A Termination Analyzer for Java Bytecode Based on Path-Length

FAUSTO SPOTO
Università di Verona
and
FRED MESNARD and ÉTIENNE PAYET
IREMIA, LIM, Université de la Réunion

It is important to prove that supposedly terminating programs actually terminate, particularly if those programs must be run on critical systems or downloaded into a client such as a mobile phone. Although termination of computer programs is generally undecidable, it is possible and useful to prove termination of a large, nontrivial subset of the terminating programs. In this article, we present our termination analyzer for sequential Java bytecode, based on a program property called *path-length*. We describe the analyses which are needed before the path-length can be computed such as sharing, cyclicity, and aliasing. Then we formally define the path-length analysis and prove it correct with respect to a reference denotational semantics of the bytecode. We show that a constraint logic program P_{CLP} can be built from the result of the path-length analysis of a Java bytecode program P and formally prove that if P_{CLP} terminates, then P also terminates. Hence a termination prover for constraint logic programs can be applied to prove the termination of P. We conclude with some discussion of the possibilities and limitations of our approach. Ours is the first existing termination analyzer for Java bytecode dealing with any kind of data structures dynamically allocated on the heap and which does not require any help or annotation on the part of the user.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics; program analysis*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, Java, Java bytecode, termination analysis

ACM Reference Format:
Spoto, F., Mesnard, F., and Payet, É. 2010. A termination analyzer for Java bytecode based on path-length. ACM Trans. Program. Lang. Syst. 32, 3, Article 8 (March 2010), 70 pages.
DOI = 10.1145/1709093.1709095 http://doi.acm.org/10.1145/1709093.1709095

Authors' addresses: F. Spoto, Dipartimento di Informatica, Università di Verona, Strada le Grazie 15, 37134 Verona, Italy; email: fausto.spoto@univr.it; F. Mesnard, É. Payet, IREMIA, LIM, Université de la Réunion, 15 avenue René Cassin, BP7151, 97715 Saint-Denis Messag, Cedex 9, France.

DOI 10.1145/1709093.1709095 http://doi.acm.org/10.1145/1709093.1709095

1. INTRODUCTION

It is well-known that a general procedure for determining which computer programs terminate does not exist for Turing-complete programming languages [Turing 1936]. Nevertheless, it is becoming ever more important to prove that programs terminate. This is because software is used in critical systems where nontermination might lead to disaster. Moreover, software is increasingly deployed in embedded tools such as mobile phones. If a program downloaded into a mobile phone does not terminate, the phone might require a tedious shutdown; worse, users might complain to the originator of the software or to the phone company itself, which accounts for extra costs on their part, or might decide not to download software anymore. The software industry is paying more and more attention to software quality and would like to issue a *certificate* attesting to that quality. A proof of termination about the programs in the software should definitely be part of the certificate. Moreover, the compiler industry is interested in termination proofs. For instance, the latest version of Sun's Java compiler rejects nonterminating class initializers; however, the test for nontermination is so rudimentary that virtually all nonterminating initializers escape that test. For these reasons, termination is considered as a challenge for current software verification [Leavens et al. 2007].

Programmers can often argue for the termination of the programs they write. This means that termination of computer programs can be proved by humans, at least for a large class of programs. However, programmers are often very erroneously convinced of the termination of programs which are later found to diverge in some special or unexpected cases: almost everyone has had the experience of having to stop a program which apparently was not terminating. This means the human proofs of termination are error-prone and generally unreliable. This problem becomes more acute for modern programming languages, such as the object-oriented ones, especially if they are low-level languages with very complex semantics.

Java bytecode [Lindholm and Yellin 1999] is a low-level, object-oriented programming language, usually resulting from the compilation of a *source* Java program. It can be seen as a machine-independent, type-safe, object-oriented, imperative assembly language. Although it was born both with and for Java, it is now also used as a compilation target for other programming languages. The Java bytecode available on the Internet or downloaded into mobile phones is often provided as a set of Java bytecode classes without the corresponding source code. The source code is not made available for one or more reasons: because of commercial choice, to shorten the download time, or because source code does not even exist since the bytecode is the result of software transformations or specializations. These considerations entail that termination proofs for Java bytecode software have real industrial interest. Moreover, one can prove the termination of a Java source program by proving the termination of the derived Java bytecode (assuming the compiler to be correct), while the converse is false: many Java bytecode programs do not directly correspond to a Java program.

Previous research has developed automatic *termination analyses*, that is, formal techniques for proving, automatically, the termination of large classes of computer programs. When these analyses prove termination, then the analyzed program actually terminates, while the converse is generally false. Although there is a variety of proposed techniques, the underlying common idea is that of finding some well-founded measure, called in turn *norm*, *ranking function*, or *level mapping*, that strictly decreases along loops or in recursive calls.

Most of the work on termination analysis has been applied to term rewriting systems, functional and logic programming languages, whose semantics is typically simple and well understood. Proofs of termination for imperative programs that use dynamic data structures are much more complex than the corresponding proofs for functional or logical languages which do not have destructive updates. In order to foresee the possible effects of destructive updates, it is important to compute information about the shape of the heap of the system at runtime. Namely, *sharing* and *cyclicity* of data structures play an important role in imperative programs, while they are forbidden or practically never used in functional and logical languages. Since cyclicity can lead to nontermination of some iterations over the data structures, it must be taken into account for a correct termination analysis. It has been proved that sharing adds to the power of LISP programs since it allows one to write computationally cheaper algorithms [Pippenger 1997]. No similar result is known for cyclicity. Nevertheless, the extensive use of sharing and cyclicity in current Java programs entails that a realistic static analysis must take them into account.

Things become still more complex with object-oriented languages where dynamic dispatch, inheritance, instance and class initializations must be taken into account. Cyclicity becomes omnipresent there, for instance, all exceptions are cyclical in Java. If we consider the Java bytecode language, its low-level nature presents further challenges, such as the unstructuredness of the code and the presence of an operand stack of variable height. For instance, this requires the tracking of precise definite aliasing between local variables and stack elements, which is not the case for high-level languages. Without such information (or similar), abstract domains and static analyses, which are sufficiently precise for high-level languages, might not be precise enough for a low-level language [Logozzo and Fähndrich 2008].

It therefore follows that an automatic technique for proving the termination of Java bytecode programs is a long way from being a simple *extension* of similar techniques already existing for functional and logical languages. To the contrary, it requires a set of preliminary static analyses, such as sharing, cyclicity, and aliasing analyses, and strict adherence to all the details of the semantics of the language.

For this reason, we have recently defined an abstract analysis, called *path-length*, which uses preliminary sharing, cyclicity, and aliasing analyses to build an overapproximation (hence, safe approximation) of the maximal length of a path of pointers that can be followed from each program variable [Spoto et al. 2006]. This may be seen as an extension to data structures of the *linear restraints* of Cousot and Halbwachs [1978].

In this article, we make the following contributions.

(1) We define the path-length analysis for sequential Java bytecode, dealing with any kind of data structures, and prove that it is formally correct using the abstract interpretation framework [Cousot and Cousot 1977].
(2) We define how a Constraint Logic Program (CLP) program is derived from the path-length analysis of a Java bytecode program and prove that if the derived CLP program terminates then also the original Java bytecode program terminates.
(3) We describe our implementation of a termination analysis for sequential Java bytecode, based on path-length, inside the JULIA analyzer [Spoto 2008a], coupled with the BINTERM termination prover. It is a fully automatic system able to scale up to programs of 1200 methods, including all the analyses necessary to build the path-length constraints. This shows the potential of both JULIA and BINTERM.

In this article we only consider a nontrivial subset of Java bytecode so that, for instance, point 2 is limited to that subset. However, note that this is standard in the analysis of Java bytecode since the chosen bytecodes are representative of a large family of bytecodes (namely, they include those manipulating the heap) and the missing bytecodes perform tedious stack manipulations or deal with concurrency (that we do not handle). By considering all bytecodes, we would just make the article clumsy.

We stress the fact that the implementation is not a prototype but a robust and reliable system, resulting from many years of programming work. It includes class, `null` pointer, initialization, sharing, cyclicity, aliasing and path-length analyses, and it with all constructs of Java bytecode, including the jsr and ret instructions, as well as with exceptions. It has been tested on very large programs (up to 10,000 methods) and extensively debugged; it is also used by a big industrial company in the USA for information-flow analysis of very large programs. To the best of our knowledge, it is the first fully automated implementation of a termination analyzer for full sequential Java bytecode with no invention of adhoc algorithms for dealing with specific complex programs. Moreover, it is the first termination analysis for imperative programs able to deal, automatically and with satisfying precision, with any kind of data structures dynamically allocated in memory.

Two lines of works are strictly related to ours and deserve some discussion.

— In Albert et al. [2007a, 2008], it has been shown how the results of the path-length analysis can be used to translate the analyzed imperative program into a CLP which can then be fed to a termination prover for logic programs. In the same spirit, path-length has also been used in Albert et al. [2007b] to infer upward approximations of the computational cost of Java bytecode methods. They use it to translate imperative programs into constraint logic programs over which cost analysis is performed. Our translation into CLP programs is not identical to that used in these papers, but Albert et al. [2007a, 2008] remain the closest to our work. Note, in particular, that Albert et al. [2007a] has been published before the first submission of our paper. We have received the benchmarks analyzed in Albert et al. [2007a, 2008] from the authors of those papers; their analysis with our tool is shown in Figure 16.

—The Terminator system [Cook et al. 2006b] proves termination of C programs. A crucial innovation with respect to termination consists in its use of transition invariants [Podelski and Rybalchenko 2004b], which are computed using techniques for least-fixpoint calculation and abstraction. Transition invariants enable the use of a ranking function generator for simple-while programs, which can be implemented by constraint solving [Podelski and Rybalchenko 2004a]. Termination is proved over primitive types without dynamic data structures. This is the main difference from our work, which is in principle able to deal with any data structure dynamically allocated in the heap. Terminator uses model-checking to explore the set of reachable states of the program. The use of model-checking allows one to also test concurrent programs. The distinguishing feature of Terminator is its ability to improve the analysis by exploiting counterexamples found during the model-checking [Cook et al. 2005]. This feature, which is missing in our work, can lead to very precise analyses, sometimes at the expense of efficiency. Terminator can deal with pointers in the sense that it models dereferencing. However, it does not deal with iterations over dynamic data structures [Cook et al. 2006a, p. 425]. It has been successfully used for the verification of operating systems drivers of nontrivial size [Cook et al. 2006a]. The weak modeling of the heap in Terminator has been overcome in Berdine et al. [2006] where termination of C programs with lists is proved by using the shape analysis in Distefano et al. [2006], which is based on separation logic [Reynolds 2000; Ishtiaq and O'Hearn 2001]. Their work has some similarities to ours since they build a linear constraint from the program by using the shape analysis to gather information about the size of the lists. However, they do not support functions, as the underlying shape analysis. They claim that their work can be applied to many data structures, but they only consider linked lists; the derivation of linear constraints from the shape analysis is not proved correct. Note that they are based on a separation logic for lists only and also that a more advanced version of that logic [Berdine et al. 2007a] still considers flavors of lists only as well as the interprocedural shape analysis in Gotsman et al. [2006]. Their work has been generalized in Berdine et al. [2007b], so that termination with lists is an instance of a generic framework which proves a well-founded variance of some variables at a specific program point. The generalization does not affect the results about the data structures which can be modeled in the heap during the shape analysis. Compared to our work, we remark that we consider every kind of data structure in the heap. Although it is true that more advanced shape analyses can determine the shape of any data structure in memory, not just lists, there is no mention in the preceding works of the translation of the results of such shape analyses into numerical constraints that can later be used to prove the termination of the program. That is, those papers miss a formal definition of how the linear constraints are built when a destructive update modifies some data structure, not just lists (see *putfield* in Definition 37), as well as a formal definition of how the linear constraints are built for method calls that might modify data structures in the heap (see *extend* in Definition 44). Moreover, we provide formal proofs of the correctness of the construction of those numerical constraints, while this is not the case in the papers just discussed. This is far from being a detail. As the reader can check, those two definitions are the most complex in this article and their correctness proof requires careful and nontrivial arguments. For a pratical comparison with our tool, we have analyzed three of the programs in Cook et al. [2006a]. Namely, program `Numerical1` in Figure 16 is the program in Figure 3 of Cook et al. [2006a], program `Numerical2` is the program in Figure 11 of Cook et al. [2006a], and program `Numerical3` is the diverging program in Figure 7 of Cook et al. [2006a]. The same paper contains a utility function of a Windows device driver (Figure 1 of Cook et al. [2006a]) and analyzes a set of Windows device drivers (in its Figure 12). We cannot analyze such drivers because there is no way of writing Windows device drivers in Java. The same paper analyzes the Ackermann function (also analyzed in Figure 16), coupled with a program that uses pointers to integers, which do not exist in Java (Figure 4 of Cook et al. [2006a]). The benchmarks analyzed in Berdine et al. [2006] are all loops of Windows device drivers which, again, we cannot analyze. The simple iteration over a list in Figure 5 of Berdine et al. [2006] is included in the analysis of `List` in Figure 16.

The rest of this article is organized as follows. Section 2 gives an overview of our analyzer through its application to some programs, hence showing how it deals correctly with some of the subtlest aspects of the semantics of the language. Section 3 defines the syntax of a small but nontrivial subset of the Java bytecode that we use in our definitions and proofs. Section 4 describes all the preliminary analyses that we perform before the path-length analysis. Section 5 defines an operational and an equivalent denotational semantics of our subset of the Java bytecode. Section 6 defines the path-length analysis and proves it correct with respect to the denotational semantics of Section 5. Section 7 defines the translation from Java bytecode into CLP over path-length and proves that, if the CLP program terminates, then the original Java bytecode also terminates. Section 8 reports some experiments with our analysis. Section 9 discusses related works. Section 10 discusses limitations, future directions of research, and then concludes. Most of the proofs are available in an electronic appendix, available in the ACM Digital Library.

2. EXAMPLES OF OUR TERMINATION ANALYSIS

This section presents examples of termination analysis with our tool. All examples can be tested online through a Web interface [Spoto et al. 2008]. The input of the analysis is a Java bytecode program P, its output is an enumeration of its methods, divided into those whose calls in P definitely terminate, those whose calls in P might diverge because of a loop inside their code (methods that

```
public class Sharing {

  private Sharing next;

  public Sharing(Sharing next) {
    this.next = next;
  }

  public void expand(Sharing other) {
    Sharing cursor = this;
    while (cursor != null) {
      other.next = new Sharing(null);
      other = other.next;
      cursor = cursor.next;
    }
  }
}
```

```
public void expand(Sharing);
 0:  aload_0
 1:  astore_2
 2:  aload_2
 3:  ifnull 31
 6:  aload_1
 7:  new Sharing
 10: dup
 11: aconst_null
 12: invokespecial
      Sharing.<init>(Sharing):void
 15: putfield next
 18: aload_1
 19: getfield next
 22: astore_1
 23: aload_2
 24: getfield next
 27: astore_2
 28: goto 2
 31: return
```

Fig. 1. An example where sharing is needed to model the effects of a destructive update.

introduce nontermination), and those whose calls in P might diverge but only because they call one of the previous diverging methods (methods that *inherit* nontermination).

Let us start from an example which shows the problems induced by the destructive updates. The program on the left of Figure 1 implements a simple linked list with an expand method that scans the list corresponding to the this object and expands the first node of the parameter other by as many nodes as the length of the list. Figure 1 shows, on the right, the Java bytecode of the expand method, where local variables 0, 1, and 2 stand, respectively, for this, other, and cursor. The while loop has been compiled into a nonnull check for cursor (lines 2 and 3), which directs to the end of the loop, and into a goto (line 28) which iterates the loop. This Java bytecode (contained in a .class file) is what we really analyze but we report the source Java code for the convenience of the reader, since it is easier to understand. In the rest of this section, we will only report source code. It must be clear, however, that our analysis does not use the source code at all.

Assume that expand is called as follows.

```
public static void main(String[] args) {
  Sharing sh1 = new Sharing(new Sharing(new Sharing(null)));
  Sharing sh2 = new Sharing(new Sharing(null));
  sh1.expand(sh2);
}
```

The preceding call to expand terminates. This is because sh1 is finite, so that the iteration inside the while loop of expand must eventually reach its end. Our analyzer correctly spots this behavior (<init> is the name of a constructor in Java bytecode).

```
All calls to these methods terminate:
  public static Sharing.main(java.lang.String[]):void
  public Sharing.expand(Sharing):void
  public Sharing.<init>(Sharing)
```

Let us now modify the main method a bit.

```
public static void main(String[] args) {
  Sharing sh1 = new Sharing(new Sharing(new Sharing(null)));
  sh1.expand(sh1.next);
}
```

The list sh1 is still finite, but we get a different answer this time.

```
All calls to these methods terminate:
  public Sharing.<init>(Sharing)

Some calls to these methods might not terminate:
  public static Sharing.main(java.lang.String[]):void  [inherits]
  public Sharing.expand(Sharing):void                  [introduces]
```

This means that JULIA identifies a possible divergence for the calls to expand, which induces divergence also for main, which calls expand. The result is perfectly correct: while expand expands the list sh1.next, it also expands the list sh1 initially bound to cursor, so that the loop does not terminate. This is made possible by the destructive update at line 15 of the bytecode in Figure 1: the putfield next bytecode adds new nodes after the first two nodes of sh1, unlinking everything which was previously there.

The aforesaid behavior is not featured by logic programs where data structures are not *mutable*, so that the path-length constraints of the data structure bound to a variable cannot be updated. For instance, the logical unification of

```
Sh1 = sharing(sharing(sharing(Sh2)))
```

constrains the length of Sh1 to be 3 plus the length of Sh2 and this constraint *cannot be changed anymore*: data structures are only created in pure logic or functional languages, never destroyed. In imperative programs, instead, the binding

```
sh1 = new Sharing(new Sharing(new Sharing(sh2)))
```

constrains the length of sh1 to be 3 plus the length of sh2, but this constraint can be updated at any time, as soon as one updates sh1 or sh1.next or sh1.next.next or sh1.next.next.next, that is, as soon as one updates something that shares with sh1. In the expand method in Figure 1, the list sh1 (i.e., this) gets expanded whenever other shares some data structure with sh1, as in the last example of main. This justifies the fact that we need a preliminary *sharing analysis* [Secci and Spoto 2005] in order to perform a precise termination analysis of Java bytecode programs.

Let us show now how cyclicity of data structures can affect the termination of Java bytecode methods. Consider the following `main` method.

```
public static void main(String[] args) {
  Sharing sh1 = new Sharing(new Sharing(new Sharing(null)));
  Sharing sh2 = new Sharing(new Sharing(null));
  sh1.next.next.next = sh1;
  sh1.expand(sh2);
}
```

The analyzer cannot prove the termination of `expand`.

```
All calls to these methods terminate:
  public Sharing.<init>(Sharing)

Some calls to these methods might not terminate:
  public static Sharing.main(java.lang.String[]):void  [inherits]
  public Sharing.expand(Sharing):void                  [introduces]
```

This is correct since the statement `sh1.next.next.next = sh1` makes `sh1` a cyclical list. Therefeore, the `while` loop inside `expand` does not terminate. This justifies why we need a preliminary cyclicity analysis [Rossignoli and Spoto 2006] as an ingredient of our termination analysis of Java programs.

One might be tempted to postulate that the analyzed programs do not use cyclical data structures. This hypothesis is sensible for functional or logical programming languages, where cyclicity is forbidden by the so-called occur-check of pattern-matching and unification, or it is allowed but typically never used by programmers. This hypothesis is instead nonsense for imperative programs, where cyclicity is extensively used: graphs are often used in imperative programs and graphs are typically cyclical. All exceptions are cyclical in Java because of their `cause` field which points to the exception itself; data structures used by compilers are typically cyclical. Our experiments with cyclicity analysis suggest that, on the average, around one-third of the data structures created by a Java bytecode program are cyclical.

It must be clear, however, that taking cyclicity into account does not mean that as soon as a method works over cyclical data structures, its termination cannot be proved. Consider, for instance, the class in Figure 2, which implements a linked list of `Objects` and a set of recursive methods over such a list. Our analyzer finds out that *all* calls inside this class terminate. Nevertheless, cyclicity is created by the statement `l2.tail.tail = l2` inside `main` and `l2` is subsequently passed as an argument to `append`. However, the call `l1.append(l2)` is not affected by the cyclicity of its `l2` argument but only by the cyclicity of its implicit `l1` argument. Since `l1` is *not* cyclical, termination is proved.

The latter example shows that our analysis works correctly also in the presence of recursion, as well as for methods, such as `alternate`, whose termination depends on alternate progression along their arguments.

Let us show some examples now where a termination analysis must take into account the complex semantics of Java bytecode. The class in Figure 3

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

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

  private void iter() {
    if (tail != null) tail.iter();
  }

  private List append(List other) {
    if (tail == null) return new List(head,other);
    else return new List(head,tail.append(other));
  }

  private List reverseAcc(List acc) {
    if (tail == null) return new List(head,acc);
    else return tail.reverseAcc(new List(head,acc));
  }

  private List reverse() {
    if (tail == null) return this;
    else return tail.reverse().append(new List(head,null));
  }

  private List alternate(List other) {
    if (other == null) return this;
    else return new List(head,other.alternate(tail));
  }

  public static void main(String[] args) {
    List l1 = new List(new Object(),new List(new Object(),null));
    List l2 = new List(new Object(),new List(new Object(),null));
    l1.alternate(l2);
    l2.tail.tail = l2;
    l1.append(l2);
    l1.iter();
    l1.reverseAcc(null);
    l1.reverse();
  }
}
```

Fig. 2. A linked list of `Objects` with a set of recursive methods that work over it.

has a `main` method which contains a loop over an integer variable `i`. This loop terminates since the statement `i += 2` inside its body increases `i`, which is bound from above by 20. Our analyzer proves the termination of `main` but only if we perform a preliminary `null` *pointer analysis* of the code. This is because without such analysis, it is impossible to know if the `exc.f = 5` assignment will raise a `NullPointerException` or not. If the exception is raised, the catcher would catch it and reenter the loop without executing the statement `i += 2`.

```
public class Exc {
  private int f;

  public static void main(String[] args) {
    Exc exc = new Exc();
    int i = 0;
    while (i < 20) {
      try {
        if (i > 10) exc.f = 5;
        i += 2;
      }
      catch (NullPointerException e) {}
    }
  }
}
```

Fig. 3. An example of termination in the presence of exceptions.

```
public class Init {
  public void m() {
    new A();
  }

  public void n() {
    A.f = 13;
  }
}
```

Fig. 4. An example dealing with instance and class initialization.

Hence the program would diverge. This example shows that our analyzer deals faithfully with the semantics of this exception.

Figure 4 shows a very simple class `Init`. Class `A` is not shown yet on purpose. Many programmers would conclude that both methods `m` and `n` terminate, regardless of the way one calls them. We can have JULIA prove this by running our termination analysis in *library mode*, which means that the public methods of some class(es) are analyzed without making any hypothesis on their calling context. For instance, the analyzer does not assume any order about which of `m` and `n` is called before the other; it does not assume that any class has been already instantiated before calling `m` or `n`, unless for `Init` itself and some system classes. The results of this analysis might look surprising (`<clinit>` is the name of a class initializer in Java bytecode).

```
All calls to these methods terminate:
  public Init.<init>()

Some calls to these methods might not terminate:
  public Init.m():void                              [inherits]
  public Init.n():void                              [inherits]
  public A.<init>()                                 [introduces]
  package static A.<clinit>():void                  [introduces]
```

Only the (implicit) constructor of `Init` is found to terminate. Methods `m` and `n` *inherit* nontermination because they call some other method that may not terminate. This is correct, since class `A` is defined as follows.

```
public class A {
  public static int f;

  public A() {
    while (true) {}
  }

  static {
    int a = 0;
    while (a == 0) {}
  }
}
```

The instance initializer of `A` diverges, and it is (implicitly) called by method `m`. The class initializer of `A` diverges also, and it is (implicitly) called by both methods `m` and `n`. We recall that the static initializer of class `A` is called, in Java bytecode, only *the first time* that a class is instantiated, or one of its static fields is read or written, or one of its static methods is called.

Assume now that we have the following `main` method inside class `Init`, which fixes the calling contexts of methods `m` and `n`.

```
public static void main(String[] args) {
  new Init().m();
  new Init().n();
}
```

Reverting to a traditional analysis from `main` instead of the library mode, JULIA yields the following result.

```
All calls to these methods terminate:
  public Init.<init>()
  public Init.n():void

Some calls to these methods might not terminate:
  public Init.m():void                               [inherits]
  public static Init.main(java.lang.String[]):void   [inherits]
  public A.<init>()                                  [introduces]
  package static A.<clinit>():void                   [introduces]
```

However, it only does so if a preliminary class initialization analysis is performed. This analysis finds out that, inside method `n`, class `A` has been already initialized by the `new A()` statement inside method `m`, so that no call to the static initializer of `A` happens inside `n` and the method terminates. It is true, however, that that method is never reached since the call to `m` diverges. This example shows that the subtle aspects of the semantics of instance and class initialization of Java are faithfully respected by our analysis.

```
public class Virtual {
  public static void main(String[] args) {
    Node d = new Div();
    Node n = new Nil();
    int l = Integer.parseInt(args[0]);
    while (l-- > 0) n = new Internal(n,n);
    System.out.println(n.height());
  }
}

public abstract class Node {
  public abstract int height();
}

public class Internal extends Node {
  private Node next1;
  private Node next2;
  public Internal(Node next1, Node next2) {
    this.next1 = next1;
    this.next2 = next2;
  }

  public int height() {
    return 1 + Math.max(next1.height(),next2.height());
  }
}

public class Nil extends Node {
  public int height() {
    return 0;
  }
}

public class Div extends Node {
  public int height() {
    // this goes into an infinite recursive loop
    return height();
  }
}
```

Fig. 5. An example dealing with the dynamic dispatch mechanism over nonlinear data structures.

We conclude with an example that shows that our analyzer deals correctly with the dynamic dispatch mechanism of object-oriented languages and with nonlinear data structures. Figure 5 shows a program dealing with a binary tree, implemented as a sequence of `Node`s of several kinds: `Internal` nodes have two successor nodes, while `Nil` and `Div` nodes have no successor. Note that this data structure is not a list nor a one selector data structure. The `height` method is expected to yield the height of the tree but it diverges for `Div` nodes since it calls itself recursively indefinitely. Correctly, our analyzer concludes that all calls inside this program terminate. This is because, although a `Div` object is created by the first statement of `main`, that object does not flow into `n`, so that the call `n.height()`, as well as those recursively activated by the redefinition of method `height` inside `Internal`, never lead to the redefinition of `height` inside `Div`. Hence the program terminates.

If we modify the second statement of `main` into `Node n = new Div()`, we get the following (correct) result.

