conducts control-flow and data-flow analysis by abstractly executing the source program. While evaluating the source program, the definitional abstract interpreter triggers abstract effects (e.g., mutating the abstract heap) and yields abstract values (e.g., intervals). At the core of this approach is a *generic interpreter*, which defines the evaluator but is parametric in how effects and values are abstracted [6]. Keidel et al. [8] showed that the generic interpreter can not only be instantiated to define various static analyses, but also to derive a concrete reference semantics. Moreover, the instantiated analyses are sound relative to the concrete reference semantics if the abstractions for values and effects are sound, but independently of the generic interpreter. Definitional abstract interpreter also support configurable context-sensitive fixed-point algorithms [7] for fine-tuning the performance of an analysis.

Recently, Brandl et al. [2] presented an open-source analysis platform called *Sturdy* for definitional abstract interpreters and showed that it scales to real-world languages and real-world programs. Sturdy consists of reusable abstractions for values, effects, and fixed-points implemented in Scala. Brandl et al. used Sturdy to develop a generic interpreter for WebAssembly and used it to derive static analyses that outperformed the competition on real-world code in terms of precision and soundness.

In this paper, we employ Sturdy to develop an abstract definitional interpreter for Java bytecode. We define a generic interpreter for bytecode instructions that is parametric in how values and effects are represented and abstracted. In contrast to other JVM analysis frameworks, we do not need an analysis IR such as Jimple, but can describe an interpreter for the stack-based JVM directly. In doing so, we find that many abstractions provided by Sturdy are indeed reusable, including the abstractions for numbers, call frames, and exceptions. However, the mapping from JVM instructions to these abstractions is non-trivial and requires careful consideration. For example, the JVM features offset jumps, where the program counter is shifted, yet we were able to model these jumps as exceptions and reuse their abstraction. Another issues was that definitional abstract interpreters have never been used for an object-oriented language before. Therefore, we designed a new reusable representation and abstraction for objects, including their concrete reference semantics, which can be reused for other object-oriented languages in the future.

In summary, we make the following contributions:

- We define a generic interpreter for Java bytecode, from which we can derive a concrete reference semantics and abstract interpreters (Section 2).
- We design a reusable interface, concrete semantics, simple abstraction for Java-style objects (Section 3).
- We discuss lessons learnt and limitations (Section 4).

2 A Generic Interpreter for Java Bytecode

A generic interpreter is parametric in how values and effects are represented. If defined correctly, the generic interpreter can be instantiated with concrete values and effects to yield a concrete interpreter, but also to obtain different abstract interpreters. In this section, we gradually define a generic interpreter in Scala for Java bytecode. The full code is available open source.[1]

Numeric values and the stack. Figure 1 shows the basic template of the generic interpreter. The generic interpreter takes a type parameter `V`, which abstracts over the actual representation

[1] https://gitlab.rlp.net/smarx01/jvm-bytecode-in-sturdy-scala/-/tree/master/sturdy-jvm-bytecode

FTfJP '24, September 20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1111-4/24/09
https://doi.org/10.1145/3678721.3686226

```
trait GenericInterpreter[V]:
  val i32ops: IntegerOps[V]
  val f64ops: FloatOps[V]
  val stack: OperandStack[V]

  def eval(inst: Instruction): Unit = inst match
    case inst: POP =>
       stack.popOrAbort()
    case inst: DUP =>
      val v = stack.peekOrAbort()
      stack.push(v)
    case inst: BIPUSH =>
       stack.push(i32ops.integerLit(inst.value))
    case inst: FMUL =>
      val (v1,v2) = stack.pop2OrAbort()
      stack.push(i32ops.mul(v1,v2))
    case ...
```

Figure 1: A basic generic interpreter for Java bytecode.

of values. It then declares abstract fields such as `i32ops` and `stack`, which abstract over operations on values and effects. For example, `i32ops` must implement `IntegerOps[V]`, which declares a multitude of integer operations. When instantiating the generic interpreter, definitions of these fields must provide actual implementations of these interfaces.

```
trait IntegerOps[V]:
  def mul(v1: V, v2: V): V
  def bitAnd(v1: V, v2: V): V
  ...
```

All interfaces for values and effects used in Section 2 are part of the Sturdy library of value and effect components. This also includes implementations of these interfaces. For example, there is an implementation `IntegerOps[Int]` to be used in the concrete interpreter as well as an implementation `IntegerOps[Interval]` to be used in an interval analysis. This reuse drastically reduced the development effort for us.

