Distinctness and Sharing Domains for Static Analysis of Java Programs

Isabelle Pollet[1⋆], Baudouin Le Charlier[1], and Agostino Cortesi[2⋆⋆]

[1] University of Namur, Belgium
[2] Ca' Foscari University, Italy

Abstract. The application field of static analysis techniques for object-oriented programming is getting broader, ranging from compiler optimizations to security issues. This leads to the need of methodologies that support reusability not only at the code level but also at higher (semantic) levels, in order to minimize the effort of proving correctness of the analyses. Abstract interpretation may be the most appropriate approach in that respect. This paper is a contribution towards the design of a general framework for abstract interpretation of Java programs. We introduce two generic abstract domains that express type, structural, and sharing information about dynamically created objects. These generic domains can be instantiated to get specific analyses either for optimization or verification issues. The semantics of the domains are precisely defined by means of concretization functions based on mappings between concrete and abstract locations. The main abstract operations, i.e., upper bound and assignment, are discussed. An application of the domains to source-to-source program specialization is sketched to illustrate the effectiveness of the analysis.

Keywords: Abstract Interpretation, Static Analysis, Type Analysis, Program Specialization.

1 Introduction

The application field of static analysis techniques for object-oriented programming is getting broader, ranging from compiler optimizations to security issues. This leads to the need of methodologies that support reusability not only at the code level but also at higher (semantic) levels, in order to minimize the effort of proving correctness of the analyses.

In this paper we introduce and discuss the foundations of a project aimed at defining and implementing a Java code analyzer based on abstract interpretation [9], a general semantics based methodology for static analysis that has been successfully applied to a large number of purposes mostly in declarative programming languages. In order to derive program properties that hold for every possible execution, an abstract interpreter 'executes' the program over a non

⋆ Supported by the Belgian National Fund for Scientific Research (FNRS).

⋆⋆ Partially supported by MURST projects 'Certificazione Automatica di Programmi mediante Interpretazione Astratta' and 'Interpretazione Astratta, Type Systems e Analisi Control-Flow'.

J. Lindskov Knudsen (Ed.): ECOOP 2001, LNCS 2072, pp. 77–98, 2001.

standard domain of (so-called) abstract values. In the case of type analysis, abstract values denote sets of types (i.e., the possible types of the actual instances that may arise at run time). The analysis is conservative, i.e., no erroneus result can be derived, but the results can be inaccurate in some cases. Inaccuracy is an obvious consequence of the fact that any non-trivial property of program execution is undecidable.

The reason to adopt abstract interpretation is twofold. First, the conceptual simplicity and soundness of this technique ensure that most properties of the resulting analysis (precision, completeness, modularity, scalability) depend only upon the choice of a concrete semantics, an abstract domain of properties, and abstract operations that safely approximate the corresponding concrete operations. Second, the design of abstract interpreters has already been successfully developed for other language paradigms (in particular for declarative programming) and we are confident that this approach may have a similar impact on object-oriented programs. Despite the great amount of scientific contributions on object-oriented languages, most of the works on semantics that may support static analyses mainly focused on Data Flow and Type System approaches, while disregarding abstract interpretation. Investigating this methodology for Java programs is thus one of the original contributions of this paper.

The first step in our work has been a precise description of an operational semantics for a sublanguage of Java [22]. The concrete semantics is expressed as a transition system on (finite) graph-descriptions of execution states. A graph-description represents both the environment (i.e., the mapping of each variable to its location), and the store (i.e., the mapping of each location to its value). We do not report on these concrete semantics issues in this paper; we focus instead on the definition of two generic abstract domains for type analysis, namely a distinctness domain and a sharing domain.

These two generic domains share the same key idea: abstract environments and stores look like concrete ones but the values of base types are disregarded (since we focus at this time on type analysis) and types are approximated when needed, keeping the structure as much as possible. This homomorphism between abstract and concrete domain facilitates the understanding of abstract structures.

The two generic abstract domains differ as they correspond to dual ways to look at structural sharing. In the first domain, distinctness of abstract locations can be interpreted as a definite information while structural sharing only means that the corresponding concrete structures may share. In the second domain, structural sharing is a definite information while distinctness of abstract locations only means that the corresponding concrete locations may be distinct. The first approach leads to the 'distinctness domain'. The second one leads to the 'sharing domain'.

Both abstract domains are generic as they are parameterized on a primitive abstract domain whose elements represent sets of concrete types. They can be specialized by a suitable choice of these abstract descriptions to yield specific

analysis. They basically integrate type analysis [1, 5] and shape analysis [6, 11, 12, 14, 25].

The rest of the paper is organized as follows. Section 2 outlines the main features of the domains, including a discussion of domain operations (orderings, upper bounds and convergence issues) as well as a discussion of the (abstract) assignment operation, which is one of the most essential and critical step in the design of the abstract semantics. Section 3 presents an application in the area of static detection of dynamic dispatching, describing the use of the abstract domain in program specialization. Sections 4 and 5 discuss related and future work.

We adopt in this paper an intuitive and informal presentation based on examples. Technical material, including definitions, algorithms and proofs, can be found in [22, 23]. We just give here the most basic definitions to provide some feel for the formal treatment behind the intuition.

2 Domains and Operations

The abstract interpretation methodology may be briefly described by the following three steps [9, 21].

1. Define a concrete (operational) semantics, i.e., a formal representation of concrete execution states and of the transition rules corresponding to statement executions. This step, of course, is language-dependent.
2. Define an abstract semantics, i.e., a non-standard domain whose elements represent sets of concrete execution states, and a suite of abstract operations that safely approximate the corresponding concrete ones.
3. Define a generic algorithm, parameterized on the abstract domain, that computes a (post-)fixpoint of the abstract semantics, thus yielding safe information about concrete program executions.

Once points 1 and 3 above are settled, the analysis can be easily tuned according to specific tasks. It just needs to choose a suitable abstract domain, and the correctness of the analysis will follow for free.

In the rest of this section we introduce the main features of concrete and abstract semantics that fit the picture above in the case of object-oriented programs. The abstract domains have been designed for Java but the same approach can be used for other object-oriented languages as well.

2.1 Concrete (Standard) Domain

The concrete domain[1] we consider is a classical product 'environment-store'. Its elements are graphs whose nodes are 'elementary instances' linked, by an edge, to their 'super-instance' and to their fields. Technically, a store is a mapping from an arbitrary and infinite set of *locations* to the set of values (basic values or instances). A *location* can be viewed as the address of a field or of a variable.

