Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 131 (2005) 85–98
www.elsevier.com/locate/entcs

Towards a Complete Static Analyser for Java: an Abstract Interpretation Framework and its Implementation

Isabelle Pollet[1]

University of Namur

Baudouin Le Charlier[2]

Université Catholique de Louvain

Abstract

We present an abstract interpretation framework for a subset of Java (without concurrency). The framework uses a structural abstract domain whose concretization function is parameterized on a relation between abstract and concrete locations. When structurally incomptatible objects may be referred to by the same variable at a given program point, structural information is discarded and replaced by an approximated information about the objects (our presentation concentrates on type information). Plain structural information allows precise intra-procedural analysis but is quickly lost when returning from a method call. To overcome this limitation, relational structural information is introduced, which enables a precise inter-procedural analysis without resorting to inlining.
The paper contains an overview of the work. We describe parts of the standard and abstract semantics; then, we briefly explain the fixpoint algorithms used by our implementation; lastly, we provide experimental results for small programs.

Keywords: Abstract Interpretation, Java, Type Analysis, Pointer Analysis, Program Verification, Program Specialization.

[1] Email:ipo@info.fundp.ac.be
[2] Email:blc@info.ucl.ac.be

doi:10.1016/j.entcs.2005.01.025

Introduction

There is a broad range of applications for the static analysis of Java. However, a major issue is the correctness of the analysis itself, especially when it is used in optimizing compilers. But, designing an analysis and proving its correctness is often tedious and error-prone. It is therefore reasonable attempting to design a 'generic framework' easily adaptable to perform various kinds of analyses in order to minimize the correctness proof effort. Our work is a contribution to such a 'generic framework': We define and implement a Java code analyser based on the abstract interpretation methodology [8,9,10].

We limit our analysis to a (partly arbitrary) subset of the language. This subset is, on the one hand, representative enough of the main Java features and, on the other hand, sufficiently small to be completely dealt with in a first approach. Concurrency is the main Java feature that we eliminate. This aspect is very important but rather 'orthogonal' to the object-oriented aspects. We also assume the availability of the complete source code, ignoring, at the moment, the problem of dynamic class loading (see e.g. [17]).

We apply the abstract interpretation methodology as follows: We define a straightforward standard semantics, abstract domains and an abstract semantics on those domains, which correctly approximates the standard semantics. We finally use a post-fixpoint algorithm to compute a (relevant part of) the abstract semantics.

Our abstract domains contain structural information and closely resemble to the standard domain (consisting of an environment and a store). Abstract locations may be annotated with various kind of information, making the framework generic. Structure sharing at the abstract level can be interpreted in several different ways, at the standard level, giving rise to three variants of the abstract domains [3].

The result of an analysis is a table of abstract input/output states describing method and constructor calls that can potentially arise during an actual standard execution. Such a table is similar to (and allows one to easily build) a (precise) call graph [12,26] for the whole program.

This paper presents an overview [4] of the work and is composed of five main sections. Section 1 provides a brief overview of the standard semantics. Section 2 describes the abstract domains. Section 3 sketches the abstract semantics definition. Section 4 details the results of the analysis for small programs. Section 5 discusses the related work.

[3] A preliminary presentation of the abstract domains appeared in [20].
[4] All technical definitions can be found in [19].

```
abstract class T{ abstract void inc();}

class IntT extends T {
  int cont;
  IntT(int v) {cont=v;}
  void inc() { cont = cont + 1; }
 }

class CoupleT extends T {
    T v1;T v2;

    CoupleT(T p1, T p2) {
       v1 = p1; v2 = p2;
    }

    void inc(){
        v1.inc(); v2.inc();
    }
}
```

```
class CoupleInt extends CoupleT {
    CoupleInt(int i, int j) {
       super(new IntT(i), new IntT(j));
    }
}

class TList {
  T val; TList next;

  TList() {}
  TList(T v, TList tail) {
      val=v; next=tail; }

  void permut()
  {
      TList tmp = next;
      next = tmp.next;
      tmp.next = next.next;
      next.next = tmp;
  }
```

```
void add(T v) {
     TList aux = new TList(v,next);
     next = aux;
  }
}

class Main {
    void main()
    {
        TList l = new TList();
        l.add(new IntT(1));
        l.add(new IntT(3));
        TList one = l.next;
        TList two = l.next.next;
        l.swap();
        l.next.val = new CoupleInt(1,2);
        one.val.inc(); two.val.inc();
   }
}
```

Fig. 1. The `Swap` program: the running example of the paper

1 Target language and standard (fixpoint) semantics

We focus on a restricted subset of Java that contains the main object oriented features of the language such as inheritance and virtual method calls. To be manageable in a first approach, our sublanguage includes a limited number of basic types and control statements. We believe that most of those limitations are only syntactic. However, there are some major restrictions: we do not treat concurrency and currently perform a whole program analysis. We also do not address native methods nor reflection issues. (Figure 1 depicts a small program of the target subset.)