The JVM is a stack-based machine: instructions obtain their arguments and store their results on an operand stack. While other analysis frameworks for the JVM typically rely on a stack-free IR such as Jimple, we find definitional abstract interpreters can be used to operate on the actual instructions. Figure 1 shows how the generic interpreter interacts with the stack, where values are pushed and popped as a side effect. These operations are defined as part of the interface of the stack effect:

```
trait OperandStack[V] extends Effect:
   def push(v: V): Unit
   def popOrAbort(): V
   def peekOrAbort(): V
```

Since the shape of the operand stack is statically decidable, this effect does not lose precision during analysis.

Local and static variables. Local variables and method parameters are stored in a call frame. In contrast, static variables are global variables that are accessible from anywhere. Figure 2 shows how we can capture both kinds of variables in the generic interpreter.

```
trait GenericInterpreter[V]:
  ...
  val frame: CallFrame[Int, V] // locals are indexed
  val staticVars: Store[(ObjectType,String), V]

  def eval(inst: Instruction): Unit = inst match
    case inst: ISTORE =>
      val v = stack.popOrAbort()
      frame.setLocal(inst.lvIndex, v)
    case inst: FLOAD =>
      val v = frame.getLocalOrFail(inst.lvIndex)
      stack.push(v)
    case inst: GETSTATIC =>
      lazyInitialize(inst.cls)
      val v = staticVars.read((inst.cls, inst.name))
      stack.push(v)
    case inst: PUTSTATIC =>
      lazyInitialize(inst.cls)
      val v = stack.popOrAbort()
      staticVars.write((inst.cls, inst.name), v)
```

Figure 2: Local and static variables use different effects.

The key observation is that they are managed by different effects. We introduce a call-frame effect for local variables, yet use a store effect for static variables.

The code in Figure 2 highlights another advantage of definitional abstract interpretation. Static variables are complex in the JVM, because their initialization happens on demand and only once. We can gracefully support this feature in our generic interpreter by lazily invoking the static initializer of a class before accessing its static fields. This way, the static initializer runs before we read a static variable, but only when it is actually necessary.

Method invocation. We discuss the invocation of static methods next. The invocation of virtual methods requires dynamic dispatch, which depends on the object representation we introduce in Section 3. However, once the target method of dynamic dispatch has been identified, its execution resembles that of static methods.

Figure 3 shows how the generic interpreter implements method invocation. We first find the static method named `inst.name` in `inst.cls` with signature `inst.params`. We then pop one argument value from the stack for each parameter of the method and use the auxiliary function `call`. The `call` function initializes the local variables of the method with default values and joins them with the arguments in an indexed sequence. We run the instructions of the method in a new call frame, which contains the method's parameters and local variables. Moreover, each method has its own frame in the operand stack as to clearly distinguish which values on the stack belong to which method. Function `run` executes all given instructions using `eval`, but also handles jumps within the method as we will discuss below. Once `run` finishes, we grab the method's return value from the stack unless the return type is `void`. Then, the operand frame and call frame are discarded and method invocation is complete.

```
def eval(inst: Instruction): Unit = inst match
  case inst: INVOKESTATIC =>
    val mth = findMethod(inst.cls,
       inst.name, inst.params)
    val args = stack.popNOrAbort(inst.params.size)
    val ret = call(mth, args)
    if (!mth.returnType.isVoidType)
      stack.push(ret)

def call(mth: Method, args: Seq[V]): V =
  val locals = mth.localVariables.map(defaultValue)
  val vars = (args ++ locals).zipWithIndex.map(_.swap)
  frame.withNewCallFrame(vars) {
    stack.withNewOperandFrame() {
      run(0, mth.instructions, mth)
      if (mth.returnType.isVoidType)
        i32ops.integerLit(-1)
      else
        stack.popOrAbort()
    }
  }
```

Figure 3: Generic interpretation of static method invocation.

Jumps and exceptions. The JVM encodes control flow constructs through conditional and unconditional jumps within a method. Moreover, exceptions can interrupt the regular control flow of a method. We model both features through exceptions in the generic interpreter.