```
All calls to these methods terminate:
  public Div.<init>()
  public Internal.<init>(Node,Node)
  public Node.<init>()

Some calls to these methods might not terminate:
  public Internal.height():int                        [inherits]
  public Div.height():int                             [introduces]
  public static Virtual.main(java.lang.String[]):void [inherits]
```

This time, the redefinition of `height` inside `Div` is reached by the computation and it introduces divergence. As a consequence, the redefinition of `height` inside `Internal` also inherits divergence, while the redefinition of `height` inside `Nil` is never called.

The previous results are possible because JULIA determines precisely the set of methods that might be called at runtime by each call to a virtual method, such as `n.height()` (the set of its *possible dynamic targets*). This information is computed through a preliminary class analysis [Palsberg and Schwartzbach 1991; Spoto and Jensen 2003].

3. OUR SIMPLIFIED JAVA BYTECODE

In this section we introduce a simplification of the Java bytecode that we will consider in our examples and proofs.

In the following, a total function f is denoted by $\mapsto$ and a partial function by $\rightarrow$. The *domain* and *codomain* of a function f are $dom(f)$ and $rng(f)$, respectively. We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged (it is never reduced).

The Java Virtual Machine runs a Java bytecode program by keeping an activation stack of *states*. Each state is created by a method call and survives until the end of the call.

Definition 1. The set of *values* is $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$, where $\mathbb{L}$ is the set of *memory locations*. A *state* of the Java Virtual Machine is a triple $\langle l \parallel s \parallel \mu \rangle$, where l is an array of values, called *local variables* and numbered from 0 upwards, s is a stack of values, called the *operand stack* (in the following, just *stack*), which grows leftwards, and μ is a *memory*, or *heap*, which maps *locations* into *objects*. An object is a function that maps its fields (identifiers) into values and that embeds a class tag κ; we say that it *belongs to* class κ or is *an instance of* class κ or *has class* κ. We require that there are no dangling pointers, that is, $l \cap \mathbb{L} \subseteq dom(\mu)$, $s \cap \mathbb{L} \subseteq dom(\mu)$ and $rng(\mu(\ell)) \cap \mathbb{L} \subseteq dom(\mu)$ for every $\ell \in dom(\mu)$. We write l^k for the value of the kth local variable; we write s^k for the value of the kth stack element (s^0 is the base of the stack, s^1 is the element above, and so on); we write

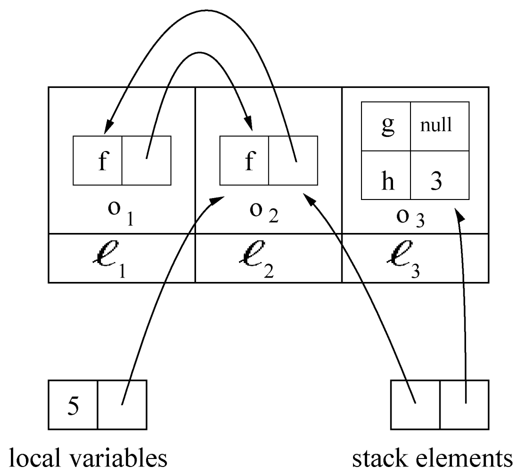

Fig. 6. The state of the Java Virtual Machine considered in Example 2.

$o(f)$ for the value of the field f of an object o. The set of all classes is denoted by $\mathbb{K}$. The set of all states is denoted by Σ. When we want to fix the exact number $\#l \in \mathbb{N}$ of local variables and $\#s \in \mathbb{N}$ of stack elements allowed in a state, we write $\Sigma_{\#l,\#s}$.

We will often write the stack in the form $x :: y :: z :: s$, meaning that x is the topmost value on the stack, y is the underlying element, and z the element still below it; s is the remaining portion of the stack and might be empty. The empty stack is written ε, as well as an empty array of local variables. When s is empty, we often omit it and write $x :: y :: z$ instead of $x :: y :: z :: \varepsilon$. Note that stacks are recursive data structures built from the empty stack ε by pushing elements on top. Hence we should write $x :: y :: z :: s :: \varepsilon$ instead of $x :: y :: z :: s$. We use the second notation for simplicity.

Example 2. Consider a memory $\mu = [\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3]$ where $o_1 = [f \mapsto \ell_2]$, $o_2 = [f \mapsto \ell_1]$ and $o_3 = [g \mapsto \texttt{null}, h \mapsto 3]$. Then a state is

$$\sigma = \langle [5, \ell_2] \parallel \ell_2 :: \ell_3 \parallel \mu \rangle ,$$

shown in Figure 6. Local variable 0 holds integer 5; local variable 1 holds ℓ_2 and is hence bound to the object o_2. The topmost element of the stack also holds ℓ_2 and is hence bound to the object o_2; the underlying element, which is the base of the stack, holds ℓ_3 and is hence bound to the object o_3. We have $\sigma \in \Sigma_{2,2}$ since σ has 2 local variables and 2 stack elements.

Example 3. We have

$$\sigma = \langle [\ell_1, \ell_2, \ell_4] \parallel \ell_3 :: \ell_2 \parallel [\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3, \ell_4 \mapsto o_4, \ell_5 \mapsto o_5] \rangle \in \Sigma_{3,2}$$

where $o_1 = [\texttt{next} \mapsto \ell_4]$, $o_2 = [\texttt{next} \mapsto \texttt{null}]$, $o_3 = [\texttt{next} \mapsto \ell_5]$, $o_4 = [\texttt{next} \mapsto \texttt{null}]$, and $o_5 = [\texttt{next} \mapsto \texttt{null}]$. This state is shown in Figure 7.

In Definition 1 we have assumed for simplicity that values can only be integers, locations, or `null`. The Java Virtual Machine deals with other primitive

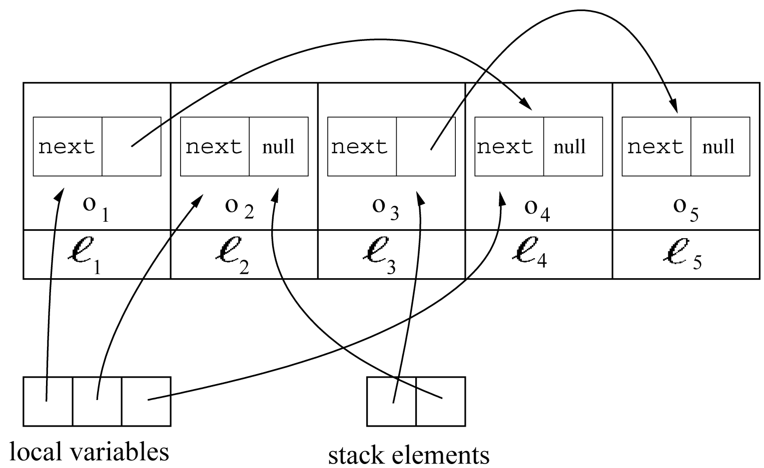

Fig. 7. The state of the Java Virtual Machine considered in Example 3.

types, as well as with arrays. This simplification is useful for our presentation, but our analyzer considers all primitive types and arrays.

Definition 4. The set of *types* of our simplified Java Virtual Machine is $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}, \texttt{void}\}$. The `void` type can only be used as the return type of methods. A method signature is denoted by $\kappa.m(t_1, \ldots, t_p) : t$ standing for a method named m, defined in class κ, expecting p explicit parameters of type, respectively, $t_1, \ldots, t_p$ and returning a value of type t, or returning no value when $t = \texttt{void}$.

We recall that, in object-oriented languages, a method $\kappa.m(t_1, \ldots, t_p) : t$ also has an *implicit* parameter of type κ called `this` inside the code of the method. Hence the actual number of parameters is $p + 1$.

We do not distinguish between methods and constructors. A constructor is just a method named `<init>` and returning `void`. Moreover, there are no static methods in our simplified Java bytecode, although the extension of our definitions to deal with static methods is not difficult and our implementation considers them.

In order to keep the notation reasonably low, we do not formalize the notion of class and the fact that an object of class κ has exactly the fields required by κ; we do not formalize the subclass relation, nor the lookup procedure for a method from a class. We will talk about the *type of a field*, meaning the static type required by the class that defines the field, as well as about the *type of a local variable* or *stack element*, meaning the static type for that local variable or stack element as computed by the type inference algorithm described in Lindholm and Yellin [1999]. However, we will give no formal definition of them.

Java bytecode instructions work over states, by affecting their operand stack, local variables, or memory. There are more than 100 Java bytecode instructions [Lindholm and Yellin 1999]. However, many of them are similar and only differ in the type of their operands. Others are not relevant in this article, such as those that perform tedious but useful stack manipulations. Hence we concentrate here on a very restricted set of 11 instructions only, which exemplify the operations that the Java Virtual Machine performs: stack manipulation, arithmetics, interaction between the stack and the local variables set, object creation and access, and method call. Our implementation considers, of course, the whole set of Java bytecode instructions.

Definition 5. The set of instructions of our simplified Java bytecode is the following (a formalization of their semantics will be given in Section 5).

const c. Pushes on top of the stack the constant c, which can be an integer or `null`.

dup. Pushes on top of the stack its topmost element, which hence gets duplicated.

new κ. Pushes on top of the stack a reference to a new object of class κ (which is properly initialized).

load i. Pushes on top of the stack the value of local variable i.

store i. Pops the topmost value from the stack and writes it into local variable i.

add. Pops the topmost two values from the stack and pushes their sum instead.

getfield f. Pops the topmost value ℓ of the stack, which must be a reference to an object o or `null`, and pushes at its place $o(f)$. If ℓ is `null`, the computation stops.

putfield f. Pops the topmost two values v (the top) and ℓ (under v) from the stack. The value ℓ must be a reference to an object o or `null`. Value v is stored into $o(f)$. If ℓ is `null`, the computation stops.

ifeq of type t. Pops the topmost element of the stack and checks if it is 0 (when t is `int`) or `null` (when $t \in \mathbb{K}$). If this is not the case, the computation stops.

ifne of type t. Pops the topmost element of the stack and checks if it is 0 (when t is `int`) or `null` (when $t \in \mathbb{K}$). If this is the case, the computation stops.

call $\kappa_1.m(t_1, \dots, t_p) : t, \dots, \kappa_n.m(t_1, \dots, t_p) : t$. Pops the topmost $p + 1$ values (the *actual parameters*) $a_0, a_1, \dots, a_p$ from the stack. Value a_0 is called the receiver of the call and must be a reference to an object o or `null`. In the latter case, the computation stops. Otherwise, a lookup procedure is started from the class κ of o upwards along the superclass chain, looking for a method called m, expecting *p formal parameters* of type $t_1, \dots, t_p$, respectively, and returning a value[1] of type t. It is guaranteed that such a method is found in a class belonging to the set $\{\kappa_1, \dots, \kappa_n\}$. That method is run from a state having an empty stack and a set of local variables bound to $a_0, a_1, \dots, a_p$.

The preceding description of bytecode instructions deserves some comments. First of all, we silently assume that the instructions are used correctly, that is, that they are applied to states where they can work. For instance, the dup instruction requires at least an element on the operand stack, or otherwise there

[1] Differently from Java, the return type of the method is used in the lookup procedure of the Java bytecode [Lindholm and Yellin 1999].

is nothing to duplicate; the getfield f and putfield f instructions need a reference to an object o or `null`, but not an integer; they require that o actually contains a field named f; putfield requires that that field has a static type consistent with the value that it is going to write inside. We assume that all these constraints are true, as well as all other structural constraints enumerated in Lindholm and Yellin [1999]. Among these constraints, a very important one is that, however one reaches a Java bytecode instruction in a program, the number and types of the stack elements and the number and types of the local variables are the same. These constraints are checked by the Java bytecode verifier of the Java Virtual Machine. Java bytecode that does not pass these checks is rejected and cannot be run.

The ifeq and ifne instructions stop the computation when the condition they embed is false. This corresponds to the fact that we are going to use those instructions as *filters* at the beginning of the two branches of a conditional. Only one branch will actually continue the execution.

In the call instruction, the set $\kappa_1.m(t_1, \dots, t_p) : t, \dots, \kappa_n.m(t_1, \dots, t_p) : t$ is an overapproximation of the set of its *dynamic targets*, that is, of those methods that might be called at runtime, depending on the runtime class of the receiver. This overapproximation is always computable by looking at the class hierarchy [Dean et al. 1995]. A better one is provided by rapid type analysis [Bacon and Sweeney 1996]. A still better approximation is provided by other examples of class analysis, such as that in Palsberg and Schwartzbach [1991]. The latter, formalized in Spoto and Jensen [2003] as an abstract interpretation of the set of states, is the one used by our implementation.

Method return is implicit in our language, as we will soon see.

Our 11 Java bytecode instructions can be used to write Java bytecode programs. In order to reason about the control flow in the code, we assume that *flat* code, as the one in Figure 1, is given a structure in terms of blocks of code linked by arrows expressing how the flow of control passes from one to another. These might be, for instance, the *basic blocks* of Aho et al. [1986], but we also require that a call instruction can only occur at the beginning of a block. For instance, Figure 8 shows the blocks derived from the code of the method `expand` in Figure 1. The numbers on the right of each instruction are the number of local variables and stack elements at the beginning of the instruction. Note that at the beginning of the methods, the local variables hold the parameters of the method.

The construction of the blocks also can be done in the presence of complex control flows such as those arising from switches, exceptions, and subroutines (the infamous jsr and ret instructions of the Java bytecode), although we do not show it here.

From now on, a *Java bytecode program* will be a graph of blocks, such as that in Figure 8. Inside each block there is one or more instructions among the 11 described before. This graph typically contains many disjoint subgraphs, each corresponding to a different method or constructor. The ends of a method or constructor, where the control flow returns to the caller, are the end of every block with no successor, such as the leftmost one in Figure 8. For simplicity, we assume that the stack there contains exactly as many elements as are needed

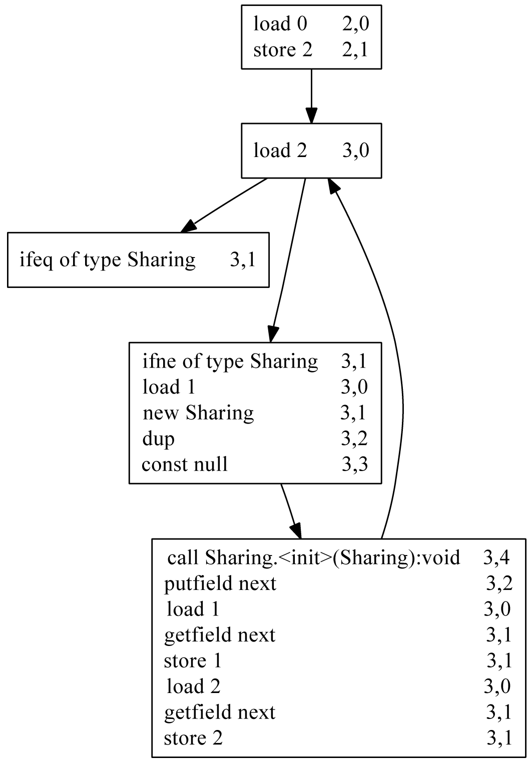

Fig. 8. Our simplified Java bytecode for the method `expand` in Figure 1. On the right of each instruction we report the number of local variables and stack elements at that program point, just before executing the instruction.

to hold the return value (normally 1 element, but 0 elements in the case of methods returning `void`, such as all the constructors).

Definition 6. We write a block containing w bytecode instructions and having m immediate successor blocks $b_1, \dots, b_m$, with $m \geq 0$ and $w > 0$, as

$$\boxed{\begin{array}{c}\mathsf{ins}_1\\ \mathsf{ins}_2\\ \cdots\\ \mathsf{ins}_w\end{array}} \rightrightarrows \begin{array}{c}b_1\\ \cdots\\ b_m\end{array} \qquad \text{or just as} \qquad \boxed{\begin{array}{c}\mathsf{ins}_1\\ \mathsf{ins}_2\\ \cdots\\ \mathsf{ins}_w\end{array}} \qquad \text{when } m = 0.$$

A *Java bytecode program* P is a graph of such blocks.

In the following, P will always stand for the program under analysis.

4. PRELIMINARY ANALYSES

Before defining the path-length analysis in Section 6, we introduce here some preliminary analyses which we assume already performed when the path-length analysis is applied. This is because the path-length analysis uses the information computed by such preliminary analyses and would be extremely imprecise without it: no termination proof could realistically be obtained.

As we mentioned in Section 1, the proofs of termination for imperative programs need information about the possible sharing of data structures between program variables, as well as about the possible cyclicity of the data structures bound to the variables. As a consequence, the first two preliminary analyses are a *possible pair-sharing* analysis (Section 4.1) and a *possible cyclicity* analysis (Section 4.2). We also use a further analysis which is a definite *aliasing* analysis (Section 4.3). The latter is needed due to the way that Java bytecode works, by copying values between local variables and stack elements. Namely, a lot of aliasing is present between local variables and stack elements (due to the instructions load and store) as well as between stack elements (due to the instruction dup). Knowledge about such aliasing is important for the precision of the path-length analysis.

Other preliminary static analyses can contribute to the precision of a subsequent path-length analysis (and hence of termination analysis) although they are not so essential as pair-sharing, cyclicity, and aliasing. Those analyses are discussed in Section 4.4.

4.1 Possible Pair-Sharing

In Section 2 we have seen that a call `sh1.expand(sh2)` terminates when `sh1` and `sh2` are bound to disjoint data structures, but does not terminate when `sh2 == sh1.next`. We have said that the different behavior is a consequence of the different sharing between `sh1` and `sh2` in the two situations. Namely, two variables share if they both reach a common location, possibly transitively [Secci and Spoto 2005].

The precision of our pair-sharing analysis can be improved if it is computed together with *possible update* or, equivalently, definite *purity* or *constancy* information [Salcianu and Rinard 2005; Genaim and Spoto 2008], with a reduced product operation [Cousot and Cousot 1979]. Update means that for each method we know which parameters might be affected by the call, in the sense that some object reachable from those parameters might be modified during the call. Note that this property is much stronger than the `const` annotation of C++, which is a simple syntactical constraint that does not prevent from modifying the objects reachable from a `const` parameter. The reduced product of pair-sharing (as in Secci and Spoto [2005]) with update is what we have implemented inside our analyzer, by using the abstract domain in Genaim and Spoto [2008]. The update component improves the precision of pair-sharing and cyclicity (Section 4.2). Assume, for instance, that the following method

```
void foo(C a, C b) {
  a = b;
 }
```

is called as `foo(x,y)` and that at the calling place variables `x` and `y` do not share with each other. Since, at the end of method `foo`, variables `a` and `b` share, our pair-sharing analysis concludes, conservatively, that variables `x` and `y` are made to share by the call, which is not the case. The update component prevents this, since it knows that no object reachable from `a` or `b` at the moment of the call is modified during the execution of foo. Hence, variables x and y cannot be made to share by the call. The example also works for cyclicity: assume that y is cyclical while x is not cyclical. The cyclicity analysis in Rossignoli and Spoto [2006] concludes that a and hence x are cyclical after the call foo(x,y), which is not the case for x. The update component knows that no object reachable from x is modified during the call and hence x cannot become cyclical. The update component also improves the precision of path-length, as we show in Section 6.

As we said previously, our pair-sharing analysis is completely context-sensitive, which means that the analysis of a method is a function from the input context for the method to the resulting sharing information at its internal and final program points. In this sense, it is a denotational static analysis. The advantage of being context-sensitive is that the approximation of the result of a method can be different for every input context for the call. Consider, for instance, the following method.