Quite usually, we do not directly analyze the source code and work with an intermediate representation, which is a 'three-address' and 'stackless' representation of the Java bytecode. In this representation, which we call *LAS*, expressions are simplified (they do not contain method nor constructor calls) and typed and all default rules are explicitly translated. The Java-*LAS* compiler introduces the necessary internal variables and verifies the type-checking rules.

The semantics of a program maps each *block identifier*, i.e., each method or constructor signature, to the corresponding store transformation. More formally, the semantics of a program *prog* is the least fixpoint of the transformation $\tau[\![prog]\!]$ of the set of *block environments*

$$\mathbb{E}nvB = \mathit{BlockId} \to (\mathbb{D} \times \mathbb{I}o \nrightarrow \mathbb{D} \times \mathbb{I}o + \mathbb{D} \times \mathbb{I}o \times \mathbb{V}al),$$

induced by the declarations of the methods and constructors of *prog*. The set $\mathbb{I}o$ is introduced to deal with a simple input-output device and the set of *local stores*, $\mathbb{D}$, which we call *standard domain*, is defined by the equations

$$\mathbb{B}ase = \mathbb{S}calar + \{\mathbf{null}, \mathbf{nonInit}\},\ \mathbb{V}al = \mathbb{B}ase + \mathbb{L}oc,$$
$$\mathbb{I}nst = \mathit{Nclass} \times (\mathbb{F}ield \nrightarrow \mathbb{V}al),\ \mathbb{F}ield = \mathit{Nclass} \times \mathit{Nfield},$$
$$\mathbb{E}nv = \mathit{LocalId} \nrightarrow \mathbb{V}al,\ \mathbb{S}tore = \mathbb{L}oc \nrightarrow \mathbb{I}nst, \mathbb{D} = \mathbb{E}nv \times \mathbb{S}tore.$$

Roughly, a local store is composed of an *environment* and a *store*. The environment maps each *local identifier* (variable or parameter) to its value and the store each *location* (address instance) to an instance. Figure 2 provides an example of local store.

$d_1 = (e, s),\ e = \{(\mathbf{this}, 0), (\mathtt{tmp}, 1)\},$
$s = \{(0, (\mathtt{TList}, f_0)), (1, (\mathtt{TList}, f_1)), (2, (\mathtt{TList}, f_2)), (3, (\mathtt{IntT}, f_3)), (4, (\mathtt{Int}, f_4))\}$
$f_0 = \{((\mathtt{TList}, \mathtt{next}), 2), ((\mathtt{TList}, \mathtt{val}), \mathbf{null})\},$
$f_1 = \{((\mathtt{TList}, \mathtt{next}), \mathbf{null}), ((\mathtt{TList}, \mathtt{val}), 3)\}$
$f_2 = \{((\mathtt{TList}, \mathtt{next}), 1), ((\mathtt{TList}, \mathtt{val}), 4)\}$
$f_3 = \{((\mathtt{IntT}, \mathtt{cont}), \mathbf{int}\ 1)\},\ f_4 = \{((\mathtt{IntT}, \mathtt{cont}), \mathbf{int}\ 3)\}$

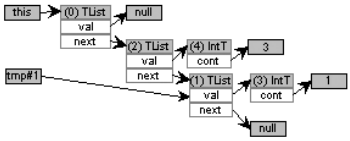


Fig. 2. An element of $\mathbb{D}$, corresponding to the method `TList.swap()`, and its graphical representation

2 Abstract domains

We use several abstract domains, i.e several abstractions for the set $\mathbb{D}$. All those domains are *structural* ones: they keep a partial view of the heap structure. When a variable or field may concretely denote values with different types, an *abstract type* is introduced in the abstract graph. Those abstract types belong to a *parameter abstract domain*, which may express various kinds of information (although we currently mainly focus on type information).