Figure 4 shows how the generic interpreter uses Sturdy's `except` effect to model jumps and throws. When evaluating an unconditional `GOTO` instruction, the generic interpreter uses `except` to issue an exceptional control flow with the target program counter (PC). For conditional jumps like `IFEQ`, the throw only occurs when the condition is met. During abstract interpretation, the `branchOps` abstraction joins the effect of `throws` with the empty effect (for "skip") when the condition is undecidable. In this case, the abstract interpreter continues a sequential analysis while also performing the jump, and it joins both results. Finally, the `ATHROW` instruction takes a throwable object from the stack and throws it.

Sturdy's exception effect is used to model exceptional control flow of the JVM. This becomes clear when studying the `run` method, which executes method bodies. Here, we use `except.tryCatch` to catch and react to thrown exceptions. For `Exc.Jump`, we simply continue execution at the targeted PC. But for `Exc.Throw`, we first have to search for an exception handler in the current method. If found, we continue the run at the exception handler. Otherwise, we escalate the exception.

Summary. We have presented key excerpts of a generic interpreter for Java bytecode. In doing so, we were able to reuse encodings of values and effects provided by Sturdy. One important omission so far is the representation of objects and operations on them, which we address in the subsequent section.

```
enum Exc[V]:
  case Jump(pc: Int)
  case Throw(exception: V)

// in GenericInterpreter:
val eqOps: EqOps[V, V]
val branchOps: BooleanBranching[V, Unit]
val except: Except[Exc[V], EV]

def eval(inst: Instruction): Unit = inst match
  case inst: GOTO => //unconditional jump
    except.throws(Exc.Jump(pc + inst.offset))
  case inst: IFEQ => //conditional jump
    val v = stack.popOrAbort()
    val isEq = eqOps.equ(v, i32ops.integerLit(0))
    branchOps.boolBranch(isEq) {
      except.throws(Exc.Jump(pc + inst.offset))
    } { /* skip */ }
  case inst: ATHROW =>
    val thrown = stack.popOrAbort()
    except.throws(Exc.Throw(thrown))

def run(pc: Int, is: Map[Int, Instruction], mth:Method)=
  except.tryCatch {
    val currInst = is(pc)
    eval(currInst, mth, pc)
    val nextPC = currInst.indexOfNextInstruction(pc)
    run(nextPC, is, mth)
  } {
    case Exc.Jump(targetPC) =>
      run(targetPC, is, mth)
    case Exc.Throw(exc) =>
      getExceptionHandler(mth, pc, exception) match {
        case Some(h) =>
          stack.push(exception)
          run(h.handlerPc, is, mth)
        case None =>
          except.throws(Exc.Throw(exception))
  }
```

Figure 4: Using exceptions for jumps and JVM exceptions.

3 Object Representation and Operations

The representation of and operations on objects is crucial in any JVM analysis. To the best of our knowledge, definitional abstract interpreters have never been used to analyze an object-oriented language before. In particular, Sturdy does not provide support for objects. This paper contributes the design and implementation of a reusable representation for objects for Sturdy. We also added support for mutable arrays and their operations, but elided arrays from the remaining presentation.

3.1 Concrete Object Operations

Figure 5 shows our language-independent interface for interoperating with statically typed objects. The interface is highly parametric, in particular:

```
trait ObjectOps[Cls,Mth,AllocSite,FieldSite,OV,V,Ty]:
  type FID = (Cls,String)
  type MID = (Cls,String)
  def create(cls: Cls, s: AllocSite,
             vals: Seq[(FID,FieldSite,V)]): OV
  def getField(obj: OV, f: FID): V
  def setField(obj: OV, f: FID, v: V): Unit
  def invoke(obj: OV, m: MID, args: Seq[(V,Ty)])
            (call: (OV,Mth,Seq[V]) => V): V
```

Figure 5: A generic representation of object operations.

- `Cls` represents the class/type of the object. For the JVM, this is the object's classfile.
- `Mth` is the method representation used by `Cls`.
- `AllocSite` is the (context-sensitive) allocation site of the object. This is used by analyses to collapse objects that should be treated the same.
- `FieldSite` is the (context-sensitive) allocation site for fields. This is used for configuring field-sensitivity.
- `OV` is the representation of object values.
- `V` is the representation of field and argument values.
- `Ty` is the static type of method arguments, used in `invoke` for resolving overloaded methods.