```
public Sharing m(Sharing x) {
  return x;
}
```

If one calls this.m(x) in a program point (a *context*) where this and x share, then its result and this share after the call, while they do not share if one calls it in a program point where this and x do not share. A context-sensitive analysis supports this kind of reasoning since the approximation of a method is functional (*denotational*). A noncontext-sensitive analysis, instead, provides an approximation for the output of the method which is consistent with *all* possible calls to the method. In the previous example, a noncontext-sensitive analysis assumes that this and x share after the call, with no regard to the input context. All our preliminary analyses and the path-length analysis that we will define in Section 6 are context-sensitive since they are based on denotational semantics so that they denote methods with relational, functional approximations.

The implementation of a context-sensitive analysis depends on the specific analysis. In general, one distinguishes between properties of the input and properties of the output of a denotation, such as pairs sharing in the input and pairs sharing in the output. Then one builds constraints between those properties. These constraints are often logical implications implemented as binary decision diagrams [Bryant 1986], as explained in Rossignoli and Spoto [2006] and Spoto [2008b]. This is the case of our pair-sharing analysis also. In other cases, they are numerical constraints. For instance, in Section 6, the approximation of a method is a polyhedron over input ($\check{v}$) and output ($\hat{v}$) variables, hence expressing a relation between the input and the output context of a method (in general, of a piece of code).

In order to show our pair-sharing analysis on a concrete example, we fix a specific input context and show the resulting approximations. Namely, Figure 9 shows the result of our pair-sharing analysis applied to the method expand in Figure 8, under the hypothesis that the method is called in a context where its parameters do not share with each other. For instance, we can assume that it is called as sh1.expand(sh2) where sh1 and sh2 do not share. On the right of each instruction we report the set of pairs of variables which might share, according

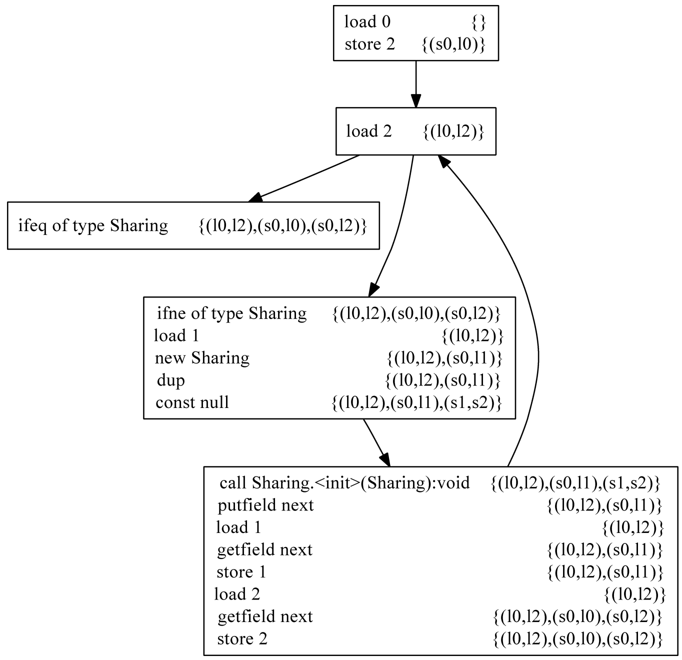

Fig. 9. A pair-sharing analysis of the method `expand` in Figure 8.

to the analysis, just before the instruction is executed. We refer to the ith local variable as `li` and to the ith stack element, from the base, as `si`. Figure 9 has been obtained by first computing the denotation for method `expand` and then fixing the input context of the denotation to compute the resulting abstract information at the output of the method. Information about internal program points (those that are not at the end of a method) has been recovered through *magic-sets* [Payet and Spoto 2007]. Since this is a *possible* pair-sharing analysis, correctness is to be understood in the sense that if two variables v_1 and v_2 actually share at runtime in a given program point, then the (unordered) pair (v_1, v_2) belongs to the approximation at that program point. The converse does not necessarily hold. For simplicity, we do not report information about reflexive sharing, that is, pairs (v, v), since all variables of reference type share with themselves when they are not `null`. We do not report the update component either.

In many cases, sharing is actually aliasing, but this is not always the case. For instance, before the first getfield `next` instruction, the sharing information computed by the analysis is `{(l0,l2),(s0,l1)}`: the top of the stack `s0` shares with `l1`. After reading the `next` field of `s0`, the approximation does not change because the value of the field `next` of `s0` is conservatively assumed to share with `l1`. This would not be the case for aliasing.

4.2 Possible Cyclicity

In Section 2 we have said that it is important, for termination analysis, to spot those variables that might be bound to cyclical data structures, since

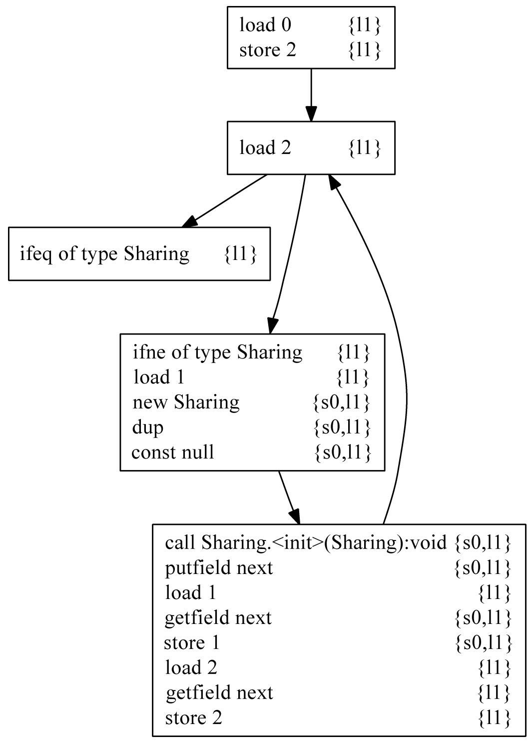

Fig. 10. A cyclicity analysis of the method `expand` in Figure 8.

iterations over such structures might diverge. Namely, a *cyclical* variable is one that reaches a loop of locations. Without cyclicity information, the only possible conservative hypothesis is that *all* variables are cyclical, so that often no proof of termination can be built.

Some aliasing and shape analyses are able to provide cyclicity information. However, also in this case, it is possible to define a more abstract domain which is just made of sets of variables which might be bound to cyclical data structures. This abstract domain, defined and proved correct in Rossignoli and Spoto [2006], can be implemented through Boolean formulas in a completely context- and flow-sensitive way, and is extremely fast in practice. It requires a preliminary sharing analysis to achieve a good level of precision. It exploits purity information, when available, to improve its precision further.

Let us fix again a specific calling context for method `expand` in Figure 1. Namely, let us assume that that method is called as `sh1.expand(sh2)` with `sh1` and `sh2` which do not share and are not cyclical. Our cyclicity analysis builds the empty approximation at every program point inside `expand`, meaning that no local variable and no stack element can be bound to a cyclical data structure inside that method.

Let us fix another calling context for `expand`. Namely, let us assume that it is called as `sh1.expand(sh2)` with `sh1` and `sh2` which do not share and with `sh2` bound to a possibly cyclical data structure (but not `sh1`). The result of the analysis is shown in Figure 10, where on the right of every instruction we have written the set of variables which might be bound to cyclical data structures according to the analysis. Since this is a *possible* cyclicity analysis, correctness means that if a variable is actually bound to a cyclical data structure at a given program point at runtime, then that variable belongs to the approximation computed by the analysis at that program point. The converse is not true in general.

Figure 10 shows that local variable 1, which holds `sh2` in our example, is everywhere potentially bound to a cyclical data structure. When a load 1 instruction pushes its value on the stack, also the top of the stack, which is `s0` there, becomes potentially bound to a cyclical data structure. This is true until that element is popped from the stack.

4.3 Definite Aliasing

Two variables are *aliases* when they are bound to the same value. If this value is a location, then they must be bound to the same data structure (and hence they share); if it is an integer, then this integer must be the same. In both cases, many properties of the two variables are the same, as, for instance, their path-length of Section 6. Hence we want to track *definite* aliasing of variables since their path-length must be the same and this information improves the path-length analysis. It is important to remark that we need definite aliasing, introduced by Java bytecodes such as load, store, and dup, rather than possible aliasing.

We have developed a very simple domain for definite aliasing. It tracks the set of pairs of variables which are definitely aliases. The load, store, and dup bytecodes introduce aliasing into the set. When a variable is modified, the pairs where it occurs are removed from the set. Also this analysis is completely context- and flow-sensitive.

Figure 11 shows the aliasing information computed for the `expand` method in Figure 8 for a calling context such as `sh1.expand(sh2)` where `sh1` and `sh2` are not aliases. On the right of each instruction we report the set of pairs of variables which are definitely aliases according to the analysis. Reflexive aliasing is not reported since a variable is always an alias of itself. This is a definite aliasing analysis. Hence correctness means that if two variables are reported to be aliases in the approximation computed by the analysis at a given program point, then those two variables are actually always aliases at that program point at runtime. The converse is not true in general.

One can see that the analysis finds out that when the dup instruction is executed, the base of the stack, `s0`, is definitely an alias of `l1`. After the dup, the two topmost elements on the stack are also definitely aliases, so that the pair `(s1,s2)` is present in the approximation of the subsequent const `null` instruction.

If two variables are definitely aliases in a program point, then they are also possibly sharing there. This is why the sets in Figure 11 are always included in the corresponding sets in Figure 9.

4.4 Other Preliminary Analyses

In Section 2, we have seen that some analyses can improve the precision of a subsequent path-length analysis (and then of a termination analysis based

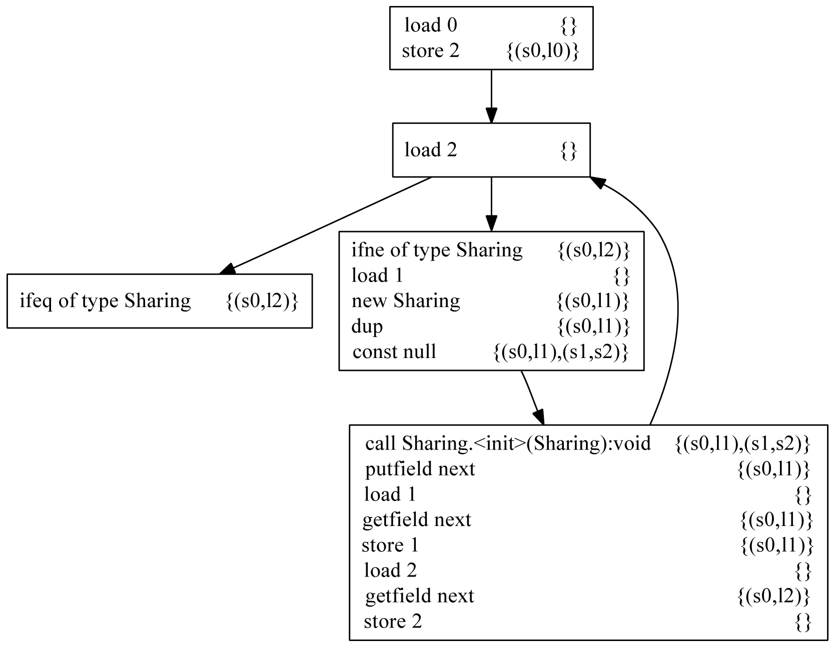

Fig. 11. A definite aliasing analysis of the method `expand` in Figure 8.

on path-length) if they are able to cut away spurious execution paths from the control-flow of the program. We have seen examples related to `null` pointer analysis (Figure 3), class initialization analysis (Figure 4), and class analysis (Figure 5).

Our JULIA analyzer is able to perform all such analyses. The `null` pointer analysis uses an abstract domain implemented through Boolean functions [Spoto 2008b]. It is a rather traditional analysis that we implement in a completely flow- and control-sensitive way. It is true that `null` pointer information is subsumed by the path-length information that we describe in Section 6: a variable contains `null` if and only if its path-length is 0. Nevertheless, our preliminary, very inexpensive `null` pointer analysis simplifies the code which is then used for the path-length analysis. Hence it is useful for the efficiency of the overall termination analysis. Moreover, it determines the non`null` fields more precisely than our path-length analysis and hence it is also useful for precision. Class initialization analysis uses a set of classes which are considered as already initialized. This set can be different in different program points since, again, we implement the analysis in a completely flow- and control-sensitive way. Class analysis is a traditional analysis for object-oriented programs, that we implement in the style of Palsberg and Schwartzbach [1991], by using a flow-sensitive abstract interpretation [Spoto and Jensen 2003].

5. SEMANTICS OF THE JAVA BYTECODE

In this section we define an operational and an equivalent denotational semantics for the Java bytecode. This means that we first define, formally, how each of our 11 instructions modifies the state of the Java Virtual Machine. Then we lift this definition to blocks of instructions. An *operational* semantics is closer to the implementation of an interpreter of the language and it is usually better understood. A *denotational* semantics is important for our purposes since we will use it later to define a *relational* abstract domain that we will call path-length (Section 6). For this reason we present both semantics, which are, however, equivalent, as proved in Payet and Spoto [2007].

We define *state transformers* with the *λ-notation*: $\delta = \lambda\sigma.\sigma'$ is a state transformer such that $\delta(\sigma) = \sigma'$ for every σ. In Definition 7 we often require a specific structure for σ; it is understood that when σ has no such structure, then $\delta(\sigma)$ is undefined. Definition 7 defines the semantics of the bytecode instructions different from call.

Definition 7. Each instruction ins different from call, occurring at a program point q, is associated with its *semantics* $ins_q : \Sigma_{l_i,s_i} \to \Sigma_{l_o,s_o}$ at q, where l_i, s_i, l_o, s_o are the number of local variables and stack elements defined at q and at the subsequent program point(s), respectively (this information is statically known [Lindholm and Yellin 1999]; see, for instance, Figure 8). We assume that $ins_q(\sigma)$ is undefined on every σ where the pairs of variables which are not computed at q by our possible pair-sharing analysis share, or where the variables which are not computed at q by our possible cyclicity analysis are cyclical, or where the pairs of variables computed at q by our definite aliasing analysis are not aliases. Otherwise, ins_q is defined as follows.

$$
\begin{aligned}
const_q\ c &= \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, c :: s \,\|\, \mu\rangle\\
dup_q &= \lambda\langle l \,\|\, top :: s \,\|\, \mu\rangle.\langle l \,\|\, top :: top :: s \,\|\, \mu\rangle\\
new_q\ \kappa &= \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, \ell :: s \,\|\, \mu[\ell \mapsto o]\rangle\\
&\qquad \text{where } \ell \text{ is a fresh location}\\
&\qquad \text{and } o \text{ is an object of class } \kappa \text{ whose fields hold 0 or } \texttt{null}\\
load_q\ i &= \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, l^i :: s \,\|\, \mu\rangle\\
store_q\ i &= \lambda\langle l \,\|\, top :: s \,\|\, \mu\rangle.\langle l[i \mapsto top] \,\|\, s \,\|\, \mu\rangle\\
add_q &= \lambda\langle l \,\|\, x :: y :: s \,\|\, \mu\rangle.\langle l \,\|(x + y) :: s \,\|\, \mu\rangle\\
getfield_q\ f &= \lambda\langle l \,\|\, \ell :: s \,\|\, \mu\rangle.\begin{cases}\langle l \,\|\, \mu(\ell)(f) :: s \,\|\, \mu\rangle & \text{if } \ell \neq \texttt{null}\\ undefined & \text{otherwise}\end{cases}\\
putfield_q\ f &= \lambda\langle l \,\|\, v :: \ell :: s \,\|\, \mu\rangle.\begin{cases}\langle l \,\|\, s \,\|\, \mu[\ell \mapsto \mu(\ell)[f \mapsto v]]\rangle & \text{if } \ell \neq \texttt{null}\\ undefined & \text{otherwise}\end{cases}\\
ifeq\ of\ type_q\ t &= \lambda\langle l \,\|\, top :: s \,\|\, \mu\rangle.\begin{cases}\langle l \,\|\, s \,\|\, \mu\rangle & \text{if } top = 0 \text{ or } top = \texttt{null}\\ undefined & \text{otherwise}\end{cases}\\
ifne\ of\ type_q\ t &= \lambda\langle l \,\|\, top :: s \,\|\, \mu\rangle.\begin{cases}\langle l \,\|\, s \,\|\, \mu\rangle & \text{if } top \neq 0 \text{ and } top \neq \texttt{null}\\ undefined & \text{otherwise.}\end{cases}
\end{aligned}
$$

The fact that these transformers are undefined when the input state does not satisfy the definite information computed by our static analyses is not restrictive, since an instruction at program point q *must* receive an input state where that information is true. For instance, the input state for the dup instruction in Figure 8 must receive an input state where l^0 does not share with l^1 (Figure 9), where l^0 is noncyclical (Figure 10), and where s^0 and l^1 are aliases (Figure 11).

Note that the store i operation might write into a local variable which was not yet used before the same instruction. In such a case, the number of local variables used in the output of the instruction is larger than the number of local variables used in its input.

Example 8. Let q be the program point where the instruction dup of Figure 8 occurs. There are 3 local variables and 2 stack elements there. Hence

$$dup_q = \lambda\langle[l^0, s^0, l^2] \,\|\, s^1 :: s^0 \,\|\, \mu\rangle.\langle[l^0, s^0, l^2] \,\|\, s^1 :: s^1 :: s^0 \,\|\, \mu\rangle \in \Sigma_{3,2} \to \Sigma_{3,3}.$$

Note that, because of the alias information in Figure 11, we require that the base of the stack is an alias of local variable 1. Moreover, μ must be such that the pairs of variables not in $\{(l^0, l^2), (s^0, l^1)\}$ (Figure 9) do not share and the variables not in $\{s^0, l^1\}$ (Figure 10) are not cyclical.

Example 9. Consider the state

$$\sigma = \langle[\ell_1, \ell_2, \ell_4] \,\|\, \ell_3 :: \ell_2 \,\|\, \underbrace{[\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3, \ell_4 \mapsto o_4, \ell_5 \mapsto o_5]}_{\mu}\rangle \in \Sigma_{3,2}$$

of Example 3. Assume that $l_i = 3$ and $s_i = 2$ and that the pair-sharing, cyclicity, and aliasing analyses give empty definite information at some program points q and r. We have

$$\begin{aligned}
(dup_q)(\sigma) &= \langle[\ell_1, \ell_2, \ell_4] \,\|\, \ell_3 :: \ell_3 :: \ell_2 \,\|\, \mu\rangle \in \Sigma_{3,3}\\
(load_q\ 1)(\sigma) &= \langle[\ell_1, \ell_2, \ell_4] \,\|\, \ell_2 :: \ell_3 :: \ell_2 \,\|\, \mu\rangle \in \Sigma_{3,3}\\
(store_q\ 2)(\sigma) &= \langle[\ell_1, \ell_2, \ell_3] \,\|\, \ell_2 \,\|\, \mu\rangle \in \Sigma_{3,1}\\
(getfield_q\ \texttt{next})(\sigma) &= \langle[\ell_1, \ell_2, \ell_4] \,\|\, \ell_5 :: \ell_2 \,\|\, \mu\rangle \in \Sigma_{3,2}\\
((getfield_q\ \texttt{next}); (putfield_r\ \texttt{next}))(\sigma) &= (putfield_r\ \texttt{next})((getfield_q\ \texttt{next})(\sigma))\\
&= \langle[\ell_1, \ell_2, \ell_4] \,\|\, \varepsilon \,\|\, \mu'\rangle \in \Sigma_{3,0}
\end{aligned}$$

where $\mu' = [\ell_1 \mapsto o_1, \ell_2 \mapsto o_2', \ell_3 \mapsto o_3, \ell_4 \mapsto o_4, \ell_5 \mapsto o_5]$ and $o_2' = o_2[\texttt{next} \mapsto \ell_5] = [\texttt{next} \mapsto \ell_5]$.

5.1 Operational Semantics

The state transformers of Definition 7 define the operational semantics of each single bytecode different from call. The semantics of the latter is more difficult to define, since it performs many operations:

(1) creation of a new state for the callee with no local variables and containing only the stack elements of the caller used to hold the actual arguments of the call;
(2) lookup of the dynamic target method on the basis of the runtime class of the receiver;

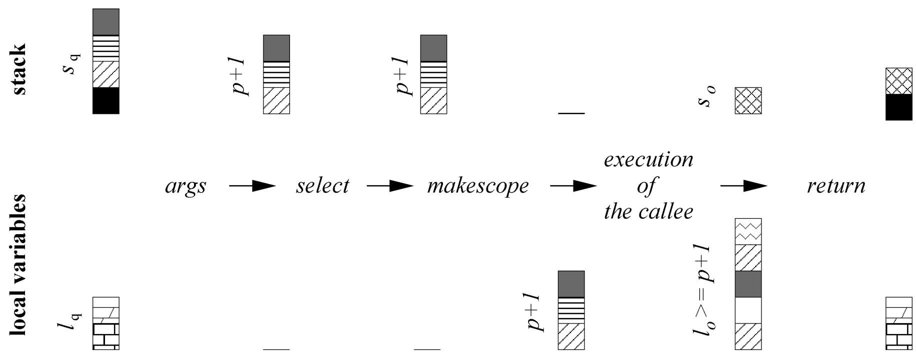

Fig. 12. The execution of a call to a method.

(3) parameter passing, that is, copying the actual arguments from the stack elements to the local variables of the callee;

(4) execution of the dynamic target method and return.

We model (1), (2), and (3) as state transformers, and (4) as the creation of a new configuration for the callee and, finally, the rehabilitation of the configuration of the caller. Figure 12 shows how each of these operations affects the stack and the local variables.

The first operation is formalized as the following state transformer.

Definition 10. Let q be a program point where a call to a method $\kappa.m(t_1, \ldots, t_p) : t$ occurs. Let l_q and s_q be the number of local variables and stack elements at q, respectively. We define

$$args_{q,\kappa.m(t_1,\ldots,t_p):t} \in \Sigma_{l_q,s_q} \rightarrow \Sigma_{0,p+1}$$

as

$$args_{q,\kappa.m(t_1,\ldots,t_p)} = \lambda\langle l \parallel a_p :: \cdots :: a_0 :: s \parallel \mu\rangle.\langle \varepsilon \parallel a_p :: \cdots :: a_0 \parallel \mu\rangle.$$

The second operation is formalized as a *filter* state transformer that checks, for each possible dynamic target method $\kappa_i.m(t_1, \ldots, t_p) : t$, with $1 \leq i \leq n$, if it is actually selected at runtime. We assume that the stack holds only the actual arguments and that the local variables of the callee are not yet initialized.

Definition 11. Let $\kappa.m(t_1, \ldots, t_p) : t$ be a method. We define

$$select_{\kappa.m(t_1,\ldots,t_p):t} : \Delta_{0,p+1\rightarrow 0,p+1}$$

as

$$\lambda\, \underbrace{\langle \varepsilon \parallel a_p :: \cdots :: a_1 :: \ell \parallel \mu\rangle}_{\sigma} . \begin{cases} \sigma & \text{if } \ell \neq \texttt{null} \text{ and the lookup procedure} \\ & \text{of a method } m(t_1, \ldots, t_p) : t \\ & \text{from the class of } \mu(\ell) \\ & \text{selects its implementation in class } \kappa \\ \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

The third operation is formalized by a state transformer that copies the stack elements into the corresponding local variables and clears the stack.

Definition 12. Let $\kappa.m(t_1, \ldots, t_p) : t$ be a method. We define

$$makescope_{\kappa.m(t_1,\ldots,t_p):t} : \Delta_{0,p+1\to p+1,0}$$

as

$$\lambda\langle \varepsilon \,\|\, a_p :: \cdots :: a_1 :: a_0 \,\|\, \mu\rangle.\langle [i \mapsto a_i \mid 0 \le i \le p] \,\|\, \varepsilon \,\|\, \mu\rangle.$$

Definition 12 formalizes the fact that the ith local variable of the callee is a copy of the element $p - i$ positions down the top of the stack of the caller.

We define now the activation stack which tracks the sequence of calls to methods.

Definition 13. A *configuration* is a pair $\langle b \,\|\, \sigma\rangle$ of a block b of the program and a state σ. It represents the fact that the Java Virtual Machine is going to execute b in state σ. An *activation* stack is a stack $c_1 :: c_2 :: \cdots :: c_n$ of configurations, where c_1 is the topmost, current, or *active* configuration. □

We can define now the *operational semantics* of a Java bytecode program.

Definition 14. The (small step) operational semantics of a Java bytecode program P is a relation $a' \Rightarrow_P a''$ providing the immediate successor activation stack a'' of an activation stack a'. It is defined by the following rules.

$$\frac{\mathsf{ins}\text{ is not a }\mathsf{call}}{\left\langle \boxed{\begin{array}{c}\mathsf{ins}\\ rest\end{array}} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix} \,\|\, \sigma\right\rangle :: a \Rightarrow \left\langle \boxed{rest} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix} \,\|\, ins(\sigma)\right\rangle :: a} \tag{1}$$

$$\frac{\begin{array}{c} b_{m_i}\text{ is the block where method } m_i = \kappa_i.m(t_1,\ldots,t_p):t \text{ starts}\\ \sigma = \langle l \,\|\, a_p :: \cdots :: a_0 :: s \,\|\, \mu\rangle, \quad \text{the } \mathsf{call} \text{ occurs at program point } q\\ \sigma' = makescope_{m_i}(select_{m_i}(args_{q,m_i}(\sigma)))\end{array}}{\left\langle \boxed{\begin{array}{c}\mathsf{call}\ m_1,\ldots,m_n\\ rest\end{array}} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix} \,\|\, \sigma\right\rangle :: a \Rightarrow \langle b_{m_i} \,\|\, \sigma'\rangle :: \langle \boxed{rest} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix} \,\|\langle l \,\|\, s \,\|\, \mu\rangle\rangle :: a} \tag{2}$$

$$\frac{}{\langle \boxed{} \,\|\langle l \,\|\, vs \,\|\, \mu\rangle\rangle :: \langle b \,\|\langle l' \,\|\, s' \,\|\, \mu'\rangle\rangle :: a \Rightarrow \langle b \,\|\langle l' \,\|\, vs :: s' \,\|\, \mu\rangle\rangle :: a} \tag{3}$$

$$\frac{1 \le i \le m}{\langle \boxed{} \rightrightarrows \begin{smallmatrix}b_1\\ \cdots\\ b_m\end{smallmatrix} \,\|\, \sigma\rangle :: a \Rightarrow \langle b_i \,\|\, \sigma\rangle :: a} \tag{4}$$

We define $a' \not\Rightarrow_P a''$ as not $a' \Rightarrow_P a''$. We also define $\Rightarrow_P^*$ as the reflexive and transitive closure of $\Rightarrow_P$.

Rule (1) executes an instruction ins, different from call, by using its semantics *ins*. The Java Virtual Machine then moves forward to run the rest of the instructions. Rule (2) calls a method. It chooses one of the possible callees, looks for the block b_{m_i} where the latter starts and builds its initial state σ', by using *args*, *select* and *makescope*. It creates a new current configuration containing

b_{m_i} and σ'. It removes the actual arguments from the old current configuration and the call from the instructions still to be executed at return time. Note that the choice of the possible callee is only apparently nondeterministic, since only one callee will be selected by the *select* function. For all the others, σ' does not exist (*select* is a partial function). Control returns to the caller by rule (3), which rehabilitates the configuration of the caller but forces the memory to be that at the end of the execution of the callee. The return value of the callee is pushed on the stack of the caller. Rule (4) applies when all instructions inside a block have been executed; it runs one of its immediate successors, if any. This rule is normally deterministic, since if a block of the Java bytecode has two or more immediate successors then they start with mutually exclusive conditional instructions and only one thread of control is actually followed.

5.2 Denotational Semantics

In denotational semantics, a state transformer takes traditionally the name of *denotation*. Denotations can be *sequentially* composed, hence modeling the sequential execution of more instructions.

Definition 15. A denotation is a partial function $\Sigma \to \Sigma$ from an *input* state to an *output* or *final* state. The set of denotations is denoted by Δ. When we want to fix the number of local variables and stack elements in the input and output states, we write $\Delta_{l_i,s_i \to l_o,s_o}$, standing for $\Sigma_{l_i,s_i} \to \Sigma_{l_o,s_o}$. Let $\delta_1, \delta_2 \in \Delta$. Their *sequential composition* is $\delta_1;\delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$, which is undefined when $\delta_1(\sigma)$ is undefined or when $\delta_2(\delta_1(\sigma))$ is undefined.

Since denotations are state transformers, Definition 7 gives the denotation of all bytecodes different from call. The denotational semantics of the latter is modeled, in a denotational fashion, by assuming that we already know the functional behavior of the selected dynamic target. As specified by the official documentation [Lindholm and Yellin 1999], it must be the case that at the beginning of the callee the operand stack is empty and the $p + 1$ lowest local variables hold the actual arguments of the call. At its end, the operand stack holds only the return value of the callee, if any, for the simplifying hypothesis of Section 3. Hence it has height $s_o = 1$ if a return value exists and $s_o = 0$ if the callee returns `void`. New local variables might exist at the end of the execution of the callee, used inside its code. Hence at the end we have $l_o \geq p+1$ local variables. Note that the initial local variables, used to store the actual parameters, might have been modified during the execution of the callee. The execution of the callee is hence a denotation $\delta \in \Delta_{0,p+1 \to l_o,s_o}$ where $s_o \in \{0, 1\}$ depending on the return type of the callee and $l_o \geq p + 1$ (Figure 12). We can *plug* this δ into each calling point to the callee. It is enough to observe that the local variables of the caller do not change during the call. Its stack must have the form $a_p :: \cdots :: a_0 :: s$, where $a_p :: \cdots :: a_0$ are the actual arguments of the call and s are the $x \geq 0$ underlying stack elements, if any. The stack elements in s do not change during the call. The $a_p :: \cdots :: a_0$ actual arguments get popped off the stack and replaced with the return value of the callee, if any. The final memory is that reached at the end of the execution of the callee. These considerations let us extend the denotation δ of a callee into that of a call to that callee.

Definition 16. Let $\kappa.m(t_1,\ldots,t_p):t$ be a method and $s_o = 0$ if $t =$ `void`, $s_o = 1$ otherwise. Let $l_o \geq p+1$. Let q be a program point where a call to $\kappa.m(t_1,\ldots,t_p):t$ occurs. Let l_q, s_q be the number of local variables and stack elements used at q, with $s_q = p+1+x$ (at least the $p+1$ actual arguments of the call must be on the stack when we call a method). We define

$$extend_{\kappa.m(t_1,\ldots,t_p):t} : \Delta_{0,p+1\to l_o,s_o} \mapsto \Delta_{l_q,s_q\to l_q,x+s_o}$$

such as, letting $\delta(\langle\varepsilon \,\|\, a_p :: \cdots :: a_1 :: a_0 \,\|\, \mu\rangle) = \langle l' \,\|\, v \,\|\, \mu'\rangle$, $extend_{\kappa.m(t_1,\ldots,t_p):t}(\delta)$ is

$$\lambda\langle l \,\|\, a_p :: \cdots :: a_1 :: a_0 :: s \,\|\, \mu\rangle. \begin{cases} \langle l \,\|\, v :: s \,\|\, \mu'\rangle & \begin{array}{l}\text{if } dom(\mu) \subseteq dom(\mu');\\ \text{every } \ell \in dom(\mu)\\ \text{which is not reachable}\\ \text{from } a_p :: \cdots :: a_1 :: a_0\\ \text{is such that } \mu(\ell) = \mu'(\ell);\\ \text{and if the } k\text{th formal}\\ \text{argument is not modified}\\ \text{by } \kappa.m(t_1,\ldots,t_p):t\\ \text{then } a_k = (l')^k\end{array}\\[2ex] \mathit{undefined} & \text{otherwise.}\end{cases}$$

Here, v stands for the return value of the callee, if any, or otherwise $v = \varepsilon$.

Note that *extend* plays the same role here as *args* and the rule for returning from a method, used in the operational semantics.

In Definition 16 we require that δ, which must be thought of as the current interpretation of $\kappa.m(t_1,\ldots,t_p):t$, does not erase locations: $dom(\mu) \subseteq dom(\mu')$. This constraint would be too strong in the presence of garbage collection (which we do not model in our formalization). In that case, that constraint should be refined by saying that reachable locations cannot be erased by δ. We also require that δ does not modify those objects which are not reachable from the actual parameters of the call. Moreover, if the kth formal parameter is not modified by method $\kappa.m(t_1,\ldots,t_p):t$, then its value is not affected by δ. Note that the latter is a syntactical property: we just look for a store k instruction in the body of $\kappa.m(t_1,\ldots,t_p):t$. If no such instruction is found, then we assert that the kth argument is not modified. All these hypotheses are sensible for our language. Making $extend_{\kappa.m(t_1,\ldots,t_p):t}(\delta)$ undefined when they do not hold is a reasonable definition. These constraints are needed in order to prove the correctness of the abstract *extend* operation of Section 6.

An *interpretation* provides a set of denotations for each block b of the program. These denotations represent the possible runs of the program from the beginning of b until the end of the method where b occur (that is, until a block with no successors). *Sets* can express nondeterministic behaviors, which is not the case in our concrete semantics, but is useful in view of the definition of the abstract semantics in Section 6. By using sets, our concrete semantics is already a *collecting semantics* [Cousot and Cousot 1977]. The operations ; and *extend* over denotations are consequently extended to sets of denotations.

Definition 17. An *interpretation* ι for a program P is a mapping from P's blocks into $\wp(\Delta)$. More precisely, if b is a block such that at its beginning there are l local variables and s stack elements and b is part of the body of a method $\kappa.m(t_1,\ldots,t_p):t$, then $\iota(b)\subseteq\Delta_{l,s\to l_o,s_o}$ where $l_o\geq l$ (new local variables might be declared in the body of the method), $s_o=0$ if $t=$ `void` and $s_o=1$ otherwise. The set of all interpretations is written $\mathbb{I}$ and is ordered by pointwise set-inclusion.

Example 18. The interpretation of the topmost block in Figure 8 must be a subset of $\Delta_{2,0\to3,1}$ since, at the end of method `expand`, there are three local variables and one stack element only. For the same reason, the interpretation of the block containing the load 2 instruction, in the same figure, must be a subset of $\Delta_{3,0\to3,1}$.

Given an interpretation ι providing an approximation of the functional behavior of the blocks of P, we can define an improved interpretation denoted by $[\![]\!]_\iota$.

Definition 19. Let $\iota\in\mathbb{I}$. We define the *denotations in ι of an instruction* ins which is not call as

$$[\![\mathsf{ins}]\!]_\iota=\{ins\},$$

where *ins* is defined in Definition 7. For call, let $m_i=\kappa_i.m(t_1,\ldots,t_p):t$ for $1\leq i\leq n$. We define

$$[\![\mathsf{call}\ m_1,\ldots,m_n]\!]_\iota=\bigcup_{1\leq i\leq n}extend_{m_i}(\{select_{m_i}\};\{makescope_{m_i}\};\iota(b_{m_i})),$$

where b_{m_i} is the block where method m_i starts. The function $[\![\_]\!]_\iota$ is extended to blocks as

$$\left[\!\!\left[\begin{array}{|c|}\hline \mathsf{ins}_1\\ \cdots\\ \mathsf{ins}_w\\ \hline\end{array}\rightrightarrows\begin{array}{c}b_1\\ \cdots\\ b_m\end{array}\right]\!\!\right]_\iota=\begin{cases}[\![\mathsf{ins}_1]\!]_\iota;\cdots;[\![\mathsf{ins}_w]\!]_\iota & \text{if } m=0\\ [\![\mathsf{ins}_1]\!]_\iota;\cdots;[\![\mathsf{ins}_w]\!]_\iota;\big(\iota(b_1)\cup\cdots\cup\iota(b_m)\big) & \text{if } m>0.\end{cases}$$

Note that the semantics of call is computed as the extension of the sequential composition of denotations that select each given possible runtime target method, then pass the parameters, and finally run the target method (Figure 12). Only one of these compositions will be defined: that leading to the target method that is selected at runtime. Note also that the semantics of a block b takes all its followers $b_1,\ldots,b_m$ into account, so that it represents all runs of the method where b occurs from b itself until its end.

The blocks of P are in general interdependent, because of loops and recursion, and a denotational semantics must be built through a fixpoint computation. Given an empty approximation $\iota\in\mathbb{I}$ of the denotational semantics, one improves it into $T_P(\iota)\in\mathbb{I}$ and iterates the application of T_P until a fixpoint, that is, a $\bar\iota$ such that $T_P(\bar\iota)=\bar\iota$. That fixpoint will be the denotational semantics of P, since it corresponds to the minimal solution of the set of equations expressed by T_P. Our analyzer actually performs smaller fixpoints on each strongly connected component of blocks rather than a huge fixpoint over all blocks. This is important for efficiency reasons but irrelevant here for our theoretical results.

Definition 20. The *transformer* $T_P : \mathbb{I} \mapsto \mathbb{I}$ for P is defined as

$$T_P(\iota)(b) = [\![b]\!]_\iota$$

for every $\iota \in \mathbb{I}$ and block b of P.

PROPOSITION 21. *T_P is additive, that is $T_P(\cup_{j\in J}\iota_j) = \cup_{j\in J} T_P(\iota_j)$, so its least fixpoint exists and is equal to $\sqcup_{i\geq 0} T_P^i$, where $T_P^0(b) = \varnothing$ for every block b of P and $T_P^{i+1} = T_P(T_P^i)$ for every $i \geq 0$ [Tarski 1955].*

Definition 22. The *denotational semantics* $\mathcal{D}_P$ of P is the least fixpoint of T_P, as computed in Proposition 21.

Our denotational semantics is defined over the *concrete* domain $\wp(\Delta)$ and uses the denotations of Definitions 7, 11, and 12 which are singleton sets in $\wp(\Delta)$. It also uses the *operators* ;, $\cup$ and *extend* over $\wp(\Delta)$ of Definitions 15 and 16 ($\cup$ is just set union). In order to define an *abstract denotational* semantics, we have to provide an abstract domain, abstract domain elements correctly approximating the singleton sets of denotations and abstract operators correct with respect to the concrete ones. In the next section we will apply this technique to the definition of an abstract domain for path-length of data structures.

As we said at the beginning of this section, our operational and denotational semantics are provably equivalent, as stated by the following result.

THEOREM 23. *Let b a block of a program P and σ_{in} an initial state for b. The functional behavior of b, as modeled by the operational semantics of Section* 5.1*, coincides with its denotational semantics of Section* 5.2*:*

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

5.3 Dealing with Exceptions

We describe here how we deal with exceptions in our semantical framework.

Figure 13 shows the transformation into basic blocks of the method `main` of the program in Figure 3. There are instructions that have not been considered in our simplification of the Java bytecode. The conditionals if_cmpXX are similar to the ifeq and ifne instructions but they work on the topmost *two* values on the stack. The instruction catch is more interesting. It is put after each instruction that might throw an exception. The idea is that it catches such exceptions. Hence it represents the entry point to the exception handlers of the method. The instruction throw throws back an exception to the caller of the method.

In order to formalize the semantics of catch and throw, we start by expanding the semantics of the other instructions. The state is split into a *normal* state and an *exceptional* state. For instance, the semantics of the dup instruction (Definition 7) becomes

$$dup_q = \lambda\langle\langle l \parallel top :: s \parallel \mu\rangle, \sigma_e\rangle.\langle\langle l \parallel top :: top :: s \parallel \mu\rangle, \textit{undefined}\rangle,$$

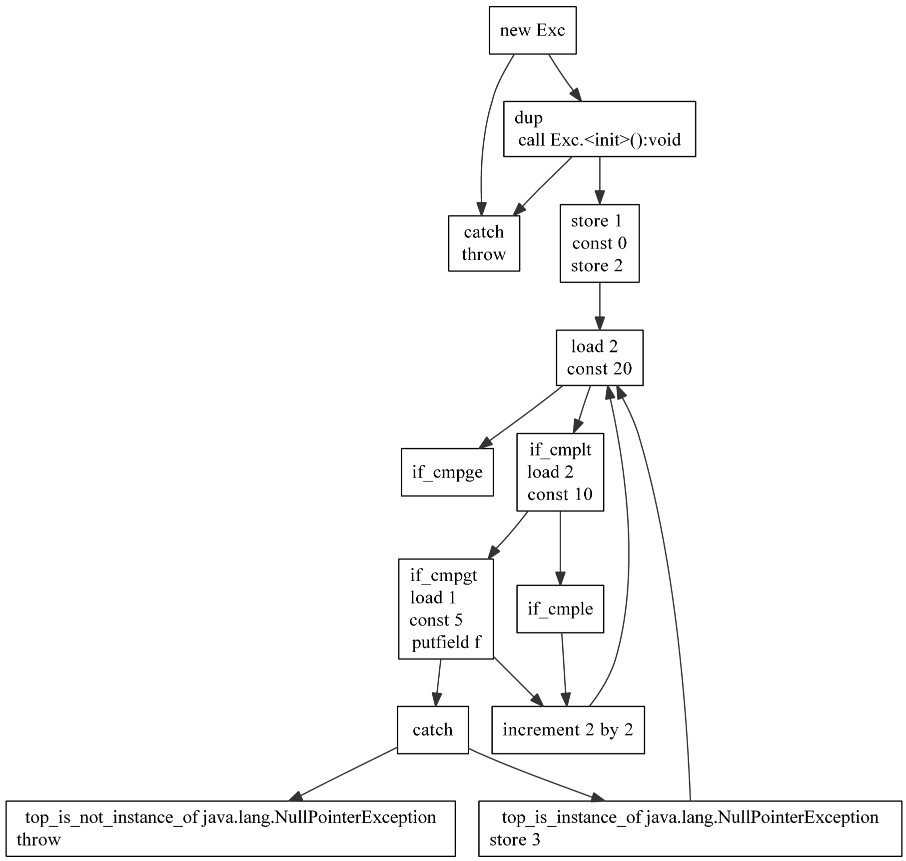

Fig. 13. Our simplified Java bytecode for the method `main` in Figure 3.

which means that dup does not use the state σ_e resulting from an exception that is thrown before it and does not throw any exception (the output exceptional state is *undefined*). Instructions that can throw an exception are modeled as in the following example. We have

$$getfield_q\ f = \lambda\langle\langle l \parallel \ell :: s \parallel \mu\rangle, \sigma_e\rangle. \begin{cases} \langle\langle l \parallel \mu(\ell)(f) :: s \parallel \mu\rangle, undefined\rangle & \text{if } \ell \neq \texttt{null} \\ \langle undefined, \langle l \parallel \ell' \parallel \mu[\ell' \mapsto npe]\rangle\rangle & \text{otherwise} \end{cases}$$

where ℓ' is a fresh location and *npe* is a `NullPointerException` object. This means that the input exceptional state is not used but there might be an output exceptional state, when the object whose field is read is actually `null`. In the latter case, the exceptional state has a stack of one element only, which is a pointer to the exception object; the output normal state is undefined.

On the same line, we can define the semantics of the throw instruction as

$$throw_q = \lambda\langle\langle l \parallel \ell :: s \parallel \mu\rangle, \sigma_e\rangle. \begin{cases} \langle undefined, \langle l \parallel \ell \parallel \mu\rangle\rangle & \text{if } \ell \neq \texttt{null} \\ \langle undefined, \langle l \parallel \ell' \parallel \mu[\ell' \mapsto npe]\rangle\rangle \end{cases}$$

where ℓ' is a fresh location and *npe* is a `NullPointerException` object. This means that the input exceptional state is not used and that this instruction always throws an exception, so that there is no output normal state. The output exceptional state is built from the original input normal state, by throwing away all stack elements but the topmost, which must be a pointer to an exception object. If that pointer is actually `null`, a `NullPointerException` is thrown instead.

The catch instruction catches an exception e which has been thrown just before that instruction. This is modeled by using the input exceptional state to find e. This is the only instruction which uses the input exceptional state and discards the input normal state.

$$catch_q = \lambda\langle\sigma_n, \sigma_e\rangle.\langle\sigma_e, undefined\rangle$$

Since some instructions might throw more than one type of exception (for instance, calls might throw all exceptions thrown by the method that they call), we need to select the right exception handler on the basis of the runtime type of the exception. This is done through top_is_instance_of and top_is_not_instance_of instructions. They check the class tag of the exception object on top of the stack.

$$top\_is\_instance\_of_q\ \kappa = \lambda\langle\langle l \parallel \ell \parallel \mu\rangle, \sigma_e\rangle. \begin{cases} \langle\langle\langle l \parallel \ell \parallel \mu\rangle, undefined\rangle & \text{if } \mu(\ell) \text{ is a } \kappa \\ \langle undefined, undefined\rangle & \text{otherwise} \end{cases}$$

With the use of split states and of the instructions catch, throw, top_is_instance_of, and top_is_not_instance_of, one can define the operational and denotational semantics of Java bytecode exactly as we already did in this section. No other change is required. It is only for simplicity that, in the next sections, we do not consider exceptions in the formalization.

We conclude this section by observing that if `null` pointer analysis is applied to the method in Figure 3 then the lowest two blocks rooted at catch and the block containing catch are removed since the putfield is found to never throw any exception. Without this analysis, there is instead an (apparent) infinite loop passing through the lower catch instruction and termination is not proved.

6. PATH-LENGTH ANALYSIS

In this section we define an abstraction of the denotations of Section 5. Namely, their variables v are abstracted into an integer path-length: if v is bound to a location then the path-length of v is the maximal length of a chain of locations that one can follow from v; if v is bound to an integer i, then the path-length of v is i itself.[2] Since the exact determination of the possible path-lengths of a variable at each given program point is undecidable, we must content ourselves with an approximation of the possible range for the path-lengths. This leads to the use of numerical constraints which are closed polyhedra [Cousot and Halbwachs 1978].

The preceding definition of path-length is formalized shortly. We first define an auxiliary function len^j which follows the chains of locations up to j steps

[2] In our implementation we also consider variables bound to arrays. Their path-length is the length of the array.

of dereference. This function is then used in the definition of the path-length function *len*.

Definition 24. Let μ be a memory (Definition 1). Let

$$\begin{aligned}
len^j(\texttt{null}, \mu) &= 0\\
len^j(i, \mu) &= i \quad \text{if } i \in \mathbb{Z}\\
len^0(\ell, \mu) &= 0 \quad \text{if } \ell \in dom(\mu)\\
len^{j+1}(\ell, \mu) &= 1 + \max\left\{len^j(\ell', \mu) \,\middle|\, \ell' \in rng(\mu(\ell)) \cap \mathbb{L}\right\} \quad \text{if } \ell \in dom(\mu)
\end{aligned}$$

for every $j \geq 0$. We assume that the maximum of an empty set is 0. The *path-length of a value v in μ* is $len(v, \mu) = \lim_{j \to \infty} len^j(v, \mu)$.

In the last case of the definition of len^j, the intersection with $\mathbb{L}$ is needed in order to consider only those values of the fields of the object $\mu(\ell)$ which are locations ℓ'. The fields of type integer of the objects are not used in the definition of the path-length.

Note that if $i \in \mathbb{Z}$ then $len(i, \mu) = len^j(i, \mu) = i$ for every $j \geq 0$ and memory μ. Similarly, $len(\texttt{null}, \mu) = len^j(\texttt{null}, \mu) = 0$ for every memory μ. Moreover, if ℓ is a location bound in μ to a cyclical data structure, then $len(\ell, \mu) = \infty$.

Example 25. Consider the memory

$$\mu = [\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3, \ell_4 \mapsto o_4, \ell_5 \mapsto o_5],$$

where $o_1 = [\texttt{next} \mapsto \ell_4]$, $o_2 = [\texttt{next} \mapsto \texttt{null}]$, $o_3 = [\texttt{next} \mapsto \ell_5]$, $o_4 = [\texttt{next} \mapsto \texttt{null}]$, and $o_5 = [\texttt{next} \mapsto \texttt{null}]$ (Example 3). We have $len(\ell_1, \mu) = 2$, $len(\ell_2, \mu) = 1$, $len(\ell_3, \mu) = 2$, and $len(\ell_4, \mu) = 1$.

We can now map a state into a *path-length assignment*, that is, a function specifying the path-length of its variables. This comes in two versions: in the *input version* $\check{len}$, the state is considered as the input state of a denotation. In the *output version* $\hat{len}$, it is considered as the output state of a denotation. We recall that l^k is the value of the kth local variable in l and s^k is the value of the kth stack element from the base of s (Definition 1).

Definition 26. Let $\langle l \parallel s \parallel \mu \rangle \in \Sigma_{\#l,\#s}$. Its *input path-length assignment* is

$$\check{len}(\langle l \parallel s \parallel \mu \rangle) = [\check{l}^k \mapsto len(l^k, \mu) \mid 0 \leq k < \#l] \cup [\check{s}^k \mapsto len(s^k, \mu) \mid 0 \leq k < \#s]$$

and, similarly, its *output path-length assignment* is

$$\hat{len}(\langle l \parallel s \parallel \mu \rangle) = [\hat{l}^k \mapsto len(l^k, \mu) \mid 0 \leq k < \#l] \cup [\hat{s}^k \mapsto len(s^k, \mu) \mid 0 \leq k < \#s].$$

Example 27. Consider the state

$$\sigma = \langle [\ell_1, \ell_2, \ell_4] \parallel \ell_3 :: \ell_2 \parallel \underbrace{[\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3, \ell_4 \mapsto o_4, \ell_5 \mapsto o_5]}_{\mu} \rangle$$

of Example 3. By using the results of Example 25 we conclude that

$$\check{len}(\sigma) = \begin{bmatrix} \check{l}^0 \mapsto len(\ell_1, \mu), \check{l}^1 \mapsto len(\ell_2, \mu), \check{l}^2 \mapsto len(\ell_4, \mu) \\ \check{s}^1 \mapsto len(\ell_3, \mu), \check{s}^0 \mapsto len(\ell_2, \mu) \end{bmatrix}$$
$$= [\check{l}^0 \mapsto 2, \check{l}^1 \mapsto 1, \check{l}^2 \mapsto 1, \check{s}^1 \mapsto 2, \check{s}^0 \mapsto 1].$$

Similarly, we have

$$\hat{len}(\sigma) = [\hat{l}^0 \mapsto 2, \hat{l}^1 \mapsto 1, \hat{l}^2 \mapsto 1, \hat{s}^1 \mapsto 2, \hat{s}^0 \mapsto 1].$$

Example 28. Consider the state σ of Example 3 and the state

$$dup_q(\sigma) = \langle [\ell_1, \ell_2, \ell_4] \parallel \ell_3 :: \ell_3 :: \ell_2 \parallel \underbrace{[\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_3, \ell_4 \mapsto o_4, \ell_5 \mapsto o_5]}_{\mu} \rangle$$

of Example 9. By Example 25 we have

$$\hat{len}(dup_q(\sigma)) = [\hat{l}^0 \mapsto 2, \hat{l}^1 \mapsto 1, \hat{l}^2 \mapsto 1, \hat{s}^2 \mapsto 2, \hat{s}^1 \mapsto 2, \hat{s}^0 \mapsto 1].$$

Definition 29. Let $l_i, s_i, l_o, s_o \in \mathbb{N}$. The set $\mathbb{PL}_{l_i, s_i \to l_o, s_o}$ of the *path-length polyhedra* contains all finite sets of integer linear constraints over the variables $\{\check{l}^k \mid 0 \le k < l_i\} \cup \{\check{s}^k \mid 0 \le k < s_i\} \cup \{\hat{l}^k \mid 0 \le k < l_o\} \cup \{\hat{s}^k \mid 0 \le k < s_o\}$, using only the $\le$ comparison operator.

Although only $\le$ is allowed in a path-length constraint, we will also write constraints such as $x = y$, standing for both $x \le y$ and $y \le x$.

Example 30. The following polyhedron belongs to $\mathbb{PL}_{3,2 \to 3,3}$.

$$pl = \left\{ \begin{array}{c} \check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2, \check{s}^0 = \hat{s}^0, \check{s}^1 = \hat{s}^1 \\ \check{s}^0 = \check{l}^1, \check{l}^0 \ge 0, \check{l}^1 \ge 0, \check{l}^2 \ge 0, \check{s}^0 \ge 0, \check{s}^1 \ge 0 \\ \check{s}^1 = \hat{s}^2 \end{array} \right\}$$

A path-length assignment fixes the values of the variables. When these values satisfy a path-length constraint, we say that they are a *model* of that constraint.

Definition 31. Let $pl \in \mathbb{PL}_{l_i, s_i \to l_o, s_o}$ and ρ be an assignment from a superset of the variables of pl into $\mathbb{Z} \cup \{\infty\}$. We say that ρ is a *model* of pl and we write $\rho \models pl$ when $pl\rho$ is true, that is, by substituting, in pl, the variables with their values provided by ρ, we get a tautological set of ground constraints.

Example 32. Consider the path-length constraint pl of Example 30 and the state σ of Example 3. By Examples 27 and 28 we have that

$$\rho = \check{len}(\sigma) \cup \hat{len}(dup_q(\sigma)) = \begin{bmatrix} \check{l}^0 \mapsto 2, \check{l}^1 \mapsto 1, \check{l}^2 \mapsto 1, \check{s}^1 \mapsto 2, \check{s}^0 \mapsto 1 \\ \hat{l}^0 \mapsto 2, \hat{l}^1 \mapsto 1, \hat{l}^2 \mapsto 1, \hat{s}^2 \mapsto 2, \hat{s}^1 \mapsto 2, \hat{s}^0 \mapsto 1 \end{bmatrix}$$

is such that

$$pl\rho = \left\{ \begin{array}{c} 2 = 2, 1 = 1, 1 = 1, 1 = 1, 2 = 2 \\ 1 = 1, 2 \ge 0, 1 \ge 0, 1 \ge 0, 1 \ge 0, 2 \ge 0 \\ 2 = 2 \end{array} \right\}.$$

Hence ρ is a model of pl.

We can now define the *concretization* of a path-length constraint. It is the set of denotations that induce input and output assignments that, together, form a model of the constraint.

Definition 33. Let $pl \in \mathbb{PL}_{l_i,s_i \to l_o,s_o}$. Its *concretization* is

$$\gamma(pl) = \left\{ \delta \in \Delta_{l_i,s_i \to l_o,s_o} \;\middle|\; \begin{array}{l} \text{for all } \sigma \in \Sigma_{l_i,s_i} \text{ such that } \delta(\sigma) \text{ is defined} \\ \text{we have } \left(\check{len}(\sigma) \cup \hat{len}(\delta(\sigma))\right) \models pl \end{array} \right\}.$$

Example 34. Consider the path-length constraint pl of Example 30. In Example 32 we have seen that the state σ of Example 3 is such that $(\check{len}(\sigma) \cup \hat{len}(dup_q(\sigma))) \models pl$, where dup_q is the denotation of the dup instruction in Figure 8, given in Example 8. However, this is true for *every* input state σ such that $dup_q(\sigma)$ is defined. This is because every such σ has the form $\langle [l^0, s^0, l^2] \,\|\, s^1 :: s^0 \,\|\, \mu \rangle$ and satisfies the static information of Figures 9, 10 and 11. Hence

$$\begin{aligned} \rho &= \check{len}(\sigma) \cup \hat{len}(dup_q(\sigma)) \\ &= \check{len}\left(\langle [l^0, s^0, l^2] \,\|\, s^1 :: s^0 \,\|\, \mu\rangle\right) \cup \hat{len}\left(\langle [l^0, s^0, l^2] \,\|\, s^1 :: s^1 :: s^0 \,\|\, \mu\rangle\right) \\ &= \begin{bmatrix} \check{l}^0 \mapsto len(l^0,\mu), \check{l}^1 \mapsto len(s^0,\mu), \check{l}^2 \mapsto len(l^2,\mu) \\ \check{s}^1 \mapsto len(s^1,\mu), \check{s}^0 \mapsto len(s^0,\mu) \\ \hat{l}^0 \mapsto len(l^0,\mu), \hat{l}^1 \mapsto len(s^0,\mu), \hat{l}^2 \mapsto len(l^2,\mu) \\ \hat{s}^2 \mapsto len(s^1,\mu), \hat{s}^1 \mapsto len(s^1,\mu), \hat{s}^0 \mapsto len(s^0,\mu) \end{bmatrix}. \end{aligned}$$

It follows that

$$pl\rho = \left\{ \begin{array}{c} len(l^0,\mu) = len(l^0,\mu), len(s^0,\mu) = len(s^0,\mu) \\ len(l^2,\mu) = len(l^2,\mu), len(s^0,\mu) = len(s^0,\mu), len(s^1,\mu) = len(s^1,\mu) \\ len(s^0,\mu) = len(s^0,\mu), len(l^0,\mu) \ge 0, len(s^0,\mu) \ge 0 \\ len(l^2,\mu) \ge 0, len(s^0,\mu) \ge 0, len(s^1,\mu) \ge 0, len(s^1,\mu) = len(s^1,\mu) \end{array} \right\}$$

which is true since variables l^0, s^0, l^2, and s^1 do not have integer type at the beginning of the execution of the dup instruction in Figure 8 and hence their path-length is nonnegative (Definition 24). In conclusion, we have

$$dup_q \in \gamma(pl).$$

We want to order our path-length constraints on the basis of their concretization: $pl_1 \le pl_2$ if and only if $\gamma(pl_1) \subseteq \gamma(pl_2)$. This results in a poset of polyhedra. The $\sqcap$ operation over sets of constraints is the union of the constraints, that is, the intersection of the polyhedra that they represent, and the $\sqcup$ operation is the *polyhedral hull* [Stoer and Witzgall 1970] of the polyhedra that they represent, that is, the smallest closed polyhedron which includes both.

In the following, we identify in the same equivalence class all elements having the same concretization. For instance, $\{x \le y+1\}$ and $\{x+2 \le y+3\}$ are the same abstract element since $\gamma(\{x \le y+1\}) = \gamma(\{x+2 \le y+3\})$.

Definition 35. The *path-length polyhedra* $\mathbb{PL}_{l_i,s_i \to l_o,s_o}$ are ordered as $pl_1 \le pl_2$ if and only if $\gamma(pl_1) \subseteq \gamma(pl_2)$. They form a *poset*, that is, $\le$ is reflexive, transitive, and antisymmetric. Their elements silently stand for their equivalence class. Their top element is the tautological constraint *true* (which stands for an empty set of linear constraints). Their least element is the constraint *false* (which stands for a constraint such as $1 \leq 0$).

By the theory of abstract interpretation, we get a correct abstract denotational semantics $\mathcal{D}_P^{\mathbb{PL}}$ for path-length as soon as we substitute the concrete denotations of Definition 7 with elements of $\mathbb{PL}$ which include them in their concretization. Moreover, we must provide the abstract counterparts over $\mathbb{PL}$ of the operations ;, $\cup$ and *extend* over $\wp(\Delta)$.

We first define a constraint stating that no local variable and no stack element is modified, that if two variables are definitely aliases, then they must have the same path-length and that all variables of reference (noninteger) type have nonnegative path-length.

Definition 36. Let $L, S \subseteq \mathbb{N}$ and q be a program point where there are l_q local variables and s_q stack elements. We define

$$\begin{aligned}
\mathit{Unchanged}_q(L,S) = {}& \{\check{l}^i = \hat{l}^i \mid i \in L\}\\
&\cup \{\check{s}^i = \hat{s}^i \mid i \in S\}\\
&\cup \left\{\check{s}^i = \check{s}^j \;\middle|\; \begin{array}{l} 0 \leq i, j < s_q \text{ and } s^i \text{ is an alias of } s^j \text{ at } q\\ \text{according to our definite aliasing analysis}\end{array}\right\}\\
&\cup \left\{\check{s}^i = \check{l}^j \;\middle|\; \begin{array}{l} 0 \leq i < s_q,\ 0 \leq j < l_q \text{ and } s^i \text{ is an alias of } l^j \text{ at } q\\ \text{according to our definite aliasing analysis}\end{array}\right\}\\
&\cup \left\{\check{l}^i = \check{l}^j \;\middle|\; \begin{array}{l} 0 \leq i, j < l_q \text{ and } l^i \text{ is an alias of } l^j \text{ at } q\\ \text{according to our definite aliasing analysis}\end{array}\right\}\\
&\cup \{\check{s}^i \geq 0 \mid 0 \leq i < s_q \text{ and } s^i \text{ does not have integer type at } q\}\\
&\cup \{\check{l}^i \geq 0 \mid 0 \leq i < l_q \text{ and } l^i \text{ does not have integer type at } q\}.
\end{aligned}$$

Let $l, s \in \mathbb{N}$. Then $\mathit{Unchanged}_q(l,s) = \mathit{Unchanged}_q(\{0,\ldots,l-1\},\{0,\ldots,s-1\})$.

Let us define the abstract counterparts of the *ins* denotations now.

Definition 37. Let $\#l, \#s$ be the number of local variables and stack elements at a program point q. The abstract counterparts of the denotations of Definition 7 are the following:

$$\begin{aligned}
\mathit{const}_q^{\mathbb{PL}}\ c &= \begin{cases} \mathit{Unchanged}_q(\#l,\#s) \cup \{c = \hat{s}^{\#s}\} & \text{if } c \in \mathbb{Z}\\ \mathit{Unchanged}_q(\#l,\#s) \cup \{0 = \hat{s}^{\#s}\} & \text{if } c = \texttt{null}\end{cases}\\
\mathit{dup}_q^{\mathbb{PL}} &= \mathit{Unchanged}_q(\#l,\#s) \cup \{\check{s}^{\#s-1} = \hat{s}^{\#s}\}\\
\mathit{new}_q^{\mathbb{PL}}\ \kappa &= \mathit{Unchanged}_q(\#l,\#s) \cup \{1 = \hat{s}^{\#s}\}\\
\mathit{load}_q^{\mathbb{PL}}\ i &= \mathit{Unchanged}_q(\#l,\#s) \cup \{\check{l}^i = \hat{s}^{\#s}\}\\
\mathit{store}_q^{\mathbb{PL}}\ i &= \mathit{Unchanged}_q(\{0,\ldots,\#l-1\} \setminus i, \{0,\ldots,\#s-2\}) \cup \{\check{s}^{\#s-1} = \hat{l}^i\}\\