Those considerations lead to an abstract domain, $\mathbb{D}^\sharp$, that looks very similar to the standard one. Its structure is definedby the equations

$$\mathbb{V}al^\sharp = \mathbb{A}^\sharp \setminus \{\bot\} + \mathbb{L}oc^\sharp, \mathbb{I}nst^\sharp = \mathit{Nclass} \times (\mathbb{F}ield \rightharpoonup \mathbb{V}al^\sharp),$$
$$\mathbb{E}nv^\sharp = \mathit{LocalId} \rightharpoonup \mathbb{V}al^\sharp,\ \mathbb{S}tore^\sharp = \mathbb{L}oc^\sharp \rightharpoonup \mathbb{I}nst^\sharp, \mathbb{D}^\sharp = \mathbb{E}nv^\sharp \times \mathbb{S}tore^\sharp + \{\bot\},$$

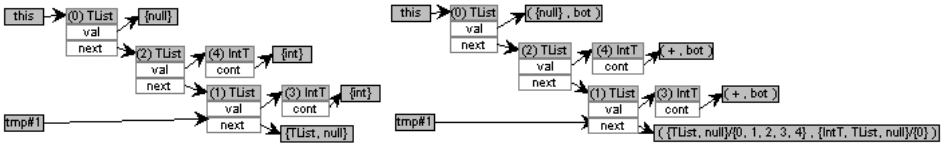
where the set $\mathbb{A}^\sharp$ denotes the parameter abstract domain.

The semantics of $\mathbb{D}^\sharp$ is specified by the means of a concretization function, which relies on the existence of a *structural morphism* between the standard local store and the abstract local store. We actually propose three variants of this concretization function based on different requirements for the underlying morphism. Those variants lead to three abstract domains that differ by the information they can express about the sharing of instances.

(i) In the *Exact Domain*, the structural information is exact (as long as the structure is kept). This domain is the most precise to abstract a single element of the standard domain but it loses most of the structural information when abstracting elements with different instance sharing.

(ii) In the *Distinctness Domain*, we can express distinction between instances but not sure sharing.

(iii) The *Sharing Domain* is in some sense the 'dual' of the Distinctness Domain. In this last domain, we are able to express sure sharing of instances but not distinctness of instances.



Abstract local store $d_1^\sharp$ Abstract local store $d_2^\sharp$

Fig. 3. Examples of abstract local states

Figure 3 depicts two elements of variants of the abstract domain that both abstract the local store of Figure 2. In the case of $d_1^\sharp$, the parameter domain only provides type information: `tmp.next` is null or denotes any instance of type `TList` and we know nothing about the value of `tmp.next.next`. In the Exact Domain, `this` denotes a structure starting with at least three distinct cells and the variable `tmp` points out the third cell of this structure. In the Distinctness Domain, `this` also denotes a 'list' with at least three distinct cells but we do not know if `tmp` points out the third cell of this list or a cell distinct from the first cells. In the Sharing Domain, cells 2 and 1 can stand for the same concrete cells but we are sure that `tmp` and `next.next` share. The parameter domain used by $d_2^\sharp$ is more sophisticated. It notably contains a *reachability component* which supplies information about all the reachable values: `tmp.next` is null or denotes an instance of `TList` that is distinct from the other instances of the graph, all instances that are reachable from the field `tmp.next` are either of type `TList` or of type `IntT`, moreover, they are distinct from the instances represented by the first abstract instance.

3 Abstract semantics

The abstract semantics of a program *prog* is a post-fixpoint of a transformation of the set of *abstract block environments*

$$\mathbb{E}nvB^\sharp = BlockId \to (\mathbb{D}^\sharp \nrightarrow \mathbb{D}^\sharp + \mathbb{D}^\sharp \times \mathbb{V}al^\sharp).$$

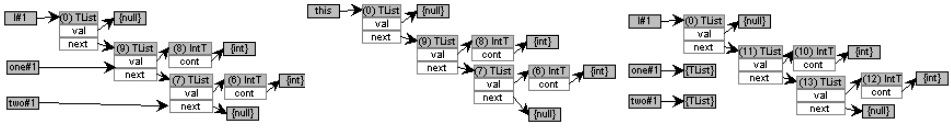
The definition of this transformation relies on an *abstract local transition* function

$$[\![.]\!]^\sharp : Statm \to \mathbb{E}nvB^\sharp \to Lab \times \mathbb{D}^\sharp \nrightarrow \wp(Lab \times \mathbb{D}^\sharp),$$

which expresses the semantics of each statement within a supposed abstract block environment. For intra-procedural analysis, the definition of this function roughly corresponds to the design of abstract operations on $\mathbb{D}^\sharp$.