In `ObjectOps`, function `create` yields a new object value, given a class, an allocation site, and a sequence of field values. Given an object value, `getField` and `setField` can be used to read and write field values for field `f` in declaring class `cls`. Finally, we can use `invoke` to perform dynamic dispatch: Given an object value, the name of a method, and a sequence of argument values with their static types, `invoke` finds the target method(s) and calls them. During analysis, this can lead to multiple method invocations, using `call` on each target method and joining their results.

It is instructive to study the concrete reference semantics of `ObjectOps`. We provide such a concrete semantics in Figure 6 for the JVM. Specifically, we model objects as immutable containers `Obj` that (i) are identified by an object ID, (ii) know their classfile, and (iii) store an address for each field. The values of fields are not part of the object itself, but stored in the heap. This way, we can mutate field values without having to exchange the object representation. When creating a new object, we first allocate an object ID and an address for each field, ignoring the allocation sites, which are only necessary during abstract interpretation. We then store the initial value of each field in the store. The allocation functions `allocO` and `allocF` are parameters to the `ConcreteObjectOps`, as is the `store` for field values. The remaining object operations are straightforward.

3.2 Generic Interpretation with Object Operations

We provided an interface for object operations. With this interface, we can now extend our generic interpreter to support instructions for creating objects, accessing fields, and invoking virtual methods.

Figure 7 shows how the generic interpreter uses `ObjectOps` to implement object creation and virtual calls. But first, we require a

```
type OID = Int
type Addr = Int
type FID = (ClassFile, String)
type FT = FieldType

case class Obj(oid: OID, cls: ClassFile,
               fields: Map[FID, Addr])

class ConcreteObjectOps[V, AllocSite, FieldSite]
    (allocO: Allocation[OID],
     allocF: Allocation[Addr],
     store: Store[Addr, V])
  extends ObjectOps[ClassFile, Method,
                    AllocSite, FieldSite, Obj, V, FT]:
  def create(cls: ClassFile, s: AllocSite,
             vals: Seq[(FID,FieldSite,V)]): Obj =
    val oid = allocO()
    val fields = vals.map { (f,site,v) =>
      val addr = allocF()
      store.write(addr, v)
      (f,addr)
    }.toMap
    Object(oid, cls, fields)
  def getField(obj: Ob, f: FID): V =
    store.read(obj.fields(f))
  def setField(obj: Obj, f: FID, v: V): Unit =
    store.write(obj.fields(f), v)
  def invoke(obj: Obj, m: (ClassFile,String),
             args: Seq[(V,FT)])
            (call: (Obj,Mth,Seq[V]) => V): V =
    val mthSig = args.map(_._2)
    val mth = obj.cls.findNextMethod(m, mthSig)
    call(obj, mth, args.map(_._1))
```

Figure 6: A concrete reference semantics for `ObjectOps`.

specialized implementation of `ObjectOps` for JVM classfiles, methods, and field types, which we reuse from OPAL [5]. Then, we can interpret the `NEW` instruction:

(1) Fetch the classfile from the `NEW` instruction.
(2) The instruction serves as allocation site for object IDs.
(3) Find relevant field declarations in the class hierarchy.
(4) Compute the initial field values.
(5) Use `objectOps` to create an object value and push that object value on the stack.

For virtual calls, we fetch the number of required arguments from the stack and combine them with the static parameter types, used for overload resolution. Additionally, we fetch the receiver object from the stack. Then, we use `objectOps` to invoke the method of the given name. For each found target method `mth`, we use `call` from Figure 3 to run the method's body, but adding the receiver object `o` to the arguments `as`.

Note that our implementation currently does not support all intricacies of the JVM for virtual calls. For example, we do not correctly identify all error cases and do not support synchronization monitors. But since the resulting generic interpreter code resembles

```
val objectOps: ObjectOps[ClassFile, Method,
    InstructionSite, FieldSite, V, V, FieldType]

def eval(inst: Instruction): Unit = inst match
  case inst: NEW =>
    val cf = inst.objectType
    val site = InstructionSite(inst)
    val allFields = ... // fields in cf and superclasses
    val fields = allFields.map { fld =>
      ( (fld.declaringClass, fld.name)
      , FieldSite(site, fld)
      , defaultValue(fld.fieldType) )
    }
    stack.push(objectOps.create(cf, site, fields))
  case inst: INVOKEVIRTUAL =>
    val args = stack.popNOrAbort(inst.params.size)
    val argsTyped = args.zip(inst.paramTypes)
    val obj = stack.popOrAbort()
    val (cls, name) = (inst.cls, inst.name)
    val ret = objectOps.invoke(obj, cls, name, args) {
      (o, mth, as) => call(mth, o +: as)
    }
    if (!mth.returnType.isVoidType){
      stack.push(ret.get)
    }
def call(mth: Method, args: Seq[V]): V =
  ... // defined in Figure 3
```