[1] Usually called 'standard domain'.

Figure 2.1 shows an element of the domain. This situation considers two variables x and y and the current instance. The variable x has a basic type, while y refers to an instance of a simple class whose field ch is equal to `null`. The current instance is of type C, which extends D. This situation happens, for instance, at the beginning of the execution of the method call `c.meth(7,a)` (this method is also depicted in Figure 2.1).

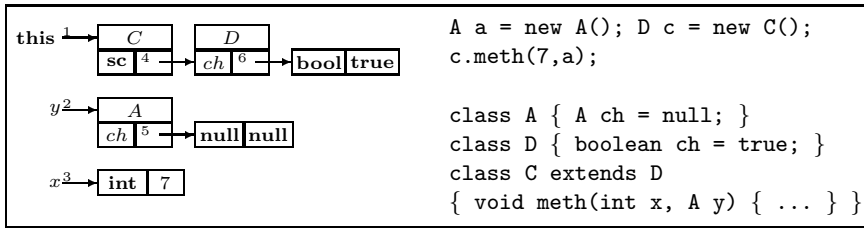

Fig. 2.1. The simple concrete situation S

The mathematical presentation of this domain is given in Definition 1. To complete this definition, we just need to add some constraints to the product $\mathbb{E}nv \times \$tore$ (see [22, 23]). In this definition, $Nclass$ is the set of all class names, while $Nfield$ is the set of possible field names and $Nvar$ the set of (local) variable names.

Definition 1 (Concrete Domain). *The set of* ***values*** *is defined as the disjoined union of the* ***basic values*** *and the* ***instances****, i.e.,* $\mathbb{V}al = \mathbb{B}ase + \mathbb{I}nst$ *where* $\mathbb{B}ase = \{\mathbf{bool}\} \times \mathbb{B}ool + \{\mathbf{int}\} \times \mathbb{Z} + \{(\mathbf{ni}, \mathbf{ni}), (\mathbf{null}, \mathbf{null})\}$ *and* $\mathbb{I}nst$ *is the largest subset of* $Nclass \times (Nfield + \{\mathbf{sc}\} \nrightarrow \mathbb{L}oc)$ *such that if* (nc, v) *belongs to* $\mathbb{I}nst$*, then* v *is one-to-one. The set* $\mathbb{E}nv$ *of* ***environments*** *is defined as the largest subset of* $Nvar + \{\mathbf{this}\} \nrightarrow Type \times \mathbb{L}oc$ *such that if* e *belongs to* $\mathbb{E}nv$ *and if* x *and* y *are two separate elements of the domain of* e*, then* $p_2(e(x)) \neq p_2(e(y))$*, with* $Type = Nclass + \{\mathbf{null}, \mathbf{int}, \mathbf{bool}, \mathbf{ni}\}$*. The set of* ***stores*** *is defined as* $\$tore = \mathbb{L}oc \nrightarrow \mathbb{V}al$*.*

2.2 Abstract Domains

The abstract domains are very similar to the concrete one but they introduce a new kind of values: the *abstract types*. For each concrete instance, either we keep complete information about its structure at the abstract level, or abstract information about its type, that we call an *abstract type*.

Our framework is completely parametric on the choice of type abstractions. We just need an abstract domain $Type^{\#}$, equipped with an order relation and a least upper bound operator. This domain should be related to the set of types $Type$ by a concretization function $\gamma : Type^{\#} \to \wp(Type)$. For instance, $Type^{\#}$ may be set equal to the powerset $\wp(Type)$, or to the set $Type$ itself (with $\gamma(t) = \{t' \preceq t\}$, i.e., the set of types specializing t); as a further example, in case of

security analyses $Type^{\#}$ may be defined as a partition of $Type$, according to protection domains.

An abstract situation approximating the concrete situation S is depicted in Figure 2.2 (we assume that $Type^{\#} = \wp(Type)$). In this abstract situation, the type of the current instance is approximated and all the structural information concerning this instance is lost.

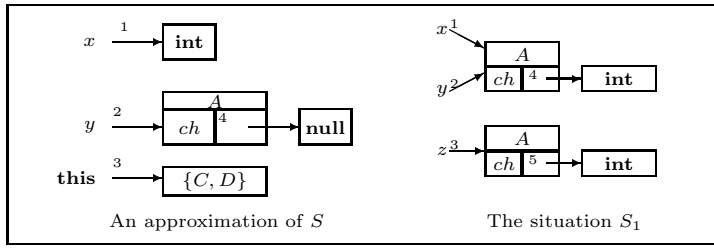

Fig. 2.2. Two abstract situations

The mathematical presentation of the abstract domains is below. Like in the concrete case, to complete this definition, we just need to add some constraints to the product $\mathbb{E}nv^{\#} \times \$tore^{\#}$ (see [22, 23]).

Definition 2 (Abstract Domain). *The set of* ***abstract values*** $\mathbb{V}al^{\#}$ *is defined as* $\mathbb{V}al^{\#} = Type + Type^{\#} + \mathbb{I}nst^{\#}$ *where* $\mathbb{I}nst^{\#}$ *is the largest subset of* $Nclass \times (Nfield + \{\mathbf{sc}\} \longrightarrow \mathbb{L}oc^{\#})$ *such that if* (nc, v) *belongs to* $\mathbb{I}nst^{\#}$ *then* v *is one-to-one. The set* $\mathbb{E}nv^{\#}$ *of* ***abstract environments*** *is defined as the largest subset of* $Nvar + \{\mathbf{this}\} \rightarrowtail Type \times \mathbb{L}oc^{\#}$, *such that if* e *belongs to* $\mathbb{E}nv^{\#}$ *and if* x *and* y *are two different elements of the domain of* e, *then* $p_2(e(x)) \neq p_2(e(y))$. *The set of* ***abstract stores*** *is defined as* $\$tore^{\#} = \mathbb{L}oc^{\#} \rightarrowtail \mathbb{V}al^{\#}$.

In the rest of the paper we discuss two generic interpretations of Definition 2, called distinctness and sharing domain, that differ in the way they express information about structural sharing of (concrete) data structures. In the distinctness domain, we assume that two distinct abstract instances always stand for different concrete instances. In the sharing domain, we dually assume that every abstract instance stands for a single concrete one. Such an information is useful notably to improve the accuracy of abstract assignment.