For inter-procedural analysis, a straight use of the abstract domains leads to the loss of most structural information when returning from a call because of the lack of relational information between abstract states. Let us illustrate this problem with a simple example: We want to execute the call `l.swap()`, within the method `Main.main()` of Figure 1, in the abstract local store $d_1^\sharp$ of Figure 4. Besides, the abstract block environment maps the corresponding entry $d_0^\sharp$ to the local store $d_0^\sharp$ itself. Actually, the provided information is only a shape information and we do not know, for instance, if location 9 of $d_0^\sharp$ denotes the same location at the beginning and at the end of the call. It can actually represent location 7 or 9 of $d_1^\sharp$ or any new location created during the call. So, we get after the call situation $d_2^\sharp$ in which the structures of *one* and *two* are completely lost.

To overcome this problem, we introduce relational information between the starting and ending states of a call by duplicating the parameters and fields. When starting a call, we make a copy of all fields values. Those copies cannot concretely be modified during the call. Thus, when returning from the call, we can perform a kind of unification between the values before the call and the values of the copies after the call. Figure 5 illustrates this idea by revisiting the example of Figure 4.



$d_1^\sharp$: before the call `l.swap()` $d_0^\sharp$: input/output in the block environment $d_2^\sharp$: after the call

Fig. 4. Return of a call without relational information (in the Exact Domain)

4 Abstract semantics computation

To compute the abstract semantics, we use, at the inter-procedural level, the generic post-fixpoint algorithm proposed in [14] combined, at the intra-procedural level, with classical monovariant or polyvariant algorithms. The inter-procedural algorithm builds a table of abstract input/output states for all methods and constructors that are potentially executed. This table is a partial description of the abstract semantics, driven by the analysis of an initial call.

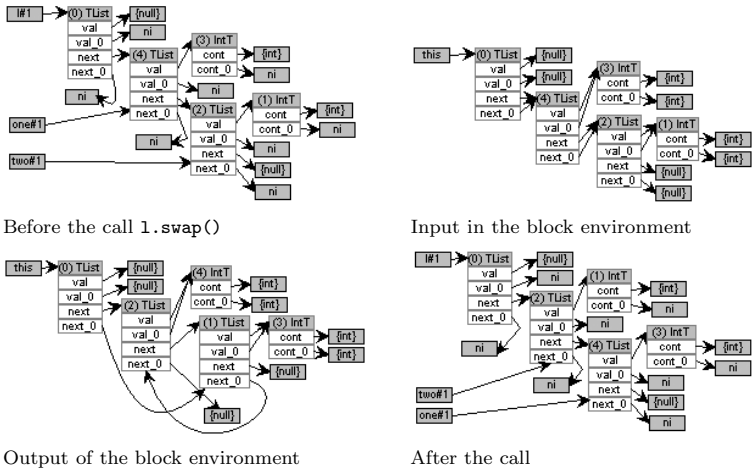


Fig. 5. Return of a call with relational information (in the Exact Domain)

5 Results

Our implementation provides a graphical interface to navigate through the partial table produced by the post-fixpoint algorithm. It also allows us to visualize the abstract states at each program point. We present parts of these graphical results[5] for two small programs: the first one in a cast verification optic and the second one as a case of program specialization.

The class `Stack` of Figure 6 implements generic stacks. The methods of the class `Job` simulate, in a simplified context, a classical way of manipulating generic data structures in Java. The method `Job.treatInt()` applies the method `Int.inc()` to all the objects stored on a stack. This method assuming that all stored objects are instances of `Int`. The method `Job.memorizeInt()` builds a stack satisfying the hypothesis, whereas the method `Job.memorize()` builds a stack that may also contain instances of `CoupleInt`. We address the problem of validate (or invalidate) the cast within the method `Job.treatInt()` in different contexts of execution.

We first consider the analysis of the method `Main.main1()`. Snapshot 1 of Figure 7 shows the only entry in the table for the method `Job.treatInt()`. The input situation depicts the stack built by the method `Job.memorizeInt()` and the output describes, as expected, an empty stack. Snapshot 2 depicts the abstract store immediately before the cast check within the statement `v = (Int) var.pop()`. We can derive from the value mapped to the internal