Figure 7: Generic interpreter for dynamic dispatch and `NEW`.

```
type FT = FieldType
case class TyObj(cls: ClassFile, mayNull: Boolean):
  def join(that: TyObj): TyObj =
    TyObj(this.cls.leastSuperType(that.cls),
          this.mayNull || that.mayNull)

class TypeObjectOps[V, AllocSite, FieldSite]
  extends ObjectOps[ClassFile, Method,
                    AllocSite, FieldSite, TyObj, V, FT]:
  def create(cls: ClassFile, s: AllocSite,
             vals: Seq[(FID,FieldSite,V)]): TyObj =
    TyObj(cls, false)
  def getField(obj: TyObj, f:FID): V =
    val cls = f._1
    topValue(cls.getFieldType(f._2))
  def setField(obj: TyObj, f: FID, v: V): Unit =
    { } // do nothing
  def invoke(obj: TyObj, m:(ClassFile,String),
             args: Seq[(V,FT)])
            (call: (TyObj,Mth,Seq[V]) => V): V =
    val mthSig = args.map(_._2)
    val supMth = obj.cls.findNextMethod(m, mthSig)
    val subMths = for (sub <- obj.cls.subClasses
                       mth <- sub.findMethod(m, mthSig))
                  yield mth
    val mths = supMth +: subMths
    val argVals = args.map(_._1)
    mapJoin(mths, mth => call(obj, mth, argVals))
```

Figure 8: An abstract semantics for `ObjectOps` using types.

a standard interpreter closely, we believe we can extend the generic interpreter accordingly.

3.3 Abstract Object Representations and Operations

By choosing abstract instantiations for `ObjectOps` and the other value and effect interfaces, the generic interpreter becomes an abstract interpreter. Moreover, the generic interpreter is sound by construction, it is sufficient to reason about value and effect instantiations [6]. Here, we illustrate how to define an abstraction for `ObjectOps`.

Figure 8 shows an abstract semantics for `ObjectOps` that implements a Class Hierarchy Analysis (CHA) [4]. Specifically, we use `TyObj` to represent abstract objects with their type `cls` and nullability information `mayNull`. When creating a new object, the abstract object carries the precise run-time type. But later during analysis, abstract objects may need to be joined such that their type represents an upper bound on the actual run-time type. When reading a field, we take the declared type of the field and produce the top abstract value for that type. For example, `TyObj(cls, true)` is the top abstract value for reference type `cls`. For `invoke`, we analyze the class hierarchy to find all implementations of `m` declared in `cls` compatible with the object type `obj.cls`. Specifically, we consider the closest implementation of `m` in `obj.cls` or one of its supertypes, and also all implementations of `m` in subclasses of `obj.cls`. We then use Sturdy's `mapJoin(xs, f)` function, which runs `f` on each element in `xs` and joins the results and effects of `f`. This way, we consider all potential call targets in `mths`.

4 Limitations and Improvements

In this section, we want to discuss the limitations of our generic interpreter for Java bytecode, which we are addressing in ongoing work. We also discuss lessons learnt about using Sturdy and, in particular, how we have been able to improve the core of Sturdy, better supporting new languages.

Limitations of the bytecode interpreter. We apply the methodology of definitional abstract interpretation to arrive at a sound analysis for Java bytecode. However, some features of the JVM make this inherently difficult, which is why many static analyses forsake soundness. Instead, we opt for a sound yet incomplete approach. For example, we only support selected few native functions in our implementation and abort analysis for unsupported ones. Our interpreter is also incomplete for reflection and dynamic class loading. Hence, our analysis fails to apply to realistic projects so far. But, we believe supporting many native functions and other complex features is feasible, because we can often implement their behavior as part of the generic interpreter. For example, we implemented `arraycopy` in terms of the `ArrayOps` interface, which we introduced to abstract from array representation and operations. Therefore, an analysis that instantiates our generic interpreter does

not have to define the meaning of array copy, but can reuse the generic interpretation.