Consider the abstract situation S_1 of Figure 2.2. This situation gives exact type information, but what does it say about structure sharing? Concrete situations potentially (i.e., with the same type information) represented by S_1 (ignoring the exact integer values) are given in Figure 2.3.

In the distinctness domain, the situation S_1 expresses that z is certainly distinct from x and y (but says nothing about the sharing of x and y). Thus the set of situations represented by S_1 in restricted to $\{s_1, s_2\}$. On the contrary, in the sharing domain S_1 expresses that x and y certainly share the same instance (but

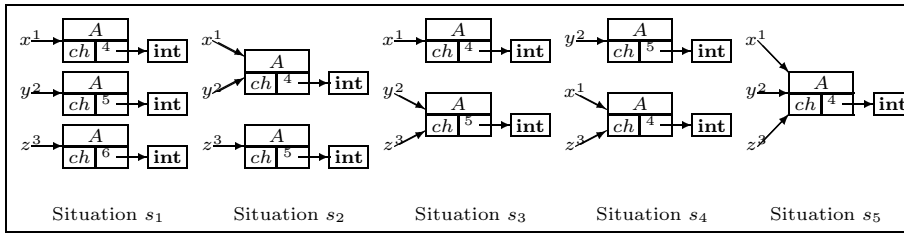

Fig. 2.3. Concrete situations potentially represented by S_1

says nothing about the distinction in regards of z). Thus, the set of represented situations is reduced to $\{s_2, s_5\}$. Briefly, we can say that, in the distinctness domain, the abstract sharing is an approximation, whereas, in the sharing domain it is an exact information.

To capture the intuition, we observe that there are obvious correspondences between concrete and abstract locations. Let us look again at Figure 2.3 and 2.2: in the case of s_1, we have the relation $f_1 = \{(1,a),(2,b),(3,c),(4,d),(5,d),(6,e)\}$, in the case of s_2, the relation $f_2 = \{(1,a),(2,b),(3,c),\ (4,d),(4,e)\}$ and, in the case of s_5, the relation $f_5 = \{(1,a),(2,b),\ (3,c),(4,d),(5,e)\}$. The relations f_1 and f_2 are both functions whereas f_2 and f_5 are both one-to-one (precisely the dual property of functionality).

The distinctness domain requires the existence of a (partial but onto) mapping of the concrete locations to the abstract locations, whereas the sharing domain requires a mapping (total but perhaps not onto) of the abstract locations to the concrete ones. These mappings must respect the types and the structure of the instances. The technical definitions of the concretization functions of these two domains rely on these mappings (see [22, 23]). We just give here the definition of the concretization function for the distinctness domain.

Definition 3 (Distinctness Domain: Concretization Function). *Let be* $a = (a_0, a_1)$ *belonging to* $\mathbb{E}nv^{\#} \times \$tore^{\#}$ *and* $d = (d_0, d_1)$ *belonging to* $\mathbb{E}nv \times \$tore$. *We will say that a* ***approximates*** d ***throughout*** f, *i.e.,* $d \xrightarrow{f} a$, *if and only if* a, d *and* f *satisfy the three following properties.*

1. $f : dom(d_1) \rightarrowtail dom(a_1)$
2. $d_0 = \{(v_1, t_1\delta_1), ..., (v_n, t_n\delta_n), (\mathbf{this}, t_{n+1}\delta_{n+1})\}$
   $a_0 = \{(v_1, t_1\alpha_1), ..., (v_n, t_n\alpha_n), (\mathbf{this}, t_{n+1}\alpha_{n+1})\}$
   $\forall i : 1 \leq i \leq n+1 : f(\delta_i) = \alpha_i$
3. $\forall l \in dom(f)$,
$$a_1(f(l)) = t \in Type \Rightarrow d_1(l) \in \mathbb{B}ase \wedge (p_1(d_1(l)) = t$$
$$a_1(f(l)) = e \in Type^{\#} \Rightarrow p_1(d_1(l)) \in \gamma(e)$$
$$a_1(f(l)) = (nc, va) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(l) = (nc, v) \in \mathbb{I}nst \\ dom(v) = dom(va) \\ \forall ch \in dom(v),\ va(ch) = f(v(ch)) \end{cases}$$

The ***concretization function*** $\gamma_d : \mathbb{D}^{\#} \rightarrow \wp(\mathbb{D})$ *of the distinctness domain is defined in* a *as* $\gamma_d(a) = \{d \mid \exists f,\ d \xrightarrow{f} a\}$.

Other domains can be defined by assuming other properties for the correspondences between concrete and abstract locations. For instance, requiring that the correspondence is one-to-one and functional leads to the (trivial) domain associating only situations with strictly the same sharing of instances (in our example, $\gamma(S_1) = \{s_2\}$). As another extreme example, accepting any correspondence (respecting the types) leads to a second (trivial) domain without any sharing information (in our example, $\gamma(S_1) = \{s_1, s_2, s_3, s_4, s_5\}$). However, these two domains are less attractive: the first one loses most of the structural information when an upper bound operation is applied, while the second one is very imprecise in the case of abstract assignment.

2.3 Orderings

Both abstract domains are endowed with a relation such that if an element a is smaller than another element a', all situations represented by a are also represented by a'. This means that the preorder is *coherent* with the concretization function. Consequently, the definition of the preorder is very similar to the definition of the concretization function. Again, the key idea is the existence of a relation between the locations of a and the locations of a' respecting the types and the structure.

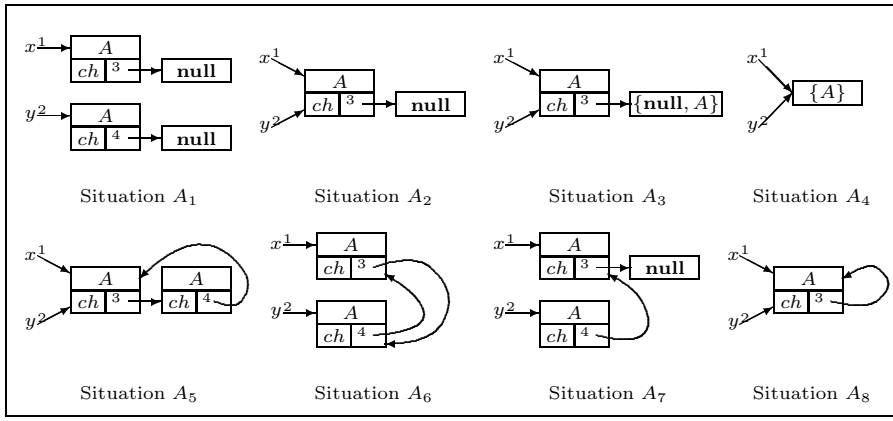

Fig. 2.4. Some abstract situations: which are approximations of others?