[5] The results supplied in this section were obtained with variants of the analyser that exploit relational information and use the Exact Domain. The parameter domain includes a reachability component.

```
abstract class Object { }

class Int extends Object{
  int cont;
  Int(int v) { cont=v; }
  void inc() { cont=cont+1; }
}

class Couple extends Object{
  Object p1; Object p2;

  Couple(Object v1, Object v2){
    p1=v1; p2=v2;
  }
}

class CoupleInt extends Couple{
  CoupleInt(int i, int j){
    super(new Int(i),new Int(j));
  }
}

class Stack {
  Object val; Stack next;
  void push(Object v){
    Stack aux = new Stack();
    aux.val = v;
    aux.next = next;
    next = aux;
  }

  Object pop(){
    Object res = next.val;
    next = next.next;
    return res;
  }
}

class Job {
  Stack var;
  Job(Stack s) { var=s; }

  void treatInt(){
    if (!(var.next == null)) {
      Int v = (Int) var.pop();
      v.inc();
      treatInt();
    }
  }
  void memorizeInt(){
    int n = IO.read();
    if (n == 0){
      n = IO.read();
      var.push(new Int(n));
      memorizeInt();
    }
  }

  void memorize(){
    int n = IO.read();
    if (n == 0){
      n = IO.read();
      var.push(new Int(n));
      memorize();
    }
    else {
      n = IO.read();
      var.push(new CoupleInt(n,n));
      memorize();
    }
  }
}
class Main {
  void main1(){
     Stack s = new Stack();
     Job j = new Job(s);
     j.memorizeInt();
     j.treatInt();
   }

   void main3(){
      Stack s = new Stack();
      Job j = new Job(s);
      j.memorize();
      j.treatInt();
    }
}
```

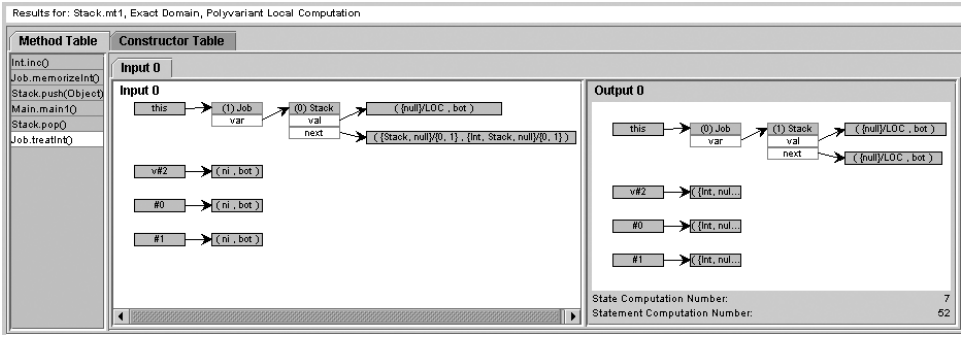
Fig. 6. The `Stack` program

variable #0, which denotes the value returned by the call `var.pop()`, that the cast is certainly valid. Nevertheless, the analysis is not globally optimal since the analyser will produce an alarm of possible null-referencing for the call `v.inc()`, although such an error will concretely never occur. This comes from the 'basic form' of the transitive component (for a stack of an arbitrary depth, the analysis of the `pop` method cannot assert that the returned value is not null). The result becomes optimal for a stack of defined depth.

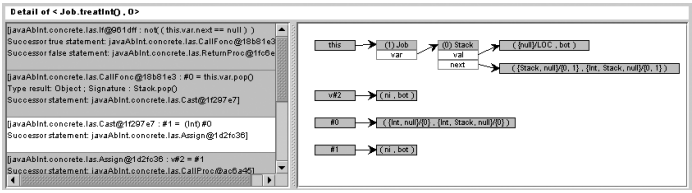
Let us now consider the analysis of the method `Main.main3()`. Snapshot 3 depicts the abstract store just before the cast check within the statement `v = (Int) var.pop()`. This time, the value mapped to the internal variable #0 does not permit to validate the cast, since it can represent an instance of `Int` or an instance of `CoupleInt`. A warning can then be raised.

The class `List` of Figure 8 implements reading and writing of homogeneous generic lists (on a simple input-output device implemented by the class `IO`). It has several extensions: `IntList` implements lists of integers, `L2List` lists of lists (and `IntL2List` lists of lists of integers), `StarList` *Lisp-like* lists (and `IntStarList` Lisp-like lists with basic integer values). We may attempt, in an optimization goal, to specialize the methods `List.readList()` and `List.writeList()` for the concrete classes `IntList`, `IntL2List` and `IntStarList`. To achieve this goal, we notably need precise type information about the target of the calls `l.getCell()` (within `List.readList()`) and `p.writeCell()` (within `List.writeList()`).