Improvements to Sturdy. While implementing the generic interpreter for JVM Bytecode in Sturdy, we have learned multiple lessons and used them to improve the platform. These improvements not only help for Java bytecode, but can be reused for other languages to be analyzed in the future. First, we added new value interfaces for arrays and objects with their operations, which did not exist. Second, we added an interface for run-time type checking, i.e., `instanceOf`. Third, we improved a few existing value interfaces: We added support for constructing `NaN` floating-point values and reworked how division by zero is treated for integer values. Specifically, Sturdy has never been used to analyze a language with recoverable division-by-zero errors. For example, the Sturdy interpreter for WebAssembly (correctly) aborts execution when the denominator is zero [2]. Instead, our interpreter for Java bytecode issues an exception, which can be caught. Overall, we found that Sturdy was well-equipped to support Java bytecode apart from these necessary improvements.

5 Related Work

In this section we want to discuss some of our design decisions and their benefits in comparison to other state-of-the-art analysis frameworks. We will primarily focus on three aspects.

A major difference is that we analyze the bytecode directly in its original stack-based form. Since all operations on the stack are decidable, we don't add any imprecision while doing so. Specifically, the definitional abstract interpreter precisely tracks how a pushed value flows to an instruction that pops it. Often, frameworks use an intermediate representation like Jimple [10], which lifts bytecode to three-address code with variables. For example, Soot [9] processes Jimple and WALA [1] processes a similar intermediate representation called WALA IR.

Another aspect that distinguishes our work is that we are not dependent on a pre-constructed control flow graph. Instead, we analyze bytecode by interpreting it in an abstract machine. In fact, our definitional abstract interpreter does not even construct a control-flow or call graph, but explores the control-flow eagerly on-the-fly. This made it relatively simple to support branching and exceptional control flow. To ensure termination, Sturdy provides a library of fixed-point combinators, which track repeating abstract interpretations and compute a fixed point [7].

Another significant advantage of our approach is that it is easy to add new abstractions of values and effects. The modularity of these abstractions makes writing new analyses quick and easy.

Finally, our work confirmed that Sturdy's abstract interpreters can be constructed modularly from analysis components for values and effects. This significantly reduced our workload for building an abstract interpreter for Java bytecode. While other frameworks like OPAL [5] also provide modular analysis components for analyzing Java bytecode, our framework also provides these modular components for easy extension to other languages, which we have not observed in other frameworks. Like already shown in this paper, we were able to reuse effect and value abstractions that have been developed for WebAssembly [2]. Indeed, we only had to design a few new abstractions, including those for objects and arrays. These are now part of the core library of Sturdy and can be reused for other languages in the future.

6 Conclusion

In this work, we introduced a novel approach for analyzing an object-oriented language using definitional abstract interpreters. We showed how to construct a generic interpreter by taking advantage of value and effect interfaces already provided by our analysis platform Sturdy. We also showed how to extend Sturdy with new value interfaces to handle object values. Furthermore, we demonstrated that Sturdy provides lots of reusable abstractions for values and effects, and we added new abstractions for objects and arrays to Sturdy.

Acknowledgement

We thank David Klopp, Armand Lego, André Pacak, Prashant Kumar and the anonymous reviewers for their valuable feedback. This work is supported by the European Research Council (ERC) under the European Union's Horizon 2023 (Grant Agreement ID 101125325).

References

[1] [n. d.]. *WALA - T.J. Watson Libraries for Analysis.* https://github.com/wala/WALA

[2] Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. 2023. Modular Abstract Definitional Interpreters for WebAssembly. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:28. https://doi.org/10.4230/LIPICS.ECOOP.2023.5

[3] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 1–25. https://doi.org/10.1145/3110256

[4] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 952)*, Walter G. Olthoff (Ed.). Springer, 77–101.

[5] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular collaborative program analysis in OPAL. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 184–196. https://doi.org/10.1145/3368089.3409765

[6] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

[7] Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. 2023. Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. *Proc. ACM Program. Lang.* 7, ICFP (2023), 955–981. https://doi.org/10.1145/3607863

[8] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

[9] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie J. Hendren. 2011. The Soot framework for Java program analysis: a retrospective. https://api.semanticscholar.org/CorpusID:11439274

[10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

Received 2024-06-26; accepted 2024-07-24