We illustrate the preorders on the situations in Figure 2.4. In the case of the distinctness domain, we directly see that $A_1 \leq A_2 \leq A_3 \leq A_4$. Indeed, A_1 gives an exact information about types and sharing whereas A_2 keeps exactly the same type information but loses the information about the distinction between x and y; A_3 is similar to A_2 but approximates the type of ch and A_4 loses all structural information but gives a coherent type information relatively to A_3. The situations A_5 and A_6 give the same type information but are not comparable

at the structural level. However, they both can be approximated by A_8, losing the distinction between the two instances but keeping the information that *ch* refers to one of these two instances. We also have that A_7 and A_8 can both be approximated by A_3, and thus by A_4. As an example, the mapping between A_7 and A_3 is $f = \{(1,1),(2,2),(3,3),(4,3)\}$.

Let us now explore the case of the sharing domain. We directly see that we have the chains $A_2 \leq A_1 \leq A_4$ and $A_2 \leq A_3 \leq A_4$. Indeed, A_2 gives exact information about types and sharing whereas A_1 keeps exactly the same type information but loses the information about the sharing between x and y. The situation A_3 is similar to A_2 but approximates the type of *ch*. Finally, A_4 loses all structural information but gives a coherent type information relatively to A_3, or to A_1. This time, A_3 is not comparable to A_1: A_1 gives and exact type information but says nothing about the sharing of x and y, whereas A_3 says that x shares with y but gives an approximation for the type of $x.ch$. The situation A_8 gives exact type and structural information, whereas the situation A_5 introduces a doubt on the length of the list (length of one or two cells). We have thus that $A_8 \leq A_5$ and, for similar reasons, we get $A_8 \leq A_6$. We also have that $A_8 \leq A_3$ and that all the situations are approximated by A_4. As an example, the mapping between A_8 and A_5 is $g = \{(1,1),(2,2),(3,3),(4,3)\}$.

The technical definitions of the preorders[2] are similar to the respective definitions of the concretization functions. See [22, 23] for these definitions and the coherence proofs.

2.4 Upper Bounds

We now tackle the problem of upper bound operators. A major feature of the distinctness domain is that it has no least upper bound operator[3], as we shall show it immediately on an example. Nevertheless, practically, we are satisfied with an upper bound operator (sufficiently precise however). An algorithm aimed at computing such an operator can be found in [23].

All situations in Figure 2.5 refer to the same environment that uses two variables x and y, of type B. The class B contains one field *ch* of type C. The class C is extended by D. Let us try to construct the least upper bound of C_1 and C_2. In the situation C_2, x and y share the same instance (i.e., at the concrete level, they can share or not). Thus, in the upper bound, they have to share as well. Furthermore, in the situation C_1, $x.ch$ refers to an instance of type C whereas, in C_2, it refers to an instance of D. Consequently, we approximate the type of $x.ch$ (which is equal to $y.ch$) and we get the situation C_3 as the upper bound. But we can reason in another way: C_1 and C_2 provide the same structure for y. Thus, we keep it in the upper bound. However, in C_2, x shares

[2] We do not get orders only because of the arbitrary choice of locations: it can be proven that two abstract elements a_1 and a_2 are *intuitively equivalent* (i.e., the same up to location renaming) if and only if $a_1 \leq a_2$ and $a_2 \leq a_1$.

[3] With the theoretical consequence that it lacks the property of being a Galois connection.

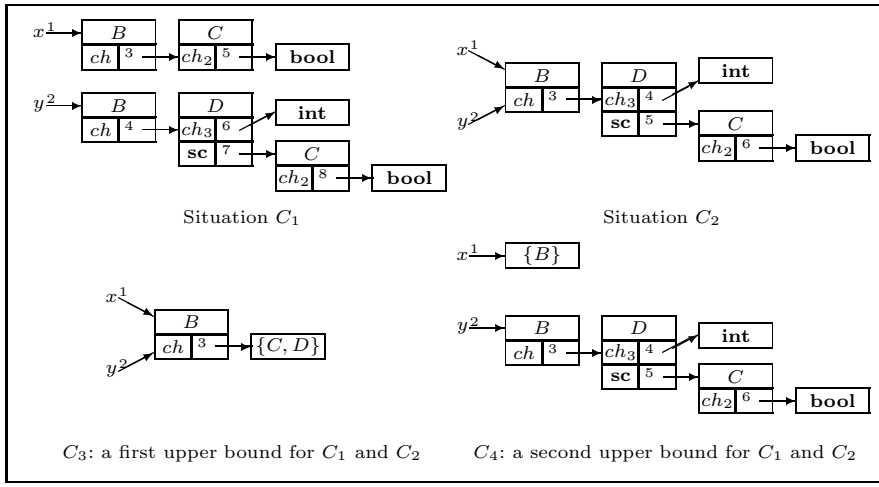

Fig. 2.5. The least upper bound does not exist in the distinctness domain.

with y but the structure of x in C_1 is incompatible with the structure of y. Thus we forget about the structure of x and we get the situation C_4 .

Since the situations C_3 and C_4 are not comparable, we have to conclude that the least upper does not always exist in this domain. Actually, we get a set of 'minimal approximations' instead of a 'minimum approximation'. Intuitively, this 'undesirable' feature stems from the fact that, in general, there are several ways to enforce structure sharing.