\mathit{add}_q^{\mathbb{PL}} &= \mathit{Unchanged}_q(\#l,\#s-2) \cup \{\check{s}^{\#s-2} + \check{s}^{\#s-1} = \hat{s}^{\#s-2}\}
\end{aligned}$$

$$getfield_q^{\mathbb{PL}}\ f = \begin{cases} Unchanged_q(\#l, \#s-1) \\ \quad \text{if } f \text{ has integer type} \\ Unchanged_q(\#l, \#s-1) \cup \{\check{s}^{\#s-1} \geq \hat{s}^{\#s-1}\} \\ \quad \text{if } f \text{ does not have integer type and } s^{\#s-1} \text{ might be cyclical} \\ \quad\quad \text{at } q \\ Unchanged_q(\#l, \#s-1) \cup \{\check{s}^{\#s-1} \geq 1 + \hat{s}^{\#s-1}\} \\ \quad \text{if } f \text{ does not have integer type and } s^{\#s-1} \text{ cannot be cyclical} \\ \quad\quad \text{at } q \end{cases}$$

$$putfield_q^{\mathbb{PL}}\ f = \begin{cases} Unchanged_q(\#l, \#s-2) \\ \quad \text{if } f \text{ has integer type} \\ Unchanged_q(L, S) \\ \quad \text{if } s^{\#s-2} \text{ might share with } s^{\#s-1} \text{ at } q \\ Unchanged_q(L, S) \cup \{\check{l}^i + \check{s}^{\#s-1} \geq \hat{l}^i \mid 0 \leq i < \#l, i \notin L\} \\ \quad \cup \{\check{s}^i + \check{s}^{\#s-1} \geq \hat{s}^i \mid 0 \leq i < \#s-2,\ i \notin S\} \\ \quad \text{otherwise} \end{cases}$$

where L are the indexes of the local variables which cannot share with $s^{\#s-2}$ at q and S the indexes x of the stack elements, with $0 \leq x < \#s-2$, which cannot share with $s^{\#s-2}$ at q

$$ifeq\ of\ type_q^{\mathbb{PL}}\ t = Unchanged_q(\#l, \#s-1) \cup \{\check{s}^{\#s-1} = 0\}$$

$$ifne\ of\ type_q^{\mathbb{PL}}\ t = \begin{cases} Unchanged_q(\#l, \#s-1) \cup \{\check{s}^{\#s-1} \geq 1\} & \text{if } t \neq \texttt{int} \\ Unchanged_q(\#l, \#s-1) & \text{otherwise.} \end{cases}$$

The abstract operations use the *Unchanged* constraint for the part of the state which they do not modify. The part which is modified is modeled explicitly. For instance, the $const^{\mathbb{PL}}$ constraint says that the new top of the stack $s^{\#s}$ has path-length c when c is an integer value and 0 when c is `null`. The $dup^{\mathbb{PL}}$ constraint copies the path-length of the old top of the stack $\check{s}^{\#s-1}$ into the path-length of the new top of the stack $\hat{s}^{\#s}$.

The definition of $getfield_q^{\mathbb{PL}}$ states that if we read the field of an object then we get a value whose path-length is no larger than the path-length $\check{s}^{\#s-1}$ of the object. Moreover, if the object cannot be cyclical, the path-length of its field is strictly smaller than $\check{s}^{\#s-1}$.

For the definition of $putfield_q^{\mathbb{PL}}$, remember that $s^{\#s-2}$ holds the object whose field f is going to be modified, and that $s^{\#s-1}$ holds the value which is going to be written inside f (Definition 7). Definition 37 states that if f has integer type then no path-length changes. Otherwise, the local variables L and stack elements S which do not share at q with the object whose field is modified (i.e., with $s^{\#s-2}$), and that still exist in the output of the instruction, do not change their path-length. The other variables are affected by the putfield instruction. Namely, if the putfield might build a cycle, that is, if the variable $s^{\#s-2}$ holding the object might share with the variable $s^{\#s-1}$ holding the value which is going to be written inside the field f of the object, then the path-length of the variables not in L and not in S is not approximated (it might become infinite). Otherwise it can only grow by the path-length of the value $s^{\#s-1}$ which is stored inside the field.

Example 38. Consider the dup instruction in Figure 8. We know that l^1 and s^0 are aliases at the program point q where the instruction occurs (Figure 11). Hence $dup_q^{\mathbb{PL}}$ is the constraint pl of Example 30.

Example 39. Let q be now the program point at the beginning of the code in Figure 8. Consider the load 0 instruction at q. There are 2 local variables at q (the parameters of the method), both of noninteger type, and no stack elements. No variables are aliases at q (Figure 11). Hence

$$\begin{aligned}
load_q^{\mathbb{PL}}\ 0 &= Unchanged_q(2,0) \cup \{\check{l}^0 = \hat{s}^0\}\\
&= \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0\} \cup \{\check{l}^0 = \hat{s}^0\}\\
&= \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^0 = \hat{s}^0\}.
\end{aligned}$$

Example 40. Let r be the program point at the beginning of the store 2 instruction in the topmost block in Figure 8. There are 2 local variables at r (the parameters of the method), both of noninteger type, and 1 stack element, of noninteger type. Variables s^0 and l^0 are aliases at r (Figure 11). Hence

$$\begin{aligned}
store_r^{\mathbb{PL}}\ 2 &= Unchanged_r(\{0,1\},\varnothing) \cup \{\check{s}^0 = \hat{l}^2\}\\
&= \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{s}^0 = \check{l}^0, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{s}^0 \geq 0\} \cup \{\check{s}^0 = \hat{l}^2\}\\
&= \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{s}^0 = \check{l}^0, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{s}^0 \geq 0, \check{s}^0 = \hat{l}^2\}.
\end{aligned}$$

Example 41. Let r be the program point at the beginning of the first getfield `next` instruction in the lowest block in Figure 8. Assume that the argument of the method might be a cyclical list. There are 3 local variables at r, all of noninteger type, and 1 stack element, of noninteger type. That stack element might be cyclical if the input argument of the method might be cyclical (Figure 10). Variables s^0 and l^1 are aliases at r (Figure 11). Hence

$$\begin{aligned}
getfield_r^{\mathbb{PL}}\ \texttt{next} &= Unchanged_r(\{0,1,2\},\varnothing) \cup \{\check{s}^0 \geq \hat{s}^0\}\\
&= \left\{\begin{array}{l}\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2,\\ \check{s}^0 = \check{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^2 \geq 0, \check{s}^0 \geq 0\end{array}\right\} \cup \{\check{s}^0 \geq \hat{s}^0\}\\
&= \left\{\begin{array}{l}\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^2 = \hat{l}^2,\\ \check{s}^0 = \check{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^2 \geq 0, \check{s}^0 \geq 0, \check{s}^0 \geq \hat{s}^0\end{array}\right\}.
\end{aligned}$$

Example 42. Let r be the program point at the beginning of the putfield `next` instruction in the lowest block in Figure 8. Assume that the argument of the method might be a cyclical list. There are 3 local variables at r, all of noninteger type, and 2 stack elements, of noninteger type. Variables s^0 and l^1 are aliases at r (Figure 11). Only variables s^0 and l^1 and variables l^0 and l^1 might share at r (Figure 9). Hence we are in the third case for $putfield_r^{\mathbb{PL}}$ in Definition 37.

We have $L = \{l^0, l^2\}$ and $S = \varnothing$. Hence

$$\begin{aligned}
\mathit{putfield}_r^{\mathbb{PL}}\ \texttt{next} &= \mathit{Unchanged}_r(L, S) \cup \{\check{l}^1 + \check{s}^1 \geq \hat{l}^1\} \\
&= \left\{ \begin{array}{l} \check{l}^0 = \hat{l}^0, \check{l}^2 = \hat{l}^2, \\ \check{s}^0 = \hat{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^2 \geq 0, \check{s}^0 \geq 0 \end{array} \right\} \cup \{\check{l}^1 + \check{s}^1 \geq \hat{l}^1\} \\
&= \left\{ \begin{array}{l} \check{l}^0 = \hat{l}^0, \check{l}^2 = \hat{l}^2, \\ \check{s}^0 = \hat{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^2 \geq 0, \check{s}^0 \geq 0, \check{l}^1 + \check{s}^1 \geq \hat{l}^1 \end{array} \right\}.
\end{aligned}$$

The intuition of this result is that locals 0 and 2 do not change their path-length, since they are not affected by the modification of the field. Local 1 (that is, `other` in Figure 1), instead, might increase its path-length by as much as the path-length of the value which is written inside the field `next`.

We also provide correct approximations for the denotations used for a method call.

Definition 43. Let $\kappa.m(t_1, \ldots, t_p) : t$ be a method. We define

$$\begin{aligned}
\mathit{args}_{q,\kappa.m(t_1,\ldots,t_p)}^{\mathbb{PL}} &= \{\check{s}^{s_q-(p+1)+i} = \hat{s}^i \mid 0 \leq i < p+1\} \\
\mathit{select}_{\kappa.m(t_1,\ldots,t_p):t}^{\mathbb{PL}} &= \mathit{Unchanged}(0, p+1) \\
\mathit{makescope}_{\kappa.m(t_1,\ldots,t_p):t}^{\mathbb{PL}} &= \{\check{s}^i = \hat{l}^i \mid 0 \leq i < p+1\}.
\end{aligned}$$

We define now the abstract counterparts of the operators ;, $\cup$ and *extend* over sets of denotations. For ;, we sequentially compose two path-length constraints by matching the output variables of the first with the input variables of the second. This is accomplished by renaming such variables into new overlined variables T, which are then projected away with the $\exists_T$ operation. The $\cup^{\mathbb{PL}}$ operation is just the polyhedral hull operation. For *extend*, recall that we assume we have already performed many preliminary static analyses (Section 4). Namely, we assume that at the program point where a call instruction occurs we know:

(1) which stack elements or local variables of the caller might share;
(2) which stack elements or local variables of the caller must be aliases of each other;
(3) which formal parameters of the callee might be updated during the execution of the callee (that is, some reachable object might change its fields);
(4) which formal parameters of the callee might be modified during the execution of the callee. This is just a syntactical property: parameter k is modified if a store k instruction occurs inside the code of the callee.

Definition 44. Let $pl_1 \in \mathbb{PL}_{l_i,s_i \to l_t,s_t}$ and $pl_2 \in \mathbb{PL}_{l_t,s_t \to l_o,s_o}$. Let us also define $T = \{\bar{l}^0, \ldots, \bar{l}^{l_t-1}, \bar{s}^0, \ldots, \bar{s}^{s_t-1}\}$. We define $pl_1 ;^{\mathbb{PL}} pl_2 \in \mathbb{PL}_{l_i,s_i \to l_o,s_o}$ as

$$pl_1 ;^{\mathbb{PL}} pl_2 = \exists_T \left(pl_1[\hat{v} \mapsto \bar{v} \mid \bar{v} \in T] \cup pl_2[\check{v} \mapsto \bar{v} \mid \bar{v} \in T] \right).$$

Let $pl_1, pl_2 \in \mathbb{PL}_{l_i,s_i \to l_o,s_o}$. We define

$$pl_1 \cup^{\mathbb{PL}} pl_2 = \text{polyhedral hull of } pl_1 \text{ and } pl_2.$$

Let $\kappa.m(t_1,\ldots,t_p):t$ be a method and $s_o = 0$ if $t =$ `void`, $s_o = 1$ otherwise. Let $l_o \geq p+1$. Let q be a program point where a call to $\kappa.m(t_1,\ldots,t_p):t$ occurs. Let l_q, s_q be the number of local variables and stack elements used at q, with $s_q = p+1+x$ (at least the $p+1$ actual arguments of the call must be on the stack when one calls a method). The actual parameters of the call at q are held in s^{x+k} with $0 \leq k < p+1$. We define

$$extend^{\mathbb{PL}}_{\kappa.m(t_1,\ldots,t_p):t} : \mathbb{PL}_{0,p+1\to l_o,s_o} \mapsto \mathbb{PL}_{l_q,s_q\to l_q,x+s_o}$$

as

$$\begin{aligned}&extend^{\mathbb{PL}}_{\kappa.m(t_1,\ldots,t_p):t}(pl) =\\ &\quad = \exists_T \left(\begin{array}{c} pl[\hat{v} \mapsto \overline{v} \mid \overline{v} \in T][\check{s}^k \mapsto \check{s}^{k+x} \mid 0 \leq k < p+1][\hat{s}^0 \mapsto \hat{s}^x] \\ \cup US \cup MSA \cup UL \cup MLA \end{array}\right)\end{aligned}$$

where

$$\begin{aligned}
T &= \{\overline{l}^0, \ldots, \overline{l}^{l_o-1}\}\\
US &= \{\check{s}^i = \hat{s}^i \mid 0 \leq i < x \text{ and } s^i \text{ cannot share with any possibly updated parameter}\}\\
MSA &= \left\{ \overline{l}^k = \hat{s}^i \;\middle|\; \begin{array}{l} 0 \leq i < x,\ 0 \leq k < p+1, \\ s^i \text{ is a definite alias of the } k\text{th parameter} \\ \text{and the latter is not modified inside the callee} \end{array} \right\}\\
UL &= \{\check{l}^i = \hat{l}^i \mid 0 \leq i < l_q \text{ and } l^i \text{ cannot share with any possibly updated parameter}\}\\
MLA &= \left\{ \overline{l}^k = \hat{l}^i \;\middle|\; \begin{array}{l} 0 \leq i < l_q,\ 0 \leq k < p+1, \\ l^i \text{ is a definite alias of the } k\text{th parameter} \\ \text{and the latter is not modified inside the callee} \end{array} \right\}.
\end{aligned}$$

Example 45. Consider the constraints of Examples 39 and 40. We have

$$\begin{aligned}
&(load^{\mathbb{PL}}_q\ 0);^{\mathbb{PL}} (store^{\mathbb{PL}}_r\ 2)\\
&= \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^0 = \hat{s}^0\}\\
&\quad ;^{\mathbb{PL}} \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{s}^0 = \check{l}^0, \check{l}^0 \geq 0, \check{s}^0 \geq 0, \check{s}^0 = \hat{l}^2\}\\
&= \exists_{\{\overline{l}^0,\overline{l}^1,\overline{s}^0\}} \left\{ \begin{array}{c} \check{l}^0 = \overline{l}^0, \check{l}^1 = \overline{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^0 = \overline{s}^0 \\ \overline{l}^0 = \hat{l}^0, \overline{l}^1 = \hat{l}^1, \overline{s}^0 = \overline{l}^0, \overline{l}^0 \geq 0, \overline{s}^0 \geq 0, \overline{s}^0 = \hat{l}^2 \end{array} \right\}\\
&= \{\check{l}^0 = \hat{l}^0, \check{l}^1 = \hat{l}^1, \check{l}^0 \geq 0, \check{l}^1 \geq 0, \check{l}^0 = \hat{l}^2\}.
\end{aligned}$$

The $extend^{\mathbb{PL}}$ operation is rather complex. Do not consider the MSA and MLA sets for the moment. Then the definition says that if we know the path-length behavior pl of the called method(s), we just have to *lift* the input stack elements of pl by x positions, since the callee starts with $p+1$ stack elements which are copies of the highest $p+1$ stack elements of the caller. The latter, however, has x more underlying elements (Definition 16). The same must be performed for the only output stack element which might be used by the callee to yield its return value. The output local variables are renamed into new overlined variables in

T which are finally removed by $\exists_T$. This definition would already be correct, but extremely imprecise. In fact, it does not say anything about the effect of the call on the set of variables $Y = \{l^i \mid 0 \le i < l_q\} \cup \{s^i \mid 0 \le i < x\}$ which contains all the local variables of the caller and the x lower stack elements of the caller, those which are not used to hold the $p+1$ parameters. This is the purpose of the UL and US sets, respectively. They say that the path-length of any $v \in Y$ is not modified by the call, but only if v cannot share with any of the parameters of the call which might be updated during the execution of the callee. This is correct since in such a case the callee has no way of modifying the objects reachable from v and hence the path-length of v cannot be affected by the call.

The definition in Spoto et al. [2006] stopped here and actually did not even use the update information, so that it only required nonsharing in the definition of the sets US and UL. Hence it was less precise. We improve it here further by using the sets of constraints MSA and MLA. They consider the case when some $v \in Y$ is well shared with the kth actual parameter, but is actually an alias of it. Furthermore, that parameter must not be modified inside the callee. In such a case, it is enough to look at the final path-length of that parameter, held in l^k inside the callee, to determine the final path-length of v.

Note that since integer variables cannot share, the path-length of any $v \in Y$ of integer type is not affected by a call instruction (it will always be included in the US or UL sets).

We can now state the *correctness* results for our path-length analysis. Namely, we prove that the path-length constraints computed by our analysis include their concrete counterparts in their concretization. We start with the instructions.

PROPOSITION 46. *Let instruction* ins*, different from* call*, occur at program point q. We have*

$$ins_q \in \gamma(ins_q^{\mathbb{PL}}).$$

Then we consider the auxiliary path-length constraints for the method call.

PROPOSITION 47. *Let $\kappa.m(t_1, \ldots, t_p) : t$ be a method. We have*

$$args_{q,\kappa.m(t_1,\ldots,t_p):t} \in \gamma(args^{\mathbb{PL}}_{q,\kappa.m(t_1,\ldots,t_p):t})$$
$$select_{\kappa.m(t_1,\ldots,t_p):t} \in \gamma(select^{\mathbb{PL}}_{\kappa.m(t_1,\ldots,t_p):t})$$
$$makescope_{\kappa.m(t_1,\ldots,t_p):t} \in \gamma(makescope^{\mathbb{PL}}_{\kappa.m(t_1,\ldots,t_p):t}).$$

Hence we consider the operators over the path-length constraints.

PROPOSITION 48. *In the conditions of Definition* 44*, we have*

$$\gamma(pl_1); \gamma(pl_2) \subseteq \gamma(pl_1 ;^{\mathbb{PL}} pl_2)$$
$$\gamma(pl_1) \cup \gamma(pl_2) \subseteq \gamma(pl_1 \cup^{\mathbb{PL}} pl_2)$$
$$extend_{\kappa.m(t_1,\ldots,t_p):t}(\gamma(pl)) \subseteq \gamma(extend^{\mathbb{PL}}_{\kappa.m(t_1,\ldots,t_p):t}(pl)).$$

We now lift to our path-length polyhedra the notion of interpretation of Definition 17.

Definition 49. A *path-length interpretation* ι for P is a map from P's blocks into $\mathbb{PL}$. More precisely, if b is a block such that at its beginning there are l local variables and s stack elements and b is part of the body of a method $\kappa.m(t_1,\ldots,t_p):t$, then $\iota(b)\in\mathbb{PL}_{l,s\to l_o,s_o}$ where $l_o\geq l$ (new local variables might be declared in the body of the method), $s_o=0$ if $t=$ `void` and $s_o=1$ otherwise. The set of all path-length interpretations is written $\mathbb{I}^{\mathbb{PL}}$ and is ordered by the pointwise extension of $\leq$.

Hence we lift the definition of denotation of an instruction or block (Definition 19).

Definition 50. Let $\iota\in\mathbb{I}^{\mathbb{PL}}$. We define the *path-length denotations in* ι *of an instruction* ins which is not call as

$$[\![\mathsf{ins}]\!]_\iota^{\mathbb{PL}}=ins^{\mathbb{PL}}.$$

For call, let $m_i=\kappa_i.m(t_1,\ldots,t_p):t$ for $1\leq i\leq n$. We define

$$[\![\mathsf{call}\ m_1,\ldots,m_n]\!]_\iota^{\mathbb{PL}}=\bigcup\nolimits^{\mathbb{PL}}_{1\leq i\leq n} extend^{\mathbb{PL}}_{m_i}\Big(select^{\mathbb{PL}}_{m_i};^{\mathbb{PL}} makescope^{\mathbb{PL}}_{m_i};^{\mathbb{PL}}\iota(b_{m_i})\Big),$$

where b_{m_i} is the block where method m_i starts. The function $[\![\_]\!]_\iota^{\mathbb{PL}}$ is extended to blocks as

$$\left[\!\!\left[\ \boxed{\begin{array}{c}\mathsf{ins}_1\\ \cdots\\ \mathsf{ins}_w\end{array}}\ \rightrightarrows\ \begin{array}{c}b_1\\ \cdots\\ b_m\end{array}\right]\!\!\right]_\iota^{\mathbb{PL}}=\begin{cases}[\![\mathsf{ins}_1]\!]_\iota^{\mathbb{PL}};^{\mathbb{PL}}\cdots;^{\mathbb{PL}}[\![\mathsf{ins}_w]\!]_\iota^{\mathbb{PL}} & \text{if } m=0\\ [\![\mathsf{ins}_1]\!]_\iota^{\mathbb{PL}};^{\mathbb{PL}}\cdots;^{\mathbb{PL}}[\![\mathsf{ins}_w]\!]_\iota^{\mathbb{PL}};\big(\iota(b_1)\cup^{\mathbb{PL}}\cdots\cup^{\mathbb{PL}}\iota(b_m)\big) & \text{if } m>0.\end{cases}$$

We can finally define a path-length denotational semantics. A technical difficulty is that we cannot define it as the least fixpoint of a $T_P^{\mathbb{PL}}$ operator, since that fixpoint does not exist in general (the union of an infinite set of polyhedra might not be a polyhedron). Hence we content ourselves with a postfixpoint of that operator, that is, an interpretation $\bar{\iota}$ such that $T_P^{\mathbb{PL}}(\bar{\iota})\leq\bar{\iota}$. A postfixpoint can be computed in a finite number of iterations through a *widening* operator over polyhedra, which forces the analysis to converge [Cousot and Halbwachs 1978]. We actually use the more precise widening operator defined in Bagnara et al. [2005].

Definition 51. The *transformer* $T_P^{\mathbb{PL}}:\mathbb{I}^{\mathbb{PL}}\mapsto\mathbb{I}^{\mathbb{PL}}$ for P is defined as

$$T_P^{\mathbb{PL}}(\iota)(b)=[\![b]\!]_\iota^{\mathbb{PL}}$$

for every $\iota\in\mathbb{I}^{\mathbb{PL}}$ and block b of P. We define a *postfixpoint* $\mathcal{D}_P^{\mathbb{PL}}$ of $T_P^{\mathbb{PL}}$, computable in a finite number of iterations, by using the widening operator defined in Bagnara et al. [2005]. Note that this widening operator keeps the polyhedra closed. Hence we can define the *path-length semantics* of P as $\mathcal{D}_P^{\mathbb{PL}}$.

THEOREM 52. *The path-length semantics is correct with respect to the concrete denotational semantics of Section 5, that is,*

$$\mathcal{D}_P\leq\gamma(\mathcal{D}_P^{\mathbb{PL}}).$$

In this section, for simplicity, we have not considered exceptions. If exceptions are taken into account, as modeled in Section 5.3, then the path-length polyhedra are split into pairs of two polyhedra: the first polyhedron relates the output normal state to the input normal state. The second polyhedron relates the output exceptional state to the input normal state. Our implementation uses this technique to deal with programs with exceptions.

We have seen that the path-length might be infinite (Definition 24) and that ∞ is allowed in the models of a polyhedron (Definition 31). Nevertheless, the polyhedra build for each bytecode do not mention ∞ explicitly (Definitions 37 and 43) and the operators on such polyhedra (Definition 44) are standard and easily implementable, for instance, in terms of the operators available in the Parma Polyhedra Library [Bagnara et al. 2008]. Hence that library or a similar one can safely be used to implement the path-length analysis.

7. COMPILATION INTO CONSTRAINT LOGIC PROGRAMS

In this section we prove that the result of a path-length analysis can be used to translate a Java bytecode program into a constraint logic program [Jaffar and Maher 1994] over path-length polyhedra (*CLP*($\mathbb{PL}$)), whose termination entails the termination of the original Java bytecode program. It is important to remark that we assume a specialized semantics of CLP computations here, where variables are always bound to integer values [Spoto et al. 2009]. This means that we do not allow *free* variables in a call to a predicate. This is consistent with the fact that we model the path-length of the variables in a state, which assigns an integer value to all the variables in the state. For instance, in the *CLP*($\mathbb{PL}$) program.

$$\begin{aligned}
&\mathtt{p}(\check{x}) \ \mathtt{:-}\{\hat{y} \geq 0\},\ \ \mathtt{b}(\hat{y}).\\
&\mathtt{b}(\check{x}) \ \mathtt{:-}\{\check{x} = \hat{y} + 1,\ \hat{y} \geq 0\},\ \ \mathtt{b}(\hat{y}).
\end{aligned}$$

we assume that a call to predicate `p` leads to a call to predicate `b` with a given, nonnegative argument $\hat{y}$. That is, a specific value for $\hat{y}$ is chosen, provided that it is nonnegative, and the computation continues with `b`. This entails that any call to `p` terminates, while this is not the case with the traditional semantics of CLP, which allows partially constrained variables [Jaffar and Maher 1994].

From now on, we assume that the blocks of code have been decorated with a unique name, as in Figure 14. In that figure, we also report the names of some program points that we will use in the examples that follow.

Definition 53. Let P be a Java bytecode program. The *CLP*($\mathbb{PL}$) program P_{CLP} derived from P is built as follows. For each block

$$\boxed{b \,\Big|\, \begin{array}{c}\mathsf{ins}_1\\ \mathsf{ins}_2\\ \cdots\\ \mathsf{ins}_w\end{array}} \rightrightarrows \begin{array}{c} b_1\\ \cdots\\ b_m\end{array}$$

in P, let $c = [\![ins_1]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![ins_w]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}$. We generate the *CLP* clauses

$$\begin{aligned}
&b(v\check{ars}) \ :\!-\ c, b_1(v\hat{ars}).\\
&\cdots\\
&b(v\check{ars}) \ :\!-\ c, b_m(v\hat{ars}).
\end{aligned} \tag{5}$$

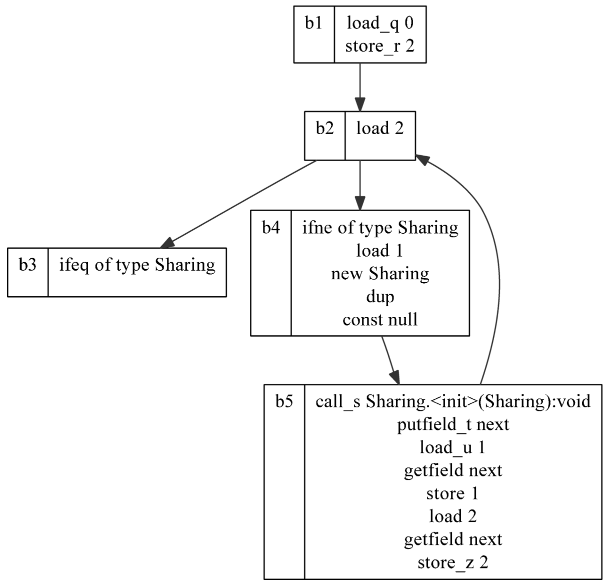

Fig. 14. The program in Figure 8, where each block is decorated with a unique name.

where $\check{vars}$ are the input local variables and stack elements at the beginning of block b and $\hat{vars}$ are the output local variables and stack elements at the end of block b (in some fixed order). Moreover, if $\mathsf{ins}_1 = \mathsf{call}_q\ m_1, \ldots, m_n$, where $m_i = \kappa_i.m(t_1, \ldots, t_p) : t$, then we also add a clause

$$b(\check{vars}) \ :\!-\ \left(args^{\mathbb{PL}}_{q,m_i} ;^{\mathbb{PL}} select^{\mathbb{PL}}_{m_i} ;^{\mathbb{PL}} makescope^{\mathbb{PL}}_{m_i}\right), b_{m_i}(\hat{l}^0, \ldots, \hat{l}^p). \qquad (6)$$

for each $1 \leq i \leq n$, where b_{m_i} is the block where method m_i begins.

The clauses (5) mimic the execution of block b, followed by the execution of one of its followers. The relation between the input state of b and that of its followers is approximated by the path-length constraint c of the code inside block b. Hence those clauses say that the execution of b from an input state σ leads to the execution of $b_1, \ldots, b_m$ from a state σ' where the variables in σ (seen as input variables) and those in σ' (seen as output variables) satisfy c. Note that no clause is generated in (5) for those blocks with no followers, since they cannot be part of a loop, so that they are not relevant for our termination analysis. If the first instruction ins_1 of block b is a call instruction (remember that we assume that call instructions can only occur at the beginning of a block), the clauses (5) assume a *complete* execution of that call, that is, they express a computation in which control has come back to the callee. This would not be enough to prove our correctness result (Theorem 56). This is because nontermination very often occurs as a consequence of an infinite recursion, so that we must also consider the case when a call does not complete its execution. To that purpose, we introduce the clauses (6). They mimic, explicitly, the execution of the callee. Namely, they single out from the stack the actual arguments of the call ($args^{\mathbb{PL}}$), then they check which dynamic target is selected ($select^{\mathbb{PL}}$), then they move the actual arguments from the stack to the lowest local variables ($makescope^{\mathbb{PL}}$), and they finally run the callee from block b_{m_i}. The latter starts its execution in a state where the stack is empty and the $p+1$ lowest local variables hold the actual arguments of the call.

Our translation into $CLP(\mathbb{PL})$ is similar in spirit to that in Albert et al. [2007a, 2008]. In both cases, a CLP program is constructed from the structure of the code, seen as a graph of blocks of code. The main difference is that they use the clauses (5), but they do not use the clauses (6). This second kind of clauses are meaningful for termination analysis, but not for cost analysis.

Example 54. Only one clause is generated for the block `b1` in Figure 14, whose instructions occur at program points that we call q and r, respectively,

$$\texttt{b1}(\check{l}^0,\check{l}^1\ )\texttt{:-}\ \left([\![load_q\ 0]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P};^{\mathbb{PL}}[\![store_r\ 2]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\right),\ \ \texttt{b2}(\hat{l}^0,\hat{l}^1,\hat{l}^2)\,.$$

which by Example 45 is

$$\texttt{b1}(\check{l}^0,\check{l}^1\ )\texttt{:-}\ \{\check{l}^0=\hat{l}^0,\check{l}^1=\hat{l}^1,\check{l}^0\geq 0,\check{l}^1\geq 0,\check{l}^0=\hat{l}^2\},\ \ \texttt{b2}(\hat{l}^0,\hat{l}^1,\hat{l}^2)\,.$$

Example 55. Consider block `b5` in Figure 14. At its beginning there are 3 local variables and 4 stack elements (Figure 8). We build two clauses for it. The first belongs to the set (5).

$$\begin{aligned}\texttt{b5}(\check{l}^0,\check{l}^1,\check{l}^2,\check{s}^0,\check{s}^1,\check{s}^2,\check{s}^3\ )\texttt{:-}\ \big(&[\![\mathsf{call}_s\ \texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing)}:\texttt{void}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P};^{\mathbb{PL}}\\&[\![\mathsf{putfield}_t\ \texttt{next}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P};^{\mathbb{PL}}[\![\mathsf{load}_u\ 1]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P};^{\mathbb{PL}}\cdots\\&\cdots;^{\mathbb{PL}}[\![\mathsf{store}_z\ 2]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\big),\ \ \texttt{b2}(\hat{l}^0,\hat{l}^1,\hat{l}^2)\,.\end{aligned}$$

The second is built since `b5` starts with a call instruction (with only one possible dynamic target). It is

$$\begin{aligned}\texttt{b5}(\check{l}^0,\check{l}^1,\check{l}^2,\check{s}^0,\check{s}^1,\check{s}^2,\check{s}^3\ )\texttt{:-}\big(&args^{\mathbb{PL}}_{s,\texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing):void}};^{\mathbb{PL}}\\&select^{\mathbb{PL}}_{\texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing):void}};^{\mathbb{PL}}\\&makescope^{\mathbb{PL}}_{\texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing):void}}\big),\\&b_{\texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing):void}}(\hat{l}^0,\hat{l}^1).\end{aligned}$$

Figure 15 shows the $CLP(\mathbb{PL})$ program generated from the blocks of method `expand` in Figure 14. Since that method calls the constructor of class `Sharing`, the last clause in Figure 15 links the code for `expand` with that for the constructor (not shown in the figure). It is interesting to observe that the last but one clause contains the constraint $\check{l}^2-1\geq\hat{l}^2$, that is, block `b5` strictly decreases the path-length of local variable 2 (variable `cursor` in Figure 1). Together with the fact that that variable has reference type and hence has nonnegative path-length, this is the key for a proof of termination for the method `expand`.

We can now state the correctness of our translation. Note that we assume that the CLP predicates are called with concrete integer values for the variables, according to our specialized semantics.

THEOREM 56. *Let P be a Java bytecode program and b a block of P. If the query $b(vars)$ has only terminating computations in P_{CLP}, for any fixed integer*

$$\texttt{b1}(\check{l}^0,\check{l}^1)\ \texttt{:-}\ \{\check{l}^0=\hat{l}^0,\check{l}^1=\hat{l}^1,\check{l}^0\geq 0,\check{l}^1\geq 0,\check{l}^0=\hat{l}^2\},\texttt{b2}(\hat{l}^0,\hat{l}^1,\hat{l}^2)\ .$$

$$\texttt{b2}(\check{l}^0,\check{l}^1,\check{l}^2)\ \texttt{:-}\ \left\{\begin{array}{c}\hat{s}^2=\hat{l}^2,\hat{s}^0=\check{l}^2,\check{l}^1=\hat{l}^1\\ \check{l}^0=\hat{l}^0,\hat{s}^0\geq 0,\check{l}^1\geq 0,\check{s}^0\geq 0\end{array}\right\},\texttt{b3}(\hat{l}^0,\hat{l}^1,\hat{l}^2,\hat{s}^0)\ .$$

$$\texttt{b2}(\check{l}^0,\check{l}^1,\check{l}^2)\ \texttt{:-}\ \left\{\begin{array}{c}\hat{s}^2=\hat{l}^2,\hat{s}^0=\check{l}^2,\check{l}^1=\hat{l}^1\\ \check{l}^0=\hat{l}^0,\hat{s}^0\geq 0,\check{l}^1\geq 0,\check{s}^0\geq 0\end{array}\right\},\texttt{b4}(\hat{l}^0,\hat{l}^1,\hat{l}^2,\hat{s}^0)\ .$$

$$\texttt{b4}(\check{l}^0,\check{l}^1,\check{l}^2,\check{s}^0)\ \texttt{:-}\ \left\{\begin{array}{c}\check{l}^0=\hat{l}^0,\check{l}^1=\hat{l}^1,\check{l}^2=\hat{l}^2,\check{s}^0\geq 1,\check{l}^0\geq 0\\ \check{l}^1\geq 0,\check{l}^2\geq 0,\hat{s}^0\geq 0,\check{s}^0=\hat{l}^2\\ \check{s}^0\geq 1,\check{l}^1=\hat{s}^0,\hat{s}^2=1,\hat{s}^1=1,\hat{s}^3=0\end{array}\right\},\texttt{b5}(\hat{l}^0,\hat{l}^1,\hat{l}^2,\hat{s}^0,\hat{s}^1,\hat{s}^2,\hat{s}^3)\ .$$

$$\texttt{b5}(\check{l}^0,\check{l}^1,\check{l}^2,\check{s}^0,\check{s}^1,\check{s}^2,\check{s}^3)\ \texttt{:-}\ \left\{\begin{array}{c}\check{l}^0=\hat{l}^0,\hat{l}^1\geq 1,\check{l}^1+\check{s}^1\geq \hat{l}^1,\\ \check{l}^2-1\geq \hat{l}^2,\hat{l}^2\geq 0,\check{l}^0\geq 0,\check{l}^1\geq 0\\ \check{l}^2\geq 0,\check{s}^0\geq 0,\check{s}_1=\check{s}_2,\check{s}^2\geq 1,\check{s}^3\geq 0\end{array}\right\},\texttt{b2}(\hat{l}^0,\hat{l}^1,\hat{l}^2)\ .$$

$$\texttt{b5}(\check{l}^0,\check{l}^1,\check{l}^2,\check{s}^0,\check{s}^1,\check{s}^2,\check{s}^3)\ \texttt{:-}\{\check{s}^2=\hat{l}^0,\check{s}^3=\hat{l}^1\},b_{\texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing):void}}(\hat{l}^0,\hat{l}^1)\ .$$

Fig. 15. The *CLP*($\mathbb{PL}$) program generated from the Java bytecode method `expand` in Figure 14. Block $b_{\texttt{Sharing.}\langle\texttt{init}\rangle\texttt{(Sharing):void}}$ is the first block of the code of the constructor of class `Sharing`.

values for vars, then all executions of a Java Virtual Machine started at block b terminate.

PROOF. We prove this result by contradiction. That is, we prove that if there is an execution of the Java Virtual Machine from block b that diverges, according to the operational semantics of Section 5.1, then the query $b(vars)$ has a divergent computation in P_{CLP} for some fixed integer values for $vars$.

Let hence

$$\sigma_1 \stackrel{\mathsf{ins}_1}{\rightarrow} \sigma_2 \stackrel{\mathsf{ins}_2}{\rightarrow} \cdots \stackrel{\mathsf{ins}_{k-1}}{\rightarrow} \sigma_k \stackrel{\mathsf{ins}_k}{\rightarrow} \cdots \tag{7}$$

be an infinite operational execution of the Java Virtual Machine from block b, starting at a state σ_1. The states in the sequence are those that are, at each step, on top of the activation stack of the Java Virtual Machine. Instruction ins_k is the instruction which makes the state on top of the activation stack evolve from σ_k to σ_{k+1}. Note that in general we have $[\![ins_k]\!]_{\mathcal{D}_P}(\sigma_k)\neq\sigma_{k+1}$ since, when ins_k is the last instruction of a method m, state σ_{k+1} is derived from σ_h, which was on top of the activation stack at the moment of the last call to m, by replacing the actual parameters with the return value (Definition 14). That call was executed by some $\mathsf{call}\ m_1,\ldots,m_n$ instruction in the program, with $m=m_i$ for some $0\leq i\leq n$. In such a case, we can identify a portion of (7). We have

$$\sigma_h \stackrel{\mathsf{args}_m}{\rightarrow} \sigma_{h+1} \stackrel{\mathsf{select}_m}{\rightarrow} \sigma_{h+2} \stackrel{\mathsf{makescope}_m}{\rightarrow} \sigma_{h+3}\cdots\cdots\sigma_k \stackrel{\mathsf{ins}_k}{\rightarrow} \sigma_{k+1} \tag{8}$$

where σ_h is the top of the activation stack at the moment of the last activation of m and ins_k terminates that activation. By the equivalence of our denotational and operational semantics (Theorem 23), we know that

$$\sigma_{k+1}=extend_m\left(\{select_m\};\{makescope_m\};\mathcal{D}_P(b_m)\right)(\sigma_h)$$

and, since our language is deterministic, we have

$$\sigma_{k+1}=\bigcup_{1\leq i\leq n} extend_{m_i}\left(\{select_{m_i}\};\{makescope_{m_i}\};\mathcal{D}_P(b_{m_i})\right)(\sigma_h)$$

that is $[\![\text{call } m_1, \cdots, m_n]\!]_{\mathcal{D}_P}(\sigma_h) = \sigma_{k+1}$. Hence we can systematically rewrite each such subsequence in (7) into a subsequence

$$\sigma_h \stackrel{\text{call } m_1,\cdots,m_n}{\rightarrow} \sigma_{k+1}.$$

Let

$$\sigma'_1 \stackrel{\text{ins}'_1}{\rightarrow} \sigma'_2 \stackrel{\text{ins}'_2}{\rightarrow} \cdots \stackrel{\text{ins}'_{k-1}}{\rightarrow} \sigma'_k \stackrel{\text{ins}'_k}{\rightarrow} \cdots \tag{9}$$

be the resulting, still infinite sequence. We now have

$$[\![\text{ins}'_k]\!]_{\mathcal{D}_P}(\sigma'_k) = \sigma'_{k+1} \tag{10}$$

for every $k \geq 0$. This sequence can still contain instructions args_m, but they must correspond to activations of method m that do not reach completion in (9). Since a call instruction can only occur at the beginning of some block b, the sequence (9) must have as a prefix

— $\sigma'_1 \stackrel{\text{ins}_1}{\rightarrow} \sigma'_2 \cdots \sigma'_w \stackrel{\text{ins}_w}{\rightarrow} \sigma'_{w+1} \cdots$, where $b = \boxed{\begin{array}{c}\text{ins}_1\\ \text{ins}_2\\ \cdots\\ \text{ins}_w\end{array}}$;

— or $\sigma'_1 \stackrel{\text{args}_{m_i}}{\rightarrow} \sigma'_2 \stackrel{\text{select}_{m_i}}{\rightarrow} \sigma'_3 \stackrel{\text{makescope}_{m_i}}{\rightarrow} \sigma'_4$, where $b = \boxed{\begin{array}{c}\text{call}_{m_1,\ldots,m_n}\\ \cdots\end{array}}$ and $1 \leq i \leq n$.

After that prefix, we will see another prefix. In the first case the new prefix will correspond to a block b' among the successors of b; in the second case, it will correspond to the beginning b_{m_i} of method m_i. By Definition 53, in the first case P_{CLP} contains the clause

$$b(v\check{ars}) \ :\!-\ \left([\![\text{ins}_1]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\text{ins}_w]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\right), b'(v\hat{ars}).$$

and in the second case it contains the clause

$$b(v\check{ars}) \ :\!-\left([\![\text{args}_{m_i}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} [\![\text{select}_{m_i}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} [\![\text{makescope}_{m_i}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\right), b_{m_i}(\hat{l}^0, \ldots, \hat{l}^p).$$

If we continue unwinding the infinite sequence (9), we hence find an infinite sequence of clauses of P_{CLP}

$$b_1(v\check{ars}_1) \ :\!-\left([\![\text{ins}'_1]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\text{ins}'_{w_1}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\right), b_2(v\hat{ars}_2).$$

$$b_2(v\check{ars}_2) \ :\!-\left([\![\text{ins}'_{w_1+1}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\text{ins}'_{w_2}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\right), b_3(v\hat{ars}_3).$$

$$\vdots$$

$$b_t(v\check{ars}_t) \ :\!-\left([\![\text{ins}'_{w_{t-1}+1}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\text{ins}'_{w_t}]\!]^{\mathbb{PL}}_{\mathcal{D}^{\mathbb{PL}}_P}\right), b_{t+1}(v\hat{ars}_{t+1}).$$

$$\vdots$$

where $b(v\check{ars}) = b_1(v\check{ars}_1)$. This is not enough to conclude that P_{CLP} has a divergent computation from the query $b(v\check{ars})$, since a CLP computation stops when its constraint store is unsatisfiable. Since the unification of the CLP atom $b_t(v\hat{ars}_t)$ with the atom $b_t(v\check{ars}_t)$ corresponds to the $;^{\mathbb{PL}}$ operation (renaming of the variables into new overlined variables and existential quantification), then we still have to prove that, for every $t \geq 1$, the constraint store

$$cs_t = [\![\mathsf{ins}'_1]\!]^{\mathbb{PL}}_{\mathcal{D}_P^{\mathbb{PL}}} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\mathsf{ins}'_{w_1}]\!]^{\mathbb{PL}}_{\mathcal{D}_P^{\mathbb{PL}}} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\mathsf{ins}'_{w_{t-1}+1}]\!]^{\mathbb{PL}}_{\mathcal{D}_P^{\mathbb{PL}}} ;^{\mathbb{PL}} \cdots ;^{\mathbb{PL}} [\![\mathsf{ins}'_{w_t}]\!]^{\mathbb{PL}}_{\mathcal{D}_P^{\mathbb{PL}}}$$

is satisfiable. By the correctness of our path-length analysis (Theorem 52) and by Propositions 46, 47, and 48, we conclude that

$$[\![\mathsf{ins}'_1]\!]_{\mathcal{D}_P} ; \cdots ; [\![\mathsf{ins}'_{w_1}]\!]_{\mathcal{D}_P} ; \cdots ; [\![\mathsf{ins}'_{w_{t-1}+1}]\!]_{\mathcal{D}_P} ; \cdots ; [\![\mathsf{ins}'_{w_t}]\!]_{\mathcal{D}_P} \in \gamma(cs_t) \qquad (11)$$

and by Eq. (10) we conclude that

$$([\![\mathsf{ins}'_1]\!]_{\mathcal{D}_P} ; \cdots ; [\![\mathsf{ins}'_{w_1}]\!]_{\mathcal{D}_P} ; \cdots ; [\![\mathsf{ins}'_{w_{t-1}+1}]\!]_{\mathcal{D}_P} ; \cdots ; [\![\mathsf{ins}'_{w_t}]\!]_{\mathcal{D}_P})(\sigma'_1) = \sigma'_{w_t+1}.$$

By (11) and Definition 33 this entails that

$$(\check{len}(\sigma'_1) \cup \hat{len}(\sigma'_{w_t+1})) \models cs_t,$$

that is, cs_t has a model and is hence satisfiable. Note that a model provides concrete integer values to each input variable and the existential operator used by $;^{\mathbb{PL}}$ requires the existence of concrete integer values for the variables at each predicate call. Hence we have found a divergent computation according to our specialized semantics. □

Let b_{start} be the initial block of our Java bytecode program P. Once the $CLP(\mathbb{PL})$ program P_{CLP} is built from P, we can use a *termination prover* for (constraint) logic programs to prove the termination of P_{CLP} from $b_{start}(\check{vars})$, and hence (Theorem 56) that of P from b_{start}.

We use the BinTerm termination prover. Compared to traditional logic programming termination provers, BinTerm deals with integer valued variables instead of nonnegative integer valued variables and takes advantage of the specialized operational semantics of $CLP(\mathbb{PL})$. The prover (see Algorithm 1) relies on the two static analysis techniques summarized next.

The first one combines closure computation with local ranking functions, as in Codish and Taboch [1999], Dershowitz et al. [2001], Lee et al. [2001], Codish et al. [2005], and Avery [2006]. We use two abstract domains: convex polyhedra and monotonicity constraints [Brodsky and Sagiv 1989] augmented with bounds. For each domain, the binary unfoldings of the abstraction of P_{CLP} are computed. Then for each binary recursive rule in the unfoldings, we try to detect a local affine ranking function.

The second technique is a specialization of that in Mesnard and Serebrenik [2008]. The call graph of P_{CLP} is decomposed into its maximal strongly connected components (*SCC*s). For each predicate in each intracomponent, a global parametric affine ranking function is defined so that it takes nonnegative values and decreases of a fixed amount from the head of each clause to its body. Then the existence of such an affine ranking function is decided by linear programming. The last part of Algorithm 1, from line 10, could include the search for more sophisticated ranking functions, as proposed, for instance, in Cousot [2005].

We have actually used an improvement of the first technique which gives better results in some cases. The idea is that, whenever predicate b in a binary

program	M	B	PR	PA	PL	proof	TE	LP	N	S
`Nested`	4	724	158	55	60	179	4	1/1	1	0
`Numerical1`	5	635	144	65	90	445	5	1/1	1	0
`Numerical3`	5	852	154	61	83	212	4	0/1	0	0
`Factorial`	5	741	159	49	43	101	5	1/1	1	0
`Ackermann`	5	765	144	57	78	222	5	1/1	1	0
`Diff`	5	805	165	71	577	12118	5	1/1	1	1
`BubbleSort`	5	804	153	70	172	660	5	1/1	1	1
`Double`	5	749	147	53	49	218	5	1/1	1	0
`Numerical2`	6	675	170	64	185	140	6	1/1	1	0
`Exc`	6	762	150	75	108	132	6	1/1	1	0
`FactSum`	6	773	151	46	70	116	6	2/2	2	0
`Hanoi`	7	874	168	88	318	216	5	1/1	1	0
`Sharing`	7	855	161	112	169	115	7	1/1	0	1
`BTree`	7	845	162	85	150	135	7	2/2	1	1
`FactSumList`	8	844	164	78	84	156	8	2/2	1	1
`Init`	10	811	150	68	34	109	8	0/2	0	0
`BinarySearchTree`	10	921	173	120	158	137	10	1/1	0	1
`Virtual`	11	908	174	107	202	108	11	2/2	1	1
`ListInt`	11	981	189	178	359	292	11	5/5	0	5
`List`	11	1044	188	256	462	213	11	5/5	0	5

Fig. 16. The termination analyses of some programs. Times are in milliseconds. *M* is the number of methods of the program, *B* is the number of its bytecodes. *PR* is the time for the preprocessing of the program, *PA* is the time for the preliminary analyses, *PL* is the time for the path-length analysis, *proof* is the time to find a proof with BINTERM, *TE* is the number of methods whose termination is proved, *LP* is the number of loops whose termination is proved, *N* is the number of loops whose termination is proved by using numerical arguments, *S* is the number of loops whose termination is proved by using arguments related to dynamic data structures in memory.

recursive rule of the form $b(v\check{a}rs)\ \texttt{:-}c, b(v\hat{a}rs')$ is called, some invariant might hold for the variables $v\check{a}rs$, as a consequence of the execution of the predicates of the program which have been called before b. This invariant can be useful to prove the termination of b. For this reason, we compute a *call contexts analysis* inspired by Gabbrielli and Giacobazzi [1994] and Codish and Taboch [1999] for the predicates in the binary unfolding of the program and use the resulting invariants to improve the quality of the termination proof for the recursive rules. As an example, consider the following *CLP*($\mathbb{PL}$) program, already unfolded in its binary form.

$$
\begin{aligned}
\texttt{entry} &\texttt{:-}\ \{\hat{y} \geq 0\},\ \ \texttt{p}(\hat{y}).\\
\texttt{p}(\check{x}) &\texttt{:-}\ \{\check{x} = \hat{y} + 1,\ \hat{y} \geq 0\},\ \ \texttt{p}(\hat{y}).\\
\texttt{p}(\check{x}) &\texttt{:-}\ \{\check{x} \leq -1,\ \hat{y} = \check{x}\},\ \ \texttt{p}(\hat{y}).
\end{aligned}
$$

The entry point of the program is predicate `entry`. Predicate `p` does not terminate in general, because of its second clause. However, any run from predicate `entry` terminates, since $\texttt{p}(\check{x})$ is invoked with a call context $\check{x} \geq 0$ which disables its second clause. Situations like this are found, for instance, in `BubbleSort` and `Double` in Figure 16. As another example, the first test of BINTERM (lines 1–2 of

Algorithm 1. BinTerm: A Termination Test

Require. a program P_{CLP}
Ensure. if BinTerm returns **true** then we have a termination proof
1: $P_1^* \leftarrow$ the binary unfoldings of P_{CLP} with respect to the polyhedral domain
2: **if** for each recursive rule of P_1^* there is an affine ranking function **then**
3: **return true**
4: **else**
5: $P_2 \leftarrow$ the abstraction of P_{CLP} with respect to the bounded monotonicity domain
6: $P_2^* \leftarrow$ the binary unfoldings of P_2
7: **if** for each recursive rule of P_2^* there is an affine ranking function **then**
8: **return true**
9: **else**
10: **if** for each *SCC* of P_{CLP}, for each predicate in this component, there is an affine ranking function **then**
11: **return true**
12: **else**
13: **return unknown**
14: **end if**
15: **end if**
16: **end if**

Algorithm 1) proves the termination of the program

$$\begin{aligned} \texttt{entry} \texttt{:-}\ & \{true\},\ \ \texttt{div2}(\hat{x}).\\ \texttt{div2}(\check{x}) \texttt{:-}\ & \{\check{x} = 2 * \hat{x}, \check{x} \geq 1\},\ \ \texttt{div2}(\hat{x}). \end{aligned}$$

while the second test of BinTerm (lines 5–7) fails, also by using call contexts. On the other hand, the presence of that second test is crucial for proving the termination of a predicate with two arguments, decreasing with respect to a lexicographical ordering.

$$\begin{aligned} \texttt{entry} \texttt{:-}\ & \{true\},\ \ \texttt{lex}(\hat{x}_1, \hat{x}_2).\\ \texttt{lex}(\check{x}_1, \check{x}_2) \texttt{:-}\ & \{\hat{x}_1 \geq 0, \hat{x}_2 \geq 0, \check{x}_2 \geq 0, \check{x}_1 \geq 1 + \hat{x}_1\},\ \ \texttt{lex}(\hat{x}_1, \hat{x}_2).\\ \texttt{lex}(\check{x}_1, \check{x}_2) \texttt{:-}\ & \{\hat{x}_1 \geq 0, \hat{x}_2 \geq 0, \check{x}_1 = \hat{x}_1, \check{x}_2 \geq 1 + \hat{x}_2\},\ \ \texttt{lex}(\hat{x}_1, \hat{x}_2). \end{aligned}$$

Finally, the example

$$\begin{aligned} \texttt{entry} \texttt{:-}\ & \{true\},\ \ \texttt{gcd}(\hat{x}_1, \hat{x}_2).\\ \texttt{gcd}(\check{x}_1, \check{x}_2) \texttt{:-}\ & \{\check{x}_1 \geq 1, \check{x}_2 \geq 1, \hat{x}_1 = \check{x}_1, \hat{x}_2 = \check{x}_2\},\ \ \texttt{gcd2}(\hat{x}_1, \hat{x}_2).\\ \texttt{gcd2}(\check{x}_1, \check{x}_2) \texttt{:-}\ & \{\check{x}_1 \geq \check{x}_2 + 1, \hat{x}_1 = \check{x}_1 - \check{x}_2, \hat{x}_2 = \check{x}_2\},\ \ \texttt{gcd}(\hat{x}_1, \hat{x}_2).\\ \texttt{gcd2}(\check{x}_1, \check{x}_2) \texttt{:-}\ & \{\check{x}_2 \geq \check{x}_1 + 1, \hat{x}_1 = \check{x}_1, \hat{x}_2 = \check{x}_2 - \check{x}_1\},\ \ \texttt{gcd}(\hat{x}_1, \hat{x}_2). \end{aligned}$$

is proved terminating thanks to the last test of BinTerm (line 10) and with the help of the call context $\check{x}_1 \geq 1, \check{x}_2 \geq 1$ which holds for any internal call to $\texttt{gcd2}(\check{x}_1, \check{x}_2)$.

8. EXPERIMENTS

In this section we describe our implementation of the termination analyzer for full Java bytecode and report some experimental results.

The analyzer [Spoto et al. 2008] is the combination of the JULIA generic static analyzer for Java bytecode [Spoto 2008a], written in Java, with the BINTERM termination prover for constraint logic programs over numerical constraints, written in Prolog. We now describe the different phases of the analysis in their order of application.

(1) The user specifies the `.class` file containing the `main()` method of the application under analysis. Alternatively, in *library mode*, the user specifies the set of `.class` files whose public methods must be analyzed. In both cases, JULIA also analyzes all reachable methods, which typically requires to load other classes than those specified by the user. This phase is implemented through an *application extraction* algorithm based on Palsberg and Schwartzbach [1991]. It is an instance of class analysis and is hence used also to compute the set of possible runtime targets for each method call (Section 3). The `.class` files are parsed using the BCEL library for bytecode manipulation (`http://jakarta.apache.org/bcel`). Most native methods are replaced with handwritten code which simulates their behavior;
(2) Number and types of local variables and stack elements at each program point are computed through the Kindall algorithm [Lindholm and Yellin 1999];
(3) Aliasing, pair-sharing, and cyclicity analyses are computed using the corresponding abstract domains implemented inside JULIA. Our pair-sharing analysis is described in Secci and Spoto [2005] and is computed in reduced product with purity information (as in Genaim and Spoto [2008]); our cyclicity analysis is described in Rossignoli and Spoto [2006]. All these analyses are computed using abstract versions of the denotational semantics of Section 5. These denotational analyses are focused at internal program points using *magic-sets* [Payet and Spoto 2007]. Pair-sharing and cyclicity abstract domain elements are implemented through binary decision diagrams, using the BUDDY library (`http://sourceforge.net/projects/buddy`). The `null` pointer [Spoto 2008b] and class initialization analyses are also performed since they might be useful for the precision of the subsequent path-length analysis (Section 2);
(4) Path-length analysis is computed with our domain described in Section 6. Abstract domain elements are closed polyhedra and have been implemented through the PPL (Parma Polyhedra Library) [Bagnara et al. 2008]. When the complexity of the operations over the polyhedra explodes (for instance, because of a high number of local variables) a worst-case assumption is made, that is, the path-length of the highest variables is not approximated;
(5) A constraint logic program is generated from the Java bytecode program, by using the result of our path-length analysis (Section 7), and is then sourced to the BINTERM termination prover for constraint logic programs.

The latter looks for appropriate termination proofs (Section 7). The results of the analysis are finally provided to the user.

Our experiments have been performed on a Linux machine based on a 64-bits dual-core AMD Opteron processor 280 running at 2.4 Ghz, with 2 gigabytes of RAM and 1 megabyte of cache, by using Sun Java Development Kit version 1.5 and SICStus Prolog version 3.12.8.

Figure 16 reports the results of the termination analysis of some small programs which are distributed together with JULIA. The source code of these programs is available, but we have not used it for the analysis, which is performed over the compiled bytecode. Programs `Factorial`, `Diff`, `BubbleSort`, `FactSum`, `Hanoi`, `BTree`, `FactSumList`, and `BinarySearchTree` are taken from Albert et al. [2007a, 2008], while `Numerical1`, `Numerical2`, and `Numerical3` are taken from Cook et al. [2006a] and contain numerical loops only (that in `Numerical3` can actually diverge). The others have been chosen in order to test the practicability of the analysis, since their termination depends on cycles, nested cycles, iterations over one or multiple data structures, and exceptions. The standard Java classes are not included in the analysis, which means that the calls to the libraries are assumed to terminate. For each program we report the number of methods, the number of bytecodes, the time spent for preprocessing (phases (1) and (2) before), the time spent for the preliminary analyses (phase (3)), the time spent for path-length analysis (phase (4)), and the time spent while looking for a termination proof through BINTERM (phase (5)). All times are in milliseconds. Figure 16 reports how many methods have been proved to terminate. In all these programs a proof of termination is found for every terminating method, so that the analysis is actually optimal. For `Init`, there are 2 methods whose termination could not be proved, since they actually diverge. They are the constructor and the static initializer of the class `A` shown in Section 2. Figure 16 then reports how many loops are proved to terminate. By *loop* we mean a strongly connected component of blocks of code containing a cycle. Hence nested Java loops result in one loop only. Similarly, mutually recursive methods form one loop only. Figure 16 reports also the number of such loops whose termination has been proved by using numerical arguments and the number of loops whose termination has been proved by reasoning over dynamic data structures. In the first case, the ranking function for the loop uses variables of the program whose type is `int`; in the second case, it uses variables of reference type. Since ranking functions in general use more than one variable, it is possible for a loop to be proved by using both numerical and structural arguments.

Figure 17 reports the results of the termination analysis of larger programs. We have chosen such programs so that they do not use native methods of the standard Java library beyond those that we have already specified, nor reflection, nor multithreading (these limitations are discussed in Section 10). This figure shows that our analysis scales to programs of up to 1000 methods, computing nontrivial calculations. `RayTracer` is a ray-tracing program involving complex floating-point calculations. The source code of this program is not available to us. `NQueens` is a solver of the *n*-queens problem, based on a library

program	M	B	PR	PA	PL	proof	TE	LP	N	S
RayTracer	243	13680	1191	7209	17678	13309	232	8 over 19	5	4
NQueens	480	33533	1464	9232	42191	54910	412	33 over 80	33	14
Kitten	1201	66941	2664	34856	88909	105365	1168	44 over 98	38	15

Fig. 17. The termination analyses of some larger programs. Times are in milliseconds. *M* is the number of methods of the program, *B* is the number of its bytecodes. *PR* is the time for the preprocessing of the program, *PA* is the time for the preliminary analyses, *PL* is the time for the path-length analysis, *proof* is the time to find a proof with BinTerm, *TE* is the number of methods whose termination is proved, *LP* is the number of loops whose termination is proved, *N* is the number of loops whose termination is proved by using numerical arguments, *S* is the number of loops whose termination is proved by using arguments related to dynamic data structures in memory.