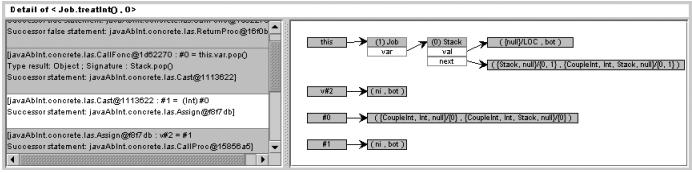
We just discuss the specialization of both methods for lists of lists of integers. In the other cases, the results are optimal. Snapshot 1 of Figure 9 provides a graphical view of the abstract semantics computed for `readList` (when analysing the method `List.main()`). Snapshot 2 depicts the local state



Snapshot 1: method table for `Job.treatInt()` (for the `Main.main1()` program)



Snapshot 2: detail of `Job.treatInt()` (for the `Main.main1()` program)



Snapshot 3: detail of `Job.treatInt()` (for the `Main.main3()` program)

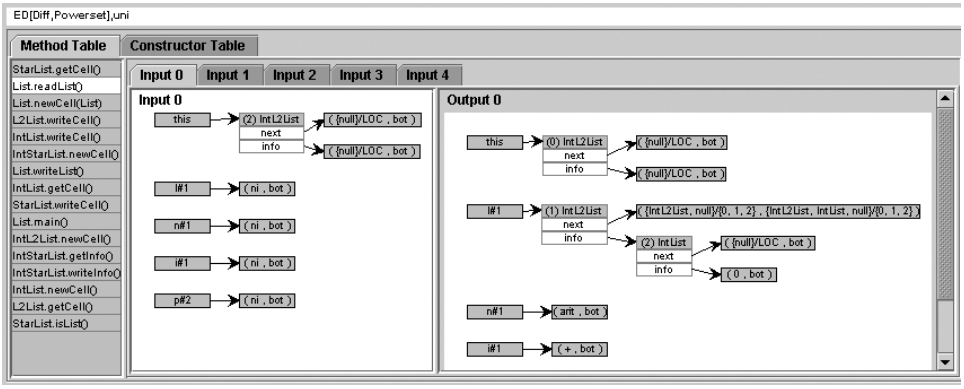
Fig. 7. Snapshots for the `Stack` program

```
abstract class List {
  List next;
  abstract List newCell();
  abstract void getCell();
  abstract void writeCell();

  List newCell(List tail){
  List l = newCell();
  l.next = tail; return l;
  }

  List readList(){
    List l = newCell(null);
    int n = IO.read(); int i = 1;
    while (i<= n){
      l.getCell();
      List p = newCell(l);
      l=p; i = i+1;
    }
    return l;
  }

  void writeList(){
    List p = next;
    while (!(p==null)){
      p.writeCell(); p = p.next;
    }
  }
```

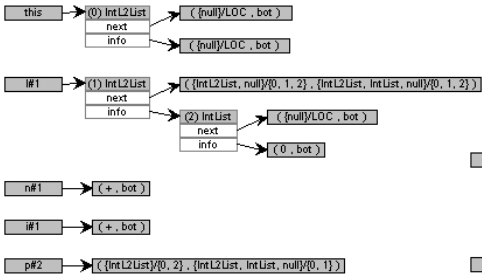
```
  void main() {
   List l=new IntList().readList();
   l.writeList();
   l=new IntL2List().readList();
   l.writeList();
   l=new IntStarList().readList();
   l.writeList();
  }
}

class IntList extends List {
  int info;
  List newCell(){ return new IntList(); }
  void getCell(){info=IO.read();}
  void writeCell(){IO.write(info);}
}

abstract class L2List extends List{
  List info;
  void getCell(){info = info.readList();}
  void writeCell(){info.writeList(); }
}

abstract class StarList extends List{
  List elem;
  boolean isList(){
    int n = IO.read(); return n == 0;
  }
```

```
  abstract void getInfo();
  abstract void writeInfo();

  void getCell() {
    if (isList()) {elem = readList(); }
    else { getInfo(); }
  }
  void writeCell(){
    if (elem == null){ writeInfo(); }
    else { elem.writeList(); }
  }
}
public class IntL2List extends L2List {
  List newCell(){
    IntL2List l = new IntL2List();
    l.info = new IntList(); return l;
  }
}
public class IntStarList extends StarList{
  int info;
  void getInfo() { info = IO.read(); }

  List newCell() {
    return new IntStarList();
  }
  void writeInfo() { IO.write(info); }
}
```

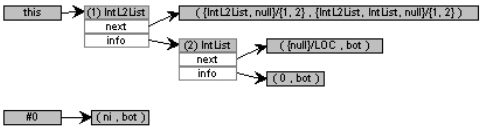
Fig. 8. The `ListBle` program



Snapshot 1: entries in the method environment for `readList()`



Snapshot 2: detail of the analysis of `readList()` for lists of lists of integers



Snapshot 3: detail of the analysis of `getCell()` for lists of lists of integers

Fig. 9. Snapshots for the `ListBle` program

before the call `l.getCell()` within the method `readList` for this entry. In this situation, the type of `l` is exact. Thus, the dynamic call can be replaced by a static one and, further, inlined. Snapshot 3 details the analysis for the entry in the method environment for this call. Again, the type of `this.info` is exactly known and the call can be made static and inlined. So, we have derived enough type information to obtain a completely specialized method (with two nested loops).