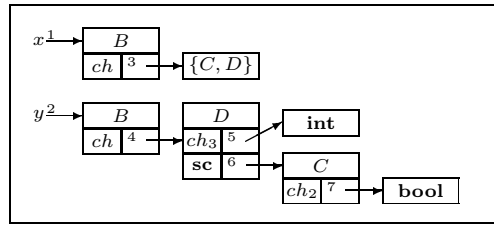

Fig. 2.6. The least upper bound for the sharing domain

We now turn to the sharing domain. We consider the same abstract situations as for the distinctness domain (they do not represent the same sets of concrete situations). In the situation C_1, x and y are referring to two different instances (i.e., just looking at these two instances, we know nothing about the concrete sharing). Thus, any upper bound must keep them different. The complete structure given for y is the same in the two situations, so we keep it. In

the situation C_1, $x.ch$ refers to an instance of C whereas in C_2 it refers to an instance of D. Therefore, the type of $x.ch$ has to be approximated and we get the situation C_6 (see Figure 2.6). We obtain here a unique most precise upper bound and, actually, in this domain, the least upper bound always exists. The algorithm computing this least upper bound can be found in [22, 23].

2.5 Convergence

Our abstract domains are both infinite. Hence, the convergence of the induced analysis requires either that the domains satisfy the ascending chain condition (i.e, every infinite ascending chain eventually stabilizes) or that some form of widening can be defined [21].

The distinctness domain satisfies the ascending chain condition. The proof is technical but it can be sketched intuitively. Let $(a_i)_{i\geq 0}$ be an ascending chain. For each couple (a_i, a_{i+1}), there is an onto function from the locations of a_i to the locations of a_{i+1}. This implies that the number of locations decreases and finally stabilizes. The subsequent elements differ only by the (concrete or abstract) types attached to their locations. Moreover, the sequences of types determined by corresponding locations of those elements are all increasing. Therefore they stabilize (if $Type^{\#}$ satisfies the ascending chain condition).

The sharing domain does not satisfy the ascending chain condition. A counter-example is depicted in Figure 2.7. However, the following property holds: every infinite ascending chain either eventually stabilizes, or each of its elements contains a cycle. To prove this property, we first remark that if a_2 contains a cycle and $a_1 \leq a_2$, then necessarily a_1 contains a cycle. Thus, if one element in a chain is cycle-free, so are the following ones. In the cycle-free case, the proof of the ascending chain condition is similar to the proof for the distinctness domain but the stabilization of the number of locations is harder to prove since this number may increase locally. In that case, however, the amount of sharing contained in an element decreases. Technically, the amount of sharing is defined as the 'number of potential new locations' that an element can produce. For instance, the situation A_2 of Figure 2.4 has a potential of one new location (in the sense that the location 3 can be approximated by two locations) whereas its approximation A_1 has a potential of zero new locations. Notice that this number can be defined only if there is no cycle. Finally, the cycle-free characterization of stabilizing chains can be used to define a widening operator that 'breaks the cycles' when the number of locations appears to increase continuously.

2.6 Abstract Assignments

In this section we discuss the most interesting abstract operation: assignment. We explain its abstract semantics by means of significant examples. A more formal presentation of this section can be found in citerapport.

Let us first focus on the distinctness domain. What is the effect of the assignment $x.ch =$ **null** on the situations D_1, D_2 and D_3 (see Figure 2.8)? In all these cases, the address of $x.ch$ is 4 and the new value to assign is **null**. In the

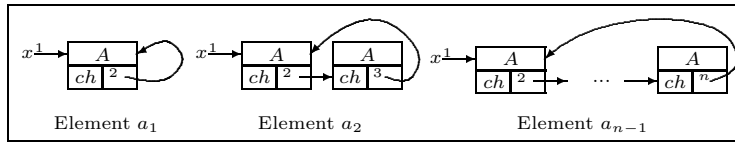

Fig. 2.7. An ascending chain in the sharing domain

case of D_1, the assignment happens exactly like in the concrete case and we get the situation D_4. Indeed, in this situation we know that x is distinct from y and thus the statement $x.ch = \mathbf{null}$ cannot influence any instance of D_1 different of x. This ideal situation happens when only one location refers to the modified instance.

To the contrary, in the situation D_2, we do not know if x and y are referring to different instances. Thus the assignment can be applied either on x or on both x and y. So the exact type of $y.ch$ is unknown. A correct approximation of the effect of the assignment is, for instance, D_5. The situation is still worse in case D_3, where z may share the same instance. D_6 represents an acceptable solution. Observe that what we have done in the two last cases is to compute an upper bound of the situation before the assignment and the situation that is obtained by simply updating the store like in the concrete case. These are situations where several locations refer to the modified instance.

Moreover, it is not always possible to compute the address of a 'designator' (i.e., a variable or field access expression) as in the previous examples. Indeed, some parts of the structures of the instances may be cut while introducing abstract types. For instance, what is the effect of $z.ch = \mathbf{null}$ on D_1? The abstract type gives no information about the sharing of any potential instance referred by z. It could share with x or with y (but not with both). Therefore the assignment may affect both instances. A correct approximation of the result in this case is D_7.

As for the 'designator' address, it is not always possible to compute the value of the assigned expression. For instance, we cannot completely evaluate the expression $z.ch$ in the situation D_1. Actually, this means that the situation D_1 gives no information at all about the type of this field. Consequently, the best information we can get is the one given by the type declaration of the field. Assuming that $z.ch$ is declared of type B, the situation D_8 approximates the application of the statement $x.ch = z.ch$ on the situation D_1 ($Cone(B)$ represents the set of types, including the **null** value, specializing B).

We now turn to the sharing domain. What is the effect of $x.ch = \mathbf{null}$ on E_1 (see Figure 2.9) in this case? Here, we have no information at all about the sharing between the different instances of A. Consequently, we must explore all possibilities of sharing relative to x and we must compute the effect of the assignment for all of them. Actually, it is sufficient to look, on the one hand, at the situation with the weakest sharing and, on the other hand, at the situation with the strongest. These two situations are, after the assignment, E_2 and E_3.