```
int AbstractStringBuilder.stringSizeOfInt(int)
AbstractStringBuilder AbstractStringBuilder.append(int)
AbstractStringBuilder AbstractStringBuilder.append(String)
boolean Class.desiredAssertionStatus()
String(char[],int,int)
void String.getChars(int,int,char[],int)
StringBuffer StringBuffer.append(String)
StringBuilder StringBuilder.append(String)
String StringBuilder.toString()
RayTracingEngine.closestIntersection(Ray,Surface[]):Intersection
RayTracingEngine.render(Surface[],Camera,Light[],int,int,...):RGB[][]
```

Fig. 18. The methods called by `RayTracer` and whose termination is not proved by our analyzer.

for binary decision diagrams. This library is included in the analysis. `Kitten` is a didactic compiler for a simple imperative object-oriented language, used by the first author for his classes. It uses highly cyclical dynamic data structures, such as abstract trees (with sharing subtrees) and graphs of basic blocks. In all these examples, the standard Java libraries have been included in the analysis. The number of methods whose termination is not proved does not include the methods that are not proved to terminate only because they call another method whose termination is not proved. That is, we only count the methods that *introduce* possible nontermination according to our analyzer.

Figure 18 shows the methods called by `RayTracer` and whose termination is not proved by our analyzer. We have investigated why our analyzer fails to prove their termination. Method `AbstractStringBuilder.stringSizeOfInt(int)` iterates over the elements of an array stored in a field of an object. However, instead of loading that array on the stack once and then using that reference during the iteration, it reloads the array at every iteration. As a consequence, our analyzer does not understand that the length of the array does not change across iterations and that the number of iterations is consequently bound from above. Method `Class.desiredAssertionStatus()` contains the following instructions.