There is a single case where the results are not totally optimal: For the call `p.writeCell()` within the method `writeList`, we are only able to derive that `p` is either of type `IntList` or `IntL2List`. This result could be improved on with minor modifications of the abstract domain. For instance, we could use an improved version of the Distinctness Domain that would allow OR-Nodes to deal with the null value.

6 Related work

The work that we have presented in this paper faithfully follows the abstract interpretation methodology [9,8,10], which we believe adequate to master the complexity of designing a correct and generic framework for a large object oriented language such as Java. More specifically, we reuse several ideas that have proven successful in the design of GAIA, a generic system for the static analysis of Prolog [15]. The GAIA system, which was originally inspired by [3], is parameterized on an abstract domain (abstracting sets of substitutions) and it has been instantiated to many different abstract domains. Our structural abstract domains are related to the domain `Pattern` [18,15] and to the parametric domain *Pat(R)*, which generalizes `Pattern` by allowing one to enhance any abstract domain R with a structural component [6,7].

There are two major differences between our work and GAIA: Java is an imperative language with destructive updating and it is object-oriented. To deal with dynamic dispatch, we determine which methods can actually be executed and we abstractly execute all of them. The biggest difficulty is destructive updating notably because it makes inter-procedural analysis much more difficult: In logic programming, data structures existing before a predicate call cannot be destructively modified; they can only be more instantiated. Thus, the return of a call can be implemented by an (abstract) unification of the result of the call with the abstract state before the call (i.e., a *backward unification*). In imperative programming, data structures that are 'passed' to a call may be completely modified or replaced by other new structures built during the call. Nevertheless, our solution to this problem is also inspired by logic programming: Adding relational information, we explicitly introduce parts of the data structures that are equals by definition (i.e., the values of the fields before the call are equal to the values of their copies after the call); so the fields can be 'unified' with their copies to get a precise picture of the global situation after the return of the call.

In the field of object oriented programming, many 'concrete' type inference algorithms have been designed to replace virtual method calls by static ones, and, more generally, to specialize object-oriented programs (see, e.g., [1,4,12,16,23,26]). The emphasis is on obtaining the best tradeoff between speed and precision of the analyses. Since none of the cited work uses structural information, they a priori are less precise than ours. Further work is needed to see whether our approach is applicable to large Java programs. We foresee that widening operations will allow us to achieve virtually any desirable tradeoff although choosing the right widening operations is not trivial.

Our proposal also has similarities with pointer analysis but our structural domains have not been designed to achieve an efficient points-to anal-

ysis [2,24,25] or a precise shape analysis [5,11,13,22,21]. They are primarily designed to make any analysis (e.g., a type analysis) equally precise with or without resorting to (a reasonable form of) inlining. We have verified that this works on several examples such as our `Swap` program. Furthermore, it is possible to integrate various forms of pointer analysis to our structural domains on the basis of the abstract locations. There are two possible approaches to improve our structural domains for pointer analysis. Either we extend the structural domain themselves to make them comparable to shape analysis domains such as [5,11,13,22] or we construct a product domain similar to *Pat(R)* [6,7] by making the shape analysis work on abstract locations instead of program variables.

Conclusion and future work

We have presented an abstract interpretation framework for a subset of Java. This framework uses structural abstract domains, which makes it possible to extend the framework with additional analyses, and it provides a precise treatment of inter-procedural analysis, through the use of relational information. We see this framework as a first step towards a completely satisfactory abstract interpretation framework for Java. The needed improvements include addressing the complete Java language, further parameterizing the abstract domains, and dealing with incomplete source code. The contribution of this work is thus to provide the semantic basis for a complete system since many improvements will amount to add new but similar definitions and to tune the abstract domains and the algorithms.

In the near future, we plan to work along two main lines. On the one hand, we will investigate variants of the structural abstract domains to find the most interesting tradeoffs between precision and efficiency of the analyses, in different situations (e.g., optimization versus verification). On the other hand, we will extend the framework to the complete Java language (still without concurrency but with provision to analyze incomplete code).