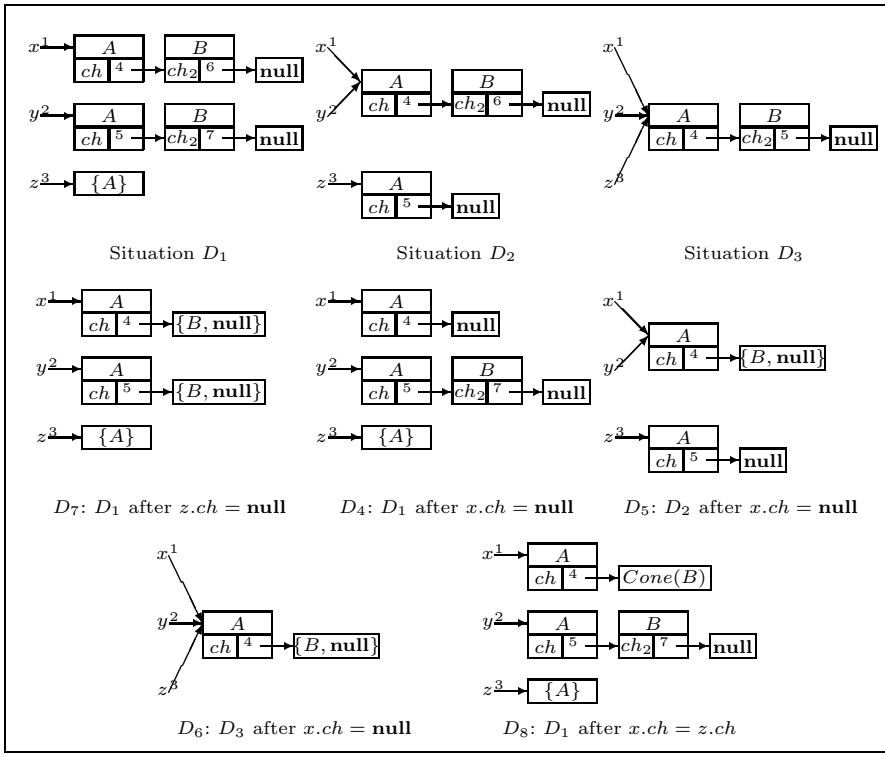

Fig. 2.8. Assignments in the distinctness domain

The abstract result E_4 of the assignment is then obtained by computing the least upper bound of E_3 and E_4. Notice that there is not always a single situation with the strongest possible sharing (like E_3). So, in general, we have to consider all of them.

The treatment of non evaluable addresses and values is similar to the case of the distinctness domain. As an example, the situation E_6 results from the application of the assignment $z.ch = \mathbf{null}$ to the situation E_5.

3 Application to Program Specialization

In this section, we briefly illustrate how our analysis can be used. The example is about program optimization. More precisely, we focus on program specialization, as getting rid of dynamic dispatching is one of the useful application of static type analysis.

For the sake of the presentation, we adopt a source-to-source approach of these transformations as it may help the reader's intuition about the actual benefits of the analysis. It is obvious that they need not to be implemented this way in a compiler.

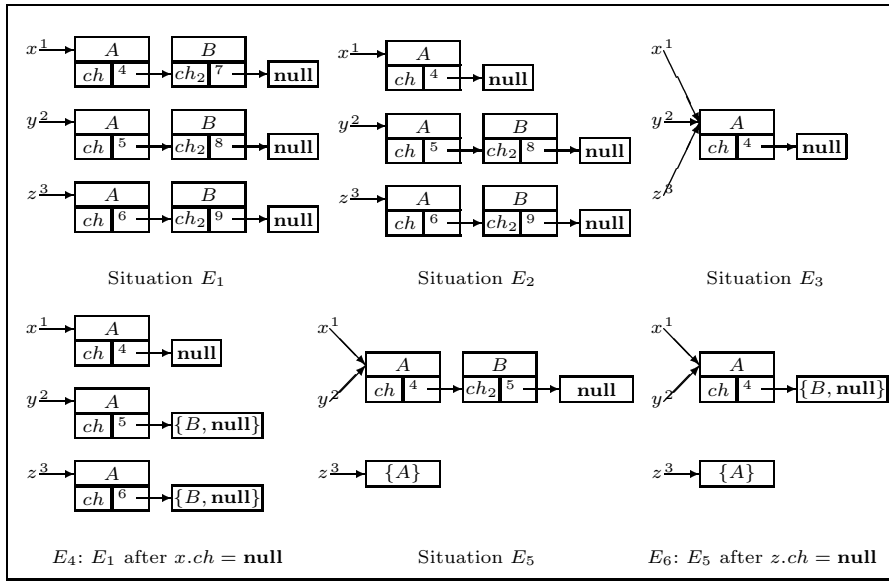

Fig. 2.9. Assignments in the sharing domain

We consider generic lists with only one main operation: the reading of a list (i.e., the method we want to analyze). We look at four Java classes[4] depicted in Figure 3.1: `List` (generic lists), `L2List` (lists of lists), `StringList` (lists of strings), and `StringL2List` (lists of string lists), with the inheritance relations: `null` < `StringList` < `List` and `null` < `StringL2List` < `L2List` < `List`.

We now attempt to specialize the generic method `readList()`[5] in the two concrete classes (i.e., `StringList` and `StringL2List`) to obtain more efficient versions.

Let us start with `StringList`. To perform this analysis, we can use a simple flat type domain: just keeping, for instance, the set of possible types for the variables. The abstract interpretation of `newCell(List)` applied to a `StringList` is trivial since it simply returns a new `StringList` instance. We now examine more cautiously the abstract interpretation of `readList()` applied to a `StringList`. Before going into the loop we simply get the mapping `this` $\rightsquigarrow$ {`StringList`}, `l` $\rightsquigarrow$ {`StringList`} since the method `getCell()` is applied on a `StringList` instance. After one iteration, this mapping becomes `this` $\rightsquigarrow$ {`StringList`}, `l` $\rightsquigarrow$ {`StringList`}, `p` $\rightsquigarrow$ {`StringList`} and will not be changed by any further iteration. This type information allows us to specialize the method in the `StringList` class in the following way.

[4] We have slightly simplified the actual code to shorten the presentation.