```
 43:  astore_3
 44:  aload_2
 45:  monitorexit
 46:  aload_3
 47:  athrow
Exception table:
```

from	to	target	type
18	42	43	any
43	46	43	any

Our analyzer thinks that the `monitorexit` instruction at line 45 might throw an exception which leads back to line 43, hence entering an infinite loop. We do not know if this can ever be the case. The proof is not easy since `monitorexit` can throw an exception when it is invoked on `null` (but this is already excluded by our `null` pointer analysis here) but also when the rules for correct bracketing with `monitorenter` are not satisfied [Lindholm and Yellin 1999]. Our analyzer does not include at the moment any analysis for this correct bracketing. Such a recursive exception handler looks, however, very strange to us and might actually be a bug in the standard Java libraries. The methods in Figure 18 dealing with strings and related classes are not proved to terminate since they might throw a `StringIndexOutOfBoundsException`, whose constructor calls back the methods for creating and appending strings. Such call-backs might throw again an exception and so on infinitely often. We suppose that such behavior cannot happen in practice, but our analyzer fails to prove it. Method `closestIntersection` terminates because of some geometrical reasoning about rays of light, as we have checked by decompiling the bytecode. Our analyzer has no hope of proving this. Method `render` contains a large number of local variables. The complexity of our analysis explodes so that a worst-case assumption is made for the method, whose termination is not proved.

Our analyzer fails to prove the termination of some methods of the standard Java library also for the other two test programs. Furthermore, it also fails to prove the termination of some methods of the application. For `NQueens`, the methods which are not proved to terminate are mainly those of the library for binary decision diagrams that perform bitwise operations, since binary decision diagrams are efficiently represented through bitmaps. To prove their termination, one needs a precise model of such bitwise operations, which our analyzer currently lacks (as well as other analyzers; see the same limitation for Terminator in Cook et al. [2006a]). For the `Kitten` compiler, our analyzer fails to prove that methods dealing with the graph of basic blocks of code actually terminate. This is a limitation of our analysis: those methods terminate since a block is never visited twice but this is not captured by our analysis (Section 10). Other methods are not proved to terminate because of some imprecision in the noncyclicity analysis: the analyzer fails to prove that the hierarchy of classes in the compiled program is noncyclical. Noncyclicity of this hierarchy is guaranteed by the semantical analysis phase of the compiler, but our analyzer is not clever enough to understand this.

9. RELATED WORK

There is a huge literature on termination analysis of computer programs and on the formal specification of the semantics and of the analysis of Java or Java bytecode. Here we provide a terse survey of the most relevant papers in those areas.

Termination analysis for logic and functional languages. Automatic termination of logical rules was studied in Ullman and Gelder [1988]. Plümer [1990] describes an early attempt to automate termination proofs for Prolog. The first results in this stream of research are summarized in De Schreye and Decorte [1994]. Termination of a logic program has also been proved through the *binary unfoldings* of the program, a set of binary clauses whose termination can be more easily assessed [Codish and Taboch 1999]. Techniques exist that infer classes of input arguments for which termination is guaranteed, rather than just proving termination for a class of inputs [Mesnard 1996; Genaim and Codish 2005; Mesnard and Bagnara 2005]. In Manolios and Vroon [2006b], static analysis and theorem proving are used to approximate in a finite way all the concrete calls among functions in a pure functional program. The result of this approximation is a set of *calling context graphs*. Using these graphs, termination is proved by arguments relying on some decreasing measures on the function parameters. This technique is improved in Manolios and Vroon [2006a] by issuing queries to a theorem prover. If the latter can solve the queries in a fixed amount of time, the precision of the analysis is improved. The use of theorem proving also allows one to get counterexamples when the analysis fails to prove termination. More recently, with the aim of improving the efficiency of the analysis, termination of term rewrite systems has been encoded into a Boolean formula which is satisfiable if and only if there exists a lexicographic path order or a multiset path order [Codish 2007]. The experiments are very promising. APROVE [Giesl et al. 2006] is one of the most advanced systems for automated termination proofs of term rewrite systems, which can also analyze Prolog and Haskell programs [Schneider-Kamp et al. 2006]. Other tools, specialized for logic programs, are cTI by Mesnard [1996], HASTA-LA-VISTA by Serebrenik and De Schreye [2002], POLYTOOL by Nguyen and De Schreye, TALP by E. Ohlebusch et al. [2000], TERMILOG by N. Lindenstrauss et al. [1997], and TERMINWEB by Taboch et al. [2002].

Termination analysis for imperative programs. Automatic termination analysis of imperative programs goes back to Floyd's seminal work [Floyd 1967]. After many years of research, it is mature enough now to apply to Java bytecode [Albert et al. 2007a, 2008] and large system code written in the C language, as the TERMINATOR system shows [Cook et al. 2006b] (see the detailed discussion in Section 1). Termination of the imperative reversal algorithm of some special kind of cyclic lists, called *panhandle* lists, is proved in Loginov et al. [2006]. A panhandle list is a cyclical list whose starting node is not part of the cycle. This is normally considered a complex problem of termination analysis and our analysis does not prove its termination. It must be noted that termination has been proved in Loginov et al. [2006] through very specific reasonings about the kind of data structure at hand (addition of ad-hoc *instrumentation relations*), while we aim at a generic and automatic termination analysis. In Bouajjani et al. [2006] counters are used to reason about the size of every region between two sharing points in one selector linked data structures, that is, again, linked lists. Counter automata are used as abstract models of the programs. This technique is used to prove termination of two sorting algorithms. The use of counters might be similar to our use of path-lengths, but their counters measure the distance between two sharing points in a list, while the path-length is the length of the maximal chain of pointers for any possible kind of data structure. The limits of their work is that only linked lists are considered. Moreover, function calls are not supported. The problem with function calls is that one needs information about sharing and *purity* [Salcianu and Rinard 2005; Genaim and Spoto 2008] of their arguments in order to model the effects of the calls on the heap [Chang and Leino 2005]. In Definition 44 we use such information to approximate method calls. In Brotherston et al. [2008], termination is proved by looking for cyclicity in the Hoare-like proof tree of the program, constructed by suitable execution rules over separation logic [Reynolds 2000; Ishtiaq and O'Hearn 2001]. The only considered data structures are lists. Function calls are not considered. By a careful choice of the predicates of separation logic, this technique can also prove the termination of the panhandle list reversal. Note that we prove termination of the program in Figure 5, which uses trees rather than flavors of lists, and that we support functions. Nevertheless, the results in Loginov et al. [2006], Berdine et al. [2006], Bouajjani et al. [2006], and Brotherston et al. [2008] show that termination analysis, tied to a specific data structure, leads to more precise results than does a general approach such as ours. For instance, it proves the termination of the panhandle list reversal, where our analysis fails.

Termination of concurrent programs. Podelski and Rybalchenko [2007] prove termination of generic concurrent programs working over integers. It is not clear how this work can be generalized to deal with dynamically allocated data structures in the heap, since sharing allows one process to modify the data of another process and this effect should be somehow modeled. The complexity of the concurrent update of memory should also be modeled, by using the results of Manson and Pugh [2001] and Manson et al. [2005]. Analysis of concurrent Java is also tackled in Cook et al. [2007]. They prove the termination of a thread by providing an abstraction of the behavior of all other concurrent threads (the environment). This abstraction can then be refined on the basis of counterexamples found during the proof. The technique might not terminate in general. They only consider the case of a finite and fixed number of threads. The generalization to the case of an unbounded number of dynamically created threads might be more difficult than it seems. Although all examples only use primitive types, there is a small comment at the end of Cook et al. [2007, page 327] saying that they have augmented their analysis with *some* data structures on the heap. We do not know which data structures have been considered and how they have been modeled in the analysis. There is no correctness proof nor example of this last augmented analysis.

Termination proofs based on nonlinear invariants. In some cases, programs terminate because some nonlinear quantity decreases over a well-founded domain. For that purpose, recent research has developed new techniques that prove termination of loops using nonlinear expressions. Bradley et al. [2005] build finite difference trees for expressions. This only works when such expressions have finite trees. Cousot [2005] builds polynomial ranking functions of nonlinear loops. It is limited to expressions that can be approximated by sums of squares and it requires heavy floating point calculations. Babic et al. [2007] proves termination by checking for possible divergence to infinite of every variables inside loops. The authors say that their technique proves termination in more cases than Bradley et al. [2005] and Cousot [2005], without requiring heavy floating point calculations. While nonlinear expressions are important for the termination of programs dealing with integer variables, it is not clear to us that they also contribute to the proof of termination of programs dealing with dynamic data structures in the heap.

Termination of floating point computations. While termination of loops over integers has been largely studied, there are only a few results about termination of loops dealing with floating point numbers. They make the analysis complex since, because of rounding errors, the expected behavior might be different from the real behavior of the program [Monniaux 2008]. Serebrenik and De Schreye [2002] prove termination of these programs by modeling the official standardized implementation of floating point numbers. They use level mappings over reals, but decreases must be bounded from below by some positive constant. In this article, we do not prove termination of loops over floating point numbers.

Formalizations of the semantics of Java. Our formalization of the semantics of Java bytecode is indebted to Klein and Nipkow [2006], where Java and Java bytecode are mathematically formalized and the compilation of Java into bytecode and its type-safeness are machine-proved. Our formalization of the state of the Java Virtual Machine (Definition 1) is similar to theirs, with the exception that we do not use a program counter nor keep the name of the current method and class inside the state. This information is not relevant for our abstraction into path-length and we avoid program counters by using blocks of code linked by arrows as concrete representation of the structure of the bytecode. Also our formalization of the heap and of the objects inside the heap is identical to theirs. Their mathematical formalization has been coded inside the Isabelle/HOL theorem prover and then used to prove the absence of overflows in a program [Wildmoser and Nipkow 2005] with the help of code annotations (invariants) which have been later computed automatically through interval analysis [Wildmoser et al. 2005]. Our formalization is denotational rather than operational since we use it to define an abstraction of a relational property of the semantics of the commands (the path-length), that is, an abstraction of the denotations. The same abstraction, based on an operational semantics, would be awkward. Another formalization of the semantics of the Java bytecode is presented in Bannwart and Müller [2005] but it is relatively different from ours in the definition of the heap and in the use of weakest preconditions rather than denotational semantics.

Abstract domains for the static analysis of Java. Our abstract domain for path-length (Section 6) abstracts a property of the heap, namely, the maximal length of a chain of pointers reachable from each variable in the program. From this point of view, it is related to a traditional *norm* used to prove termination of logic programs, which measures the *height* of a term, seen as a tree. The main difference is that, for its definition, we need precise information about the shape of the heap at runtime at each program point. Namely, we need information about sharing and cyclicity of data structures. Determining an overapproximation of the pairs of program variables that share at each program point is an extensively studied problem. There is a huge literature about *pointer* or *aliasing* analysis [Choi et al. 1993; Steensgaard 1996] and about *shape* analysis [Wilhelm et al. 2002; Distefano et al. 2006] of data structures. Many flavors of such analyses are fully qualified for computing possible sharing pairs of variables. More generally, separation logic [Reynolds 2000; Ishtiaq and O'Hearn 2001] is a framework which allows one to define analyses of properties of the heap and can express properties like sharing and cyclicity of data structures. It is known, however, that a static analysis for sharing can be much more abstract than aliasing or shape analysis, which justifies the development of abstract domains which track these properties explicitly, rather than as a side-effect [Pollet et al. 2001]. Namely, the abstract domain, defined and proved correct in Secci and Spoto [2005], is just made of sets of pairs of possibly sharing variables. This results in a static analysis which can be implemented in a completely context- and flow-sensitive way and still requires one or two orders of magnitude less time than, for instance, aliasing analysis [Payet and Spoto 2007]. It must be clear, however, that sharing is too abstract if possible aliasing is what is needed, but this is not the case in this article.

Tools for the static analysis of Java. Many tools have been devoted to the analysis or verification of Java or Java bytecode programs. Although such systems have not been used for termination analysis, we think that they could be instantiated for that purpose. They should be enriched with analyses computing information about the shape of the memory, such as our sharing and cyclicity analyses; hence some measure similar to our path-length information could be computed and termination proved by showing that, along loops and recursion, this measure is decreasing over a well-founded order. BANDERA [Corbett et al. 2000] takes a source Java program and extracts compact finite-state models of the program which can then be sourced to a model checker. It also performs some static analyses. It includes a program slicer for better efficiency and uses abstract interpretation for the finite representation of the states. JAVAPATHFINDER [Visser et al. 2003] uses model-checking to explore the states of a Java program and its scheduling sequences. As a consequence, it has been shown effective to prove properties of real-time Java [Lindstrom et al. 2005]. JMOPED [Suwimonteerabuth et al. 2007] is a test environment for a subset of Java. It uses model-checking to explore the set of states reachable from some input states taken from a testing set. It signals bugs or problems such as assertion violations, `null` pointer exceptions, and array bound violations. Testing is not in general complete, so it is hard to foresee an application of this tool to termination analysis, where termination must be proved for *all* input states. Moreover, only a subset of Java is considered, with strong limitations such as a ban of negative numbers. JMOPED has also been used for testing Java bytecode [Suwimonteerabuth et al. 2005], with strong limitations such as a bound on the heap size which prevents a `new` bytecode from occurring inside a loop.

KeY [Ahrendt et al. 2005] is a tool for the design, implementation, specification, and verification of object-oriented programs. It verifies properties expressed in the Object Constraint Language or in JML. It is a semiautomatic tool, based on theorem proving. Programs must first be annotated with the properties to prove and a theorem prover then attempts their proof with possible human interaction. Boogie [Barnett et al. 2005] is a program verifier for Spec# programs in the .NET framework. It has been recently applied to Java bytecode [Lehner and Müller 2007], by translating it into BoogiePL, the input language of Boogie. It includes a framework for abstract interpretation to build loop invariants that it uses to instrument the code. Invariants about the heap can be constructed through the abstract domain defined in Chang and Leino [2005]. Namely, this allows one to track which parts of the heap are preserved across updates and get information about *purity* of function arguments. Proofs are built through theorem-proving. The goal of this tool is the proof of *object invariants* [Leino and Wallenburg 2008], that is, data consistency properties about the objects of a program. These invariants might be violated within a small scope but must hold after each call from the external environment has completed. Object invariants are specified by the user and verified by the system. The use of *ownership* [Leino and Müller 2004; Müller 2007] allows one to model invariants which must hold of data structures as a whole rather than for single component objects. It is also possible to prove *class invariants*, which are related to static fields [Leino and Müller 2005]. A distinguishing feature of these works is the modularity of the verification, which we currently lack. These works based on theorem proving cannot be considered fully automatic since the user has to provide a specification of the property to prove and the theorem prover will likely require human intervention to reach the proof. Moreover, although it is possible, in principle, to prove termination with such techniques, we are not aware of any general technique for that purpose. In object-oriented programs the set of classes to analyze must be *extracted* from the starting class, containing the `main` method, by using some form of *application extraction*. This extraction is important in order to avoid the analysis of *all* classes, even those that are not relevant for the analysis. Our Julia tool uses a sophisticated algorithm based on Palsberg and Schwartzbach [1991], rephrased for the Java bytecode. We are not aware of other tools implementing similar, very precise application extraction techniques.

Previous publications of this material. The material presented in this article is partially based on our previous work. Secci and Spoto [2005] and Rossignoli and Spoto [2006] present the sharing and cyclicity analyses that we use in Section 4. The path-length abstract domain has been defined in Spoto et al. [2006]. The last three papers are presented for Java, while we rephrase their analyses here for Java bytecode and embed them into the semantic framework of Payet and Spoto [2007], where the operational and denotational semantics of Section 5 are presented and their equivalence is shown.

10. DISCUSSION

We have shown that our analyzer proves, automatically, termination of programs using nontrivial forms of loops and recursion (Section 2 and Figure 16).

However, as the larger analyses in Section 8 show, it cannot, of course, decide termination in all cases. Many terminating methods are not proved to terminate. We consider some of them here.

A first example are methods that work over graphs. Since graphs are typically cyclical, it is not possible for us to prove termination of such methods. Methods over graphs often terminate because visited nodes get *colored*. The set of colored nodes is typically held in a `Set`, as in the following method defined on the node of a graph.