References

[1] Agesen, O., *Constrained-based type inference and parametric polymorphism*, in: B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis (SAS'94)*, number 864 in LNCS (1994).

[2] Andersen, L., "Program Analysis and Specialisation for the C Programming Language," Ph.D. thesis, DIKU, University of Copenhagen (1994), (DIKU report 94/19).

[3] Bruynooghe, M., *A practical framework for the abstract interpretation of logic programs*, Journal of Logic Programming **10** (1991), pp. 91–124.

[4] Chambers, C., J. Dean and D. Grove, *Whole-Program Optimization of Object-Oriented Languages*, Technical Report 96–06–02, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, Washington 98195–2350 USA (1996).

[5] Chase, D. R., M. Wegman and F. K. Zadeck, *Analysis of Pointer and Structures*, in: *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language and Implementation*, White-Plains, New-York, 1990.

[6] Cortesi, A., B. Le Charlier and P. Van Hentenryck, *Combination of abstract domains for logic programming*, in: *Proceedings of the 21th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, 1994.

[7] Cortesi, A., B. Le Charlier and P. Van Hentenryck, *Combination of abstract domains for logic programming: Open product and generic pattern construction*, Science of Computer Programming **38(1–3)** (2000), pp. 27–71.

[8] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, Los Angeles, California, 1977, pp. 238–252.

[9] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *Conference Record of Sixth ACM Symposium on Programming Languages (POPL'79)*, Los Angeles, California, 1979, pp. 269–282.

[10] Cousot, P. and R. Cousot, *Abstract interpretation frameworks*, Journal of Logic and Computation **2(4)** (1992), pp. 511–547.

[11] Dor, N., M. Rodeh and M. Sagiv, *Checking cleanness in linked lists*, in: *Proceedings of the Seventh International Symposium on Static Analysis (SAS'2000)*, LNCS (2000).

[12] Grove, D., G. DeFouw, J. Dean and G. Chambers, *Call graph construction in object-oriented languages*, in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications Static Analysis (OOPSLA'97)*, 1997.

[13] Jones, N. and S. Muchnick, *A flexible approach to interprocedural data flow analysis and programs with recursive structures*, in: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'82)*, 1982.

[14] Le Charlier, B. and P. Van Hentenryck, *A general top-down fixpoint algorithm (revised version)*, Technical Report RR-93-022, Institute of Computer Science, University of Namur, Belgium, (also Brown University) (1993).

[15] Le Charlier, B. and P. Van Hentenryck, *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994), pp. 35–101.

[16] Lerner, S., D. Grove and G. Chambers, *Composing dataflow analyses and transformations*, in: *Proceedings of the ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'02)*, Portland, Oregon, 2002.

[17] Logozzo, F., *Class-level modular analysis for object oriented languages*, in: *Proceedings of the 10th Static Analysis Symposium (SAS '03*, Lectures Notes in Computer Science (2003).

[18] Musumbu, K., "Interprétation Abstraite de Programmes Prolog," Ph.D. thesis, Institute of Computer Science, University of Namur, Belgium (1990), in French.

[19] Pollet, I., "Towards a generic framework for the abstract interpretation of Java," Ph.D. thesis, Université Catholique de Louvain, Belgium, http://www.info.ucl.ac.be/~ipo (2004).

[20] Pollet, I., B. Le Charlier and A. Cortesi, *Distinctness and Sharing Domains for Static Analysis of Java Programs*, in: J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, number 2072 in Lecture Notes in Computer Science (2001).

[21] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, in: *Proceedings of the 26th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, 1999.

[22] Sagiv, M., T. Reps and R. Wilhem, *Solving shape-analysis problems in languages with destructive updating*, ACM Transactions on Programming Languages and Systems **20** (1998), pp. 1–50.

[23] Schultz, U., J. Lawall, C. Consel and G. Muller, *Towards automatic specialization of Java programs*, in: *Proceedings of ECOOP'99*, 1999, pp. 367–390.

[24] Shapiro, M. and S. Horwitz, *The effects of the precision of pointer analysis*, in: *Proceedings of the Fourth International Symposium on Static Analysis (SAS'1997)*, LNCS (1997), pp. 16–34.

[25] Steensgaard, B., *Points-to analysis in almost-linear time*, in: *Proceedings of ACM symposium on principles of programming languages*, 1996.

[26] Tip, F. and J. Palsberg, *Scalable propagation-based call graph construction algorithms*, in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'00)*, 2000.