[5] In this example, we use dynamic dispatch to simulate parametric polymorphism: the purpose of the target object of the method `readList()` is only to provide the exact type of the list to be read, at call time.

```
// Generic Lists

public abstract class List{

private List next = null;

protected abstract List newCell();
protected abstract void getCell();

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

public List readList() {
  List l = newCell(null);
  while (!SimpleIO.isEndOfReading())
    {
    l.getCell();
    List p = newCell(l);
    l=p;
    }
  return (l);
  }
}

// Lists of Lists

public abstract class L2List
     extends List{

protected List info;

final protected void getCell() {
  info = info.readList();
  }
}
```

```
// Lists of Strings

public class StringList
      extends List{

private String info;

final protected List newCell() {
  StringList sl = new StringList();
  return(sl);
  }

final protected void getCell() {
  info = SimpleIO.getString();
  }
}

// Lists of Lists of Strings

public class StringL2List
     extends L2List{

final protected List newCell() {
  StringL2List l = new StringL2List();
  l.info = new StringList();
  return (l);
  }
}
```

Fig. 3.1. Classes for list manipulation

```
public List readList()        \\ specialized for StringList
{
  StringList l = (StringList) newCell(null);
  while( SimpleIO.isEndOfReading() == false )
  {
      l.GetCell();
      StringList p = (StringList) newCell(l);
      l = p;
  }
  return(l);
}
```

Finally, we may apply inlining, yielding the following segment of code (without any dynamic dispatching) that directly applies the overriding methods `getCell` and `newCell(List)` defined in class `StringList`.

```
public List readList()          \\ for StringList, after inlining
{
  StringList l = new StringList();
  l.next = null;
  while( SimpleIO.isEndOfReading() == false )
  {
      l.info = SimpleIO.getString();
      StringList p = new StringList();
      p.next = l;
      l = p;
  }
  return(l);
}
```

The case of `StringL2List` is more interesting. In this case, the simple domain used in the previous example is not sufficient, as the analysis must record also structural information. We thus turn now to our domain (we use the distinctness domain here).

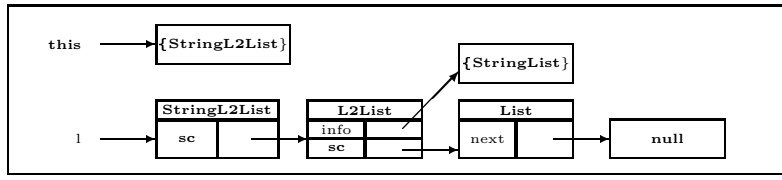

Fig. 3.2. Situation after the statement `List l = newCell(null)`

The type of the current instance is `StringL2List`. Consequently, when executing the call `newCell()` in the statement `List l = newCell(null)`, we surely execute the method of the class `StringL2List` yielding to the (abstract) situation[6] depicted in Figure 3.2. Let us examine the first iteration. We know the exact type of `l` and we apply the method `getCell()` of the class `StringL2List` (inherited from `L2List`). The statement `l.getCell()` does not modify the situation of Figure 3.2 (since we approximated the structure of the field `info`). If we now compute the effect of the sequence `List p = newCell(l); l = p;`, we obtain Figure 3.3. The computation of the body of the `while` statement is iterated and an upper bound operator is applied at each step, until a fixpoint is reached. This fixpoint is depicted in Figure 3.4 (in this case, it is simply an upper bound of the situations in Figures 3.2 and 3.3).

Just as for the `StringList` case, we can first specialize the method and then apply inlining for the calls to the methods `newCell(List)` and `getCell()`.

[6] Actually, we get a more precise information for the field `info` since we know its exact structure but we do not represent this structure here in order to simplify the pictures.

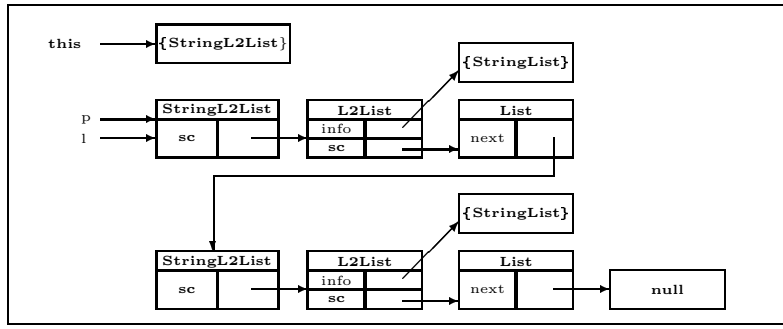

Fig. 3.3. Situation after one iteration

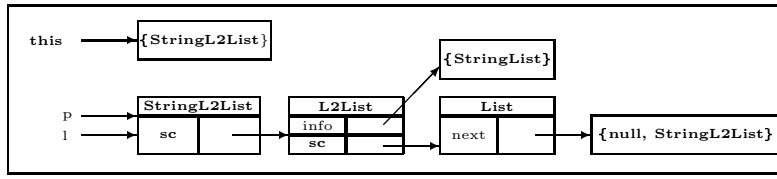

Fig. 3.4. Situation after the `while` statement

```
public List readList()       // inlining of getCell and newCell
{
  StringL2List l = new StringL2List();
  l.info = new StringList();
  l.next = null;
  while( SimpleIO.isEndOfReading() == false )
  {
      l.info = l.info.readList();
      StringL2List p = new StringL2List();
      p.info = new StringList();
      p.next = l;
      l = p;
  }
  return(l);
}
```

Moreover, we know here the exact type of the field `l.info`. Consequently, the specialized version of the method `readList()` for `StringList` will surely be executed in the statement `l.info = l.info.readList();`. Inlining the body of this method leads to the final version of the code with no dynamic dispatching anymore.

```
public List readList()       // inlining of getCell and newCell
{
  StringL2List l = new StringL2List();
  l.info = new StringList();
  l.next = null;
  while( SimpleIO.isEndOfReading() == false )
  {
      l.info = new StringList();
      l.info.next = null;
      while (SimpleIO.isEndOfReading() == false)
      {
         l.info.info = SimpleIO.getString();
         StringList p_1 = new StringList();
         p_1.next = l.info;
         l.info = p_1;
      }
      StringL2List p = new StringL2List();
      p.info = new StringList();
      p.next = l;
      l = p;
  }
  return(l);
}
```

4 Related Work

A difference between our work and other approaches is that our domains have not been designed by focusing on a single objective such as type analysis, or shape analysis, or sharing analysis (to name a few). Our approach is to define abstract domains that provide natural (i.e., understandable) and generic abstractions of the standard one. In this section, we explain how the domains can be useful for specific applications and we compare our work with related one in those areas.