```
void visit(Set<Node> coloured) {
  if (coloured.contains(this)) return;
  else coloured.add(this);

  ... visit this node and its successors, recursively ...
}
```

Here, `coloured` avoids repeated visits since a node cannot be colored twice. Termination of this (very frequent) programming pattern would follow from a proof that a node cannot be put twice in the set, that the set `coloured` does not shrink and that the set of nodes does not grow. Note that this proof cannot be obtained by simply using the size of the set as the path-length of `coloured`.

Another notable example are those methods whose termination depends on computations over real numbers, such as some approximation algorithms. In our implementation, the path-length of `float` and `double` variables is not computed, so that all such methods cannot be proved to terminate. The problem here is that numerical rounding must be taken into account for a faithful approximation of the values of real variables [Monniaux 2008]. Moreover, the set of real numbers is not well-founded even if a lower bound is considered. It might be possible here to use techniques which prove strict decrease by some positive constant [Serebrenik and De Schreye 2002].

The precision of our termination analysis is also limited by the fact that arithmetic bytecodes such as imul or idiv have no linear approximation that we can use for their path-length analysis. For the moment, we provide no path-length approximation for their result. This situation might be improved with some preliminary constant propagation, since in many cases those operations involve a variable and a constant, so that their path-length can be approximated by a linear constraint. A more general solution is to use nonlinear approximations of the path-length, such as in Bradley et al. [2005], Cousot [2005], and Babic et al. [2007]. This will increase the cost of the analysis, though.

The precision of the preliminary analyses is important for the precision of the termination analysis. For instance, our analyzer does not prove the termination of the method

```
public void expand(Sharing other) {
  Sharing cursor = this;
  while (cursor != null) {
    try {
      other.next = new Sharing(null);
```

```
        other = other.next;
        cursor = cursor.next;
      }
      catch (NullPointerException e) {
      }
    }
  }
```

when it is called with a nonnull argument other. This is because our preliminary null pointer analysis is not able to prove that other remains nonnull inside the while loop. In order to prove that result, we would need a more precise null pointer analysis and we should include the java.lang. hierarchy in the analysis, so that the analyzer can prove that the OutOfMemoryError which might be thrown by new Sharing(null) is not a subclass of NullPointerException.

In general, better information about the fields of the objects is needed in our analyses. Sharing, cyclicity, and path-length are by definition properties that involve some information about the fields. But this is not always true. For instance, integer fields of objects do not contribute to the definition of the path-length (Definition 24). As a consequence, we cannot prove termination of a loop decreasing an integer field which is bounded from below. We plan to study the applicability of the domain in Chang and Leino [2005] to our framework. It provides a way of approximating fields which is finer than ours.

It must be stressed also that our analysis is meant for *sequential* Java bytecode, not using multithreading. However, we share this limitation with most other works on termination analysis. If one allows any kind of data structures, possibly shared between threads, and an unbounded number of dynamically created threads, very little can be said about the termination of the programs. Recent research can prove only special cases, when, for instance, the number of threads is fixed in advance [Cook et al. 2007].

A final limitation of our analysis is a consequence of the use of native methods and reflection (the ability of Java programs to access, create, and modify objects, classes, and the program itself through some methods of the standard Java libraries, mostly native). We have manually provided approximations for a few hundreds of such methods, for all the static analyses that we perform. For other native methods, JULIA signals a warning to the user, meaning that the result of the analyses might not be reliable. Most native methods implementing reflection have not been manually specified. Since reflection can modify the same program under analysis, we cannot see a simple way of analyzing programs dealing with reflection.

Let us make a final consideration about the cost of our analysis. Figure 17 reports analysis of programs of up to 1201 methods, since the cost of the analysis is still relatively high. This problem is not related to preprocessing and to the preliminary analyses, which are able to scale to programs of up to 10,000 methods, but it is related to the cost of the path-length analysis and of the subsequent termination proof. A possible solution to this problem is to use less precise but more efficient abstractions or algorithms. Octagons [Miné 2006] or size-change termination in polynomial time [Ben-Amram and Lee 2007] are possible candidates. Moreover, the standard Java library classes could be analyzed once and for all, so that a path-length approximation for them can be plugged into all programs that use these libraries, instead of reanalyzing the libraries each time. Besides, library methods that are known to terminate, for instance, by using semiautomatic techniques such as theorem proving, need not be proved to terminate by our analyzer. This would increase both its efficiency and its precision.

In conclusion, our analyzer shows that a completely automatic termination proof for Java bytecode is possible. Future research will improve its precision and reduce the cost of the analysis.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their helpful comments on this work. They thank P. Hill for her help with English. They also thank R. Bagnara for his support with the installation and use of the Parma Polyhedra Library and all the developers of the PPL for providing them with the Java interface to that library.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers, Principles Techniques and Tools*. Addison Wesley.

Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., and Schmitt, P. H. 2005. The KeY Tool. *Softw. Syst. Model. 4*, 1, 32–54.

Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., and Zanardini, D. 2007a. Termination analysis of Java bytecode. In *Proceedings of the 9th International Workshop on Termination (WST'07)*. A. Serebrenik and D. Hofbauer, Eds.

Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., and Zanardini, D. 2008. Termination analysis of Java bytecode. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*. G. Barthe and F. S. de Boer, Eds. Lecture Notes in Computer Science, vol. 5051. Springer, 2–18.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2007b. Cost analysis of java bytecode. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*. R. De Nicola, Ed. Lecture Notes in Computer Science, vol. 4421. Springer, 157–172.

Avery, J. 2006. Size-Change termination and bound analysis. In *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS'06)*. M. Hagiya and P. Wadler, Eds. Lecture Notes in Computer Science, vol. 3945. Springer, 192–207.

Babic, D., Hu, A. J., Rakamaric, Z., and Cook, B. 2007. Proving termination by divergence. In *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM'07)*. IEEE Computer Society, 93–102.

Bacon, D. F. and Sweeney, P. F. 1996. Fast static analysis of c++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. *ACM SIGPLAN Not. 31*, 10, ACM, 324–341.

Bagnara, R., Hill, P. M., Ricci, E., and Zaffanella, E. 2005. Precise widening operators for convex polyhedra. *Sci. Comput. Program. 58*, 1–2, 28–56.

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program. 72*, 1–2, 3–21.

Bannwart, F. and Müller, P. 2005. A program logic for bytecode. *Electr. Not. Theor. Comput. Sci. 141*, 1, 255–273.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symposium* *on Formal Methods for Components and Objects (FMCO'05)*. F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds. Lecture Notes in Computer Science, vol. 4111. Springer, 364–387.

Ben-Amram, A. M. and Lee, C. S. 2007. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst. 29*, 1.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. 2007a. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer, 178–192.

Berdine, J., Chawdhary, A., Cook, B., Distefano, D., and O'Hearn, P. W. 2007b. Variance analyses from invariance analyses. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. M. Hofmann and M. Felleisen, Eds. 211–224.

Berdine, J., Cook, B., Distefano, D., and O'Hearn, P. W. 2006. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 386–400.

Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., and Vojnar, T. 2006. Programs with lists are counter automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 517–531.

Bradley, A., Manna, Z., and Sipma, H. 2005. Termination of polynomial programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. R. Cousot, Ed. Lecture Notes in Computer Science, vol. 3385. Springer, 113–129.

Brodsky, A. and Sagiv, Y. 1989. Inference of monotonicity constraints in datalog programs. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 190–199.

Brotherston, J., Bornat, R., and Calcagno, C. 2008. Cyclic proofs of program termination in separation logic. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. G. C. Necula and P. Wadler, Eds. ACM, 101–112.

Bryant, R. E. 1986. Graph-Based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35*, 8, 677–691.

Chang, B.-Y. E. and Leino, K. R. M. 2005. Abstract interpretation with alien expressions and heap structures. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. R. Cousot, Ed. Lecture Notes in Computer Science, vol. 3385. Springer, 147–163.

Choi, J. D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Symposium on Principles of Programming Languages (POPL'93)*. ACM, 232–245.

Codish, M. 2007. Proving termination with (boolean) satisfaction. In *Proceedings of the 17th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'07)*. A. King, Ed. Lecture Notes in Computer Science, vol. 4915. 1–7.

Codish, M., Lagoon, V., and Stuckey, P. J. 2005. Testing for termination with monotonicity constraints. In *Proceedings of the 21st International Conference on Logic Programming (ICLP'05)*. M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. 326–340.

Codish, M. and Taboch, C. 1999. A semantics basis for termination analysis of logic programs. *J. Logic Program. 41*, 1, 103–123.

Cook, B., Podelski, A., and Rybalchenko, A. 2005. Abstraction refinement for termination. In *Proceedings of the 12th Static Analysis Symposium (SAS'05)*. C. Hankin and I. Siveroni, Eds. Lecture Notes in Computer Science, vol. 3672. Springer, 87–101.

Cook, B., Podelski, A., and Rybalchenko, A. 2006a. Termination proofs for systems code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. M. I. Schwartzbach and T. Ball, Eds. ACM, 415–426.

Cook, B., Podelski, A., and Rybalchenko, A. 2006b. Terminator: beyond safety. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 415–418.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2007. Proving thread termination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. J. Ferrante and K. S. McKinley, Eds. ACM, 320–330.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*. ACM, 439–448.

COUSOT, P. 2005. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. R. Cousot, Ed. Lecture Notes in Computer Science, vol. 3385. Springer, 1–24.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*. 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages (POPL'79)*. ACM, 269–282.

COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages (POPL'78)*. ACM, 84–96.

DE SCHREYE, D. AND DECORTE, S. 1994. Termination of logic programs: The never-ending story. *Journal of Logic Programming 19/20*, 199–260.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. W. G. Olthoff, Ed. Lecture Notes in Computer Science, vol. 952. Springer, 77–101.

DERSHOWITZ, N., LINDENSTRAUSS, N., SAGIV, Y., AND SEREBRENIK, A. 2001. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Engin. Comm. Comput. 12*, 1-2, 117–156.

DISTEFANO, D., O'HEARN, P. W., AND YANG, H. 2006. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*. H. Hermanns and J. Palsberg, Eds. Lecture Notes in Computer Science, vol. 3920. Springer, 287–302.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. *Proceedings of Symposia in Applied Mathematics*, vol. 19. American Mathematical Society, Providence, Rhode Island, 19–32.

GABBRIELLI, M. AND GIACOBAZZI, R. 1994. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the ACM Symposium on Applied Computing (SAC'94)*. ACM, 394–399.

GENAIM, S. AND CODISH, M. 2005. Inferring termination conditions for logic programs using backwards analysis. *Theory Prac. Logic Program. 5*, 1-2, 75–91.

GENAIM, S. AND SPOTO, F. 2008. Constancy analysis. In *Proceedings of the 10th Workshop on Formal Techniques for Java-like Programs (FTfJP'08)*, M. Huisman, Ed.

GIESL, J., SCHNEIDER-KAMP, P., AND THIEMANN, R. 2006. Automatic termination proofs in the dependency pair framework. In *Proceddings of the 3th International Joint Conference on Automated Reasoning (IJCAR'06)*. U. Furbach and N. Shankar, Eds. Lecture Notes in Computer Science, vol. 4130. Springer, 281–286.

GOTSMAN, A., BERDINE, J., AND COOK, B. 2006. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of the 13th International Static Analysis Symposium (SAS'06)*. K. Yi, Ed. Lecture Notes in Computer Science, vol. 4134. Springer, 240–260.

ISHTIAQ, S. S. AND O'HEARN, P. W. 2001. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Symposium on Principles of Programming Languages (POPL'01)*. ACM, 14–26.

JAFFAR, J. AND MAHER, M. J. 1994. Constraint logic programming: A survey. *J. Logic Program. 19*, 20, 503–581.

KLEIN, G. AND NIPKOW, T. 2006. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst. 28*, 4, 619–695.

Leavens, G. T., Leino, K. R. M., and Müller, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects Comput. 19*, 2, 159–189.

Lee, C. S., Jones, N. D., and Ben-Amram, A. M. 2001. The size-change principle for program termination. In *Proceedings of the 28th Symposium on Principles of Programming Languages (POPL'01)*. ACM, 81–92.

Lehner, H. and Müller, P. 2007. Formal translation of bytecode into boogiePL. *Electr. Not. Theor. Comput. Sci. 190*, 1, 35–50.

Leino, K. R. M. and Müller, P. 2004. Object invariants in dynamic contexts. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*. M. Odersky, Ed. Lecture Notes in Computer Science, vol. 3086. Springer, 491–516.

Leino, K. R. M. and Müller, P. 2005. Modular verification of static class invariants. In *Proceedings of the International Symposium of Formal Methods Europe (FM'05)*. J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds. Lecture Notes in Computer Science, vol. 3582. Springer, 26–42.

Leino, K. R. M. and Wallenburg, A. 2008. Class-local object invariants. In *Proceedings of the 1st India Software Engineering Conference (ISEC'08)*. G. Schroff, P. Jalote, and S. K. Rajamani Eds., ACM, 57–66.

Lindenstrauss, N., Sagiv, Y., and Serebrenik, A. 1997. TermiLog: A system for checking termination of queries to logic programs. In *Proceedings of the 9th International Conference on Computer Aided Verfication (CAV'97)*. O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer, 444–447.

Lindholm, T. and Yellin, F. 1999. *The Java$^{\text{TM}}$ Virtual Machine Specification*, 2nd ed. Addison-Wesley.

Lindstrom, G., Mehlitz, P. C., and Visser, W. 2005. Model checking real time java using java Pathfinder. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA,05)*. D. Peled and Y.-K. Tsay, Eds. Lecture Notes in Computer Science, vol. 3707. Springer, 444–456.

Loginov, A., Reps, T. W., and Sagiv, M. 2006. Refinement-based verification for possibly-cyclic lists. In *Proceedings of Theory and Practice of Program Analysis and Compilation, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, T. W. Reps, M. Sagiv, and J. Bauer, Eds. Lecture Notes in Computer Science, vol. 4444. Springer, 247–272.

Logozzo, F. and Fähndrich, M. 2008. On the relative completeness of bytecode analysis versus source code analysis. In *Proceedings of the 17th International Conference on Compiler Construction (CC'08)*. L. Hendren, Ed. Lecture Notes in Computer Science. Springer, 197–212.

Manolios, P. and Vroon, D. 2006a. Integrating static analysis and general-purpose theorem proving for termination analysis. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 873–876.

Manolios, P. and Vroon, D. 2006b. Termination analysis with calling context graphs. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 401–414.

Manson, J. and Pugh, W. 2001. Core semantics of multithreaded java. In *Proceedings of the ACM Java Grande Conference*. ACM, 29–38.

Manson, J., Pugh, W., and Adve, S. V. 2005. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. J. Palsberg and M. Abadi, Eds. ACM, 378–391.

Mesnard, F. 1996. Inferring left-terminating classes of queries for constraint logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, M. Maher, Ed. The MIT Press, 7–21.

Mesnard, F. and Bagnara, R. 2005. cTI: A constraint-based termination inference tool for ISO-Prolog. *Theory Pract. Logic Program. 5*, 1-2, 243–257.

Mesnard, F. and Serebrenik, A. 2008. Recurrence with affine level mappings is p-time decidable for CLP(R). *Theory Pract. Logic Program. 8*, 1, 111–119.

Miné, A. 2006. The octagon abstract domain. *Higher-Order Symb. Comput. 19*, 1, 31–100.

Monniaux, D. 2008. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst. 30*, 3.

MÜLLER, P. 2007. Reasoning about object structures using ownership. In *Proceedings of the Workshop on Verified Software: Theories, Tools, Experiments (VSTTE'07)*. B. Meyer and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 4171. Springer.

NYUGEN, M. T. AND DE SCHREYE, D. 2005. Polynomial interpretations as a basis for termination analysis of logic programs. In *Proceedings of the 21st International Conference on Logic Programming (ICLP'05).* M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 311–326.

OHLEBUSCH, E., CLAVES, C., AND MARCHÉ, C. 2000. TALP: A tool for the termination analysis of logic programs. *Proceedings of the 11th International Conference on Rewriting Techniques and Applications (RTA'00)*. L. Bachmair, Ed. Lecture Notes in Computer Science, vol. 1833. Springer, 270–273.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-Oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*. A. Paepcke, Ed. ACM SIGPLAN Not., *26*, 11 ACM, 146–161.

PAYET, E. AND SPOTO, F. 2007. Magic-sets transformation for the analysis of java bytecode. In *Proceedings of the 14th International Static Analysis Symposium (SAS'07)*. H. R. Nielson and G. Filé, Eds. Lecture Notes in Computer Science, vol. 4634. Springer, 452–467.

PIPPENGER, N. 1997. Pure versus impure lisp. *ACM Trans. Program. Lang. Syst. 19*, 2, 223–238.

PLÜMER, L. 1990. *Termination proofs for logic programs*. Lecture Notes in Computer Science, vol. 446. Springer.

PODELSKI, A. AND RYBALCHENKO, A. 2004a. A complete method for synthesis of linear ranking functions. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*. B. Steffen and G. Levi, Eds. Lecture Notes in Computer Science, vol. 2937. Springer, 239–251.

PODELSKI, A. AND RYBALCHENKO, A. 2004b. Transition invariants. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS'04)*. H. Ganzinger, Ed. IEEE, 32–41.

PODELSKI, A. AND RYBALCHENKO, A. 2007. Transition predicate abstraction and fair termination. *ACM Trans. Program. Lang. Syst. 29*, 3.

POLLET, I., LE CHARLIER, B., AND CORTESI, A. 2001. Distinctness and sharing domains for static analysis of java programs. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*. Lecture Notes in Computer Science, vol. 2072. 77–98.

REYNOLDS, J. C. 2000. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of Millennial Perspectives in Computer Science, Symposium in Honour of Sir Tony Hoare*, J. Davies, B. Roscoe, and J. Woodcock, Eds. 303–321.

ROSSIGNOLI, S. AND SPOTO, F. 2006. Detecting non-cyclicity by abstract compilation into boolean functions. In *Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06)*. E. A. Emerson and K. S. Namjoshi, Eds. Lecture Notes in Computer Science, vol. 3855. Springer, 95–110.

SALCIANU, A. AND RINARD, M. C. 2005. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. R. Cousot. Ed. Lecture Notes in Computer Science, vol. 3385. Springer, 199–215.

SCHNEIDER-KAMP, P., GIESL, J., SEREBRENIK, A., AND THIEMANN, R. 2006. Automated termination analysis for logic programs by term rewriting. In *Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'06)*. G. Puebla. Ed. Lecture Notes in Computer Science, vol. 4407. Springer, 177–193.

SECCI, S. AND SPOTO, F. 2005. Pair-sharing analysis of object-oriented programs. In *Proceedings of Static Analysis Symposium (SAS'05)*. C. Hankin and I. Siveroni, Eds. Lecture Notes in Computer Science, vol. 3672. 320–335.

SEREBRENIK, A. AND DE SCHREYE, D. 2002. On termination of logic programs with floating point computations. In *Proceedings of the 9th International Symposium on Static Analysis (SAS'02)*. M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer, 151–164.

SPOTO, F. 2008a. The JULIA static analyser. `http://profs.sci.univr.it/~spoto/julia`.

SPOTO, F. 2008b. Nullness analysis in boolean form. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08)*. A. Cerone and S. Goner Eds. IEEE Computer Society, 21–30.

SPOTO, F., HILL, P. M., AND PAYET, E. 2006. Path-Length analysis for object-oriented programs. In *International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*. http://profs.sci.univr.it/~spoto/papers.html.

SPOTO, F. AND JENSEN, T. 2003. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Programm. Lang. Syst. 25*, 5, 578–630.

SPOTO, F., MESNARD, F., AND PAYET, E. 2008. Julia + BinTerm: An automatic termination prover for Java bytecode. http://spy.sci.univr.it/JuliaWeb.

SPOTO, F., LU, L., AND MESNARD, F. 2009. Using *CLP* simplifications to improve Java bytecode termination analysis. *Electr. Notes Theor. Comput. Sci. 253*, 5, 129–144.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages (POPL'96)*. 32–41.

STOER, J. AND WITZGALL, C. 1970. *Convexity and Optimization in Finite Dimensions I*. Springer.

SUWIMONTEERABUTH, D., BERGER, F., SCHWOON, S., AND ESPARZA, J. 2007. jMoped: A test environment for java programs. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer, 164–167.

SUWIMONTEERABUTH, D., SCHWOON, S., AND ESPARZA, J. 2005. jMoped: A java bytecode checker based on moped. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. N. Halbwachs and L. D. Zuck, Eds. Lecture Notes in Computer Science, vol. 3440. Springer, 541–545.

TABOCH, C., GENAIM, S., AND CODISH, M. TerminWeb: Semantic based termination analyser for logic programs. http://www.cs.bgu.ac.il/~mcodish/TerminWeb.

TARSKI, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math 5*, 285–309.

TURING, A. 1936. On computable numbers, with an application to the entscheidungsproblem. *London Math. Soc. 42*, 2, 230–265.

ULLMAN, J. D. AND GELDER, A. V. 1988. Efficient tests for top-down termination of logical rules. *J. ACM 35*, 2, 345–373.

VISSER, W., HAVELUND, K., BRAT, G. P., PARK, S., AND LERDA, F. 2003. Model checking programs. *Autom. Softw. Engin. 10*, 2, 203–232.

WILDMOSER, M., CHAIEB, A., AND NIPKOW, T. 2005. Bytecode analysis for proof carrying code. *Electr. Not. Theore. Comput. Sci. 141*, 1, 19–34.

WILDMOSER, M. AND NIPKOW, T. 2005. Asserting bytecode safety. In *Proceedings of the 14th European Symposium on Programming (ESOP'05)*. S. Sagiv, Ed. Lecture Notes in Computer Science, vol. 3444. Springer, 326–341.

WILHELM, R., REPS, T. W., AND SAGIV, S. 2002. Shape analysis and applications. In *The Compiler Design Handbook*, Y. N. Srikant and P. Shankar, Eds. 175–218.

Received September 2007; revised May 2009; accepted July 2009