4.1 Static Analysis of Object-Oriented Programs

Static analysis of object-oriented programs has attracted many researchers in recent years because mainly of the large potential for optimization related to dynamic dispatch (see e.g., [1, 3, 5]). The main difference between our approach and most existing work is that we propose a generic framework (for Java program analysis) explicitly based on abstract interpretation. We believe that this will allow us 1) to implement a generic and provably correct Java analyzer and 2) to easily integrate and combine various kinds of analyses in a modular and correct way. For instance, our approach is able to collect type information not only at the variable level (as in [5]) but also internally to data structures. The parametric domain $Type^{\#}$ can be instantiated in many ways to incorporate and improve upon other work. As an example, we can use the *Type heights* of [3] to perform escape analysis. (Note however that the approach of [3] is more relational than

ours. It is not clear yet how our abstract domains can be adapted for relational analysis.) As a third example, our approach can be used to perform a sharing analysis fulfilling the objectives of [2].

A main intuition behind our proposal is the idea of explicitly relating abstract and concrete locations by a functional mapping, which makes the interpretation of the abstract domain directly understandable. A similar technique has been recently used in [29] but the mapping between abstract and concrete objects is used to define the semantics of abstract descriptions which are first-order logic formulas, not graph-descriptions. The two approaches can be combined by using abstract locations instead of variable names as arguments in the basic predicates of [29].

4.2 Abstract Interpretation of Logic Programs

The work presented in this paper is greatly inspired by previous work on abstract interpretation of logic programs. Specifically, we apply to (a subset of) Java the same approach as in [4, 19] where a generic framework for the abstract interpretation of Prolog is proposed. Such a framework consists of an abstract semantics and of a general algorithm to compute it. The abstract semantics is proven to approximate safely the standard semantics, once and for all, and the general algorithm is proven to compute the abstract semantics as well. Thus, to get a particular correct analysis it is sufficient to provide an adequate abstract domain and to prove a few safety conditions for this domain. This approach has proven remarkably effective for the analysis of Prolog (see, e.g., [7, 8, 15, 19, 28]). The same approach is now applied for Java. Because of our past experiments with logic programs, we are confident that our approach can scale to 'real' Java programs. This belief must be validated by future work, however.

The abstract domains presented in this paper are similar to the domains presented in [19, 20] but less complex than the domains of [8, 28]; we are therefore confident that the analyses can be made practical. A major difference however is that, in Java, we have to deal with assignments. In the Prolog case, we use the domain `Pattern` [19, 20] which is similar to the sharing domain of this paper: this domain alone is enough to get precise mode analyses for Prolog [19] because instantiated structures cannot be changed; we only have to add a simple component to express distinctness of free variables. This simple component is meaningless in Java where any structure can be updated; this explains the need for the –dual– distinctness domain.

4.3 Pointer and Sharing Analysis

Many authors have worked on abstracting data structures that are dynamically created and modified by programs (see, e.g., [6, 13, 14, 24, 25, 27]). Such abstractions are useful e.g., for compile-time garbage collection, for program understanding, and for statically detecting errors such as `null` dereferencing. Both abstract domains proposed in this paper provide a form of shape analysis of the store (often called the heap) but are less expressive from this point of view than

the proposals of e.g., [6, 25, 14]. Nevertheless, our proposal has other strong points:

1. Our domains allow us to combine structural and type analyses, making it possible to infer more precise type information than most existing proposals (e.g., [1, 5]).
2. The information provided by our domains is easier to understand than in e.g., [6, 14, 25, 27] because the abstract store is homomorphic to the concrete one; therefore the domains are convenient for abstract debugging, for instance.
3. Our domains are less expensive to implement because some of the information that is represented by shape graphs in e.g., [6, 14, 25] is replaced by an abstract type information. This is similar to the difference between the abstract domains `Pattern` [19] and `Types`[18] in logic programming.
4. Our domains are generic, making it possible to combine them with other approaches in the spirit of [7]. For instance, we could combine the approaches of [6, 14, 25] with ours by using shape graphs as the 'abstract type information' when it becomes impossible to keep a (precise) homomorphic image of the concrete store. This approach has already been experienced for logic programming in [8], where the domains `Pattern` and `Types` are efficiently combined.

4.4 Static Detection of Errors

Many authors (and systems) use pointer analyses to statically detect errors in programs (see, e.g., [14, 16, 17]). Some of those proposals use a more powerful shape analysis than ours but our combination of shape and type analysis could allow us to get interesting results as well. (This must be validated experimentally.) Moreover, our type analysis is useful to validate casting conversions, which are unavoidable and error-prone in Java.

4.5 Program Specialization

The general idea of program specialization is to use static information about the program –and possibly about its intended use– to derive another equivalent and hopefully more efficient program [26]. In object-oriented programming this is generally done at the level of object code where dynamic calls to methods can be replaced by static calls and method in-lining [5]. Unfortunately, program specialization may lead to explosion of the program size. We think that our approach will allow us to derive precise type information for program specialization but we currently have no definite idea of how to use it optimally. (Asking the user to annotate the program to help the program specializer may be a solution, in some situations.)

5 Future Work

Our long-term goal is to implement a generic platform for the analysis, the verification, and the optimization of Java programs. We are currently only at the beginning of this undertaking.

At this stage, we have completed the definition of two equivalent concrete operational semantics for a simple yet significant subset of Java and the definition of the two abstract domains presented in this paper. We currently complete the definition of the abstract semantics. We have also started to implement our subset of Java as well as a graphical interface to depict concrete and abstract stores.

Our next step will be to implement a generic analyzer similar to GAIA [7, 8, 15, 19, 28] based on the abstract semantics and the domains. Then we will experiment with the system to evaluate the practical value of the domain and to tune the algorithms.

In the long term, we plan to address the following issues.

1. Variants of the abstract domains will be defined and evaluated. For instance, we can improve the expressiveness of the domains by adding various attributes to abstract locations as in e.g., [14]. More powerful (and expensive) domains can be obtained by introducing some form of 'disjunctive completion' in the domains as in [18], where so-called OR-nodes are used.
2. To get more powerful results, it is convenient to combine the two abstract domains of this paper into a single one by means of reduced product [10] or open product [7] operations. This will be investigated.
3. We will compare the usefulness of purely automatic analyses with respect to analyses of programs annotated by the programmer, in particular for program specialization.

Acknowledgements

The authors thank the ECOOP anonymous referees for their very constructive remarks as well as the reviewers of previous versions of this paper, who helped us a lot to produce this final version.

References

[1] O. Agesen. 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, Namur, September 1994. Springer-Verlag.

[2] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[3] B. Blanchet. Escape analysis for object oriented languages. application to java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications Static Analysis (OOPSLA'99)*, November 1999.

[4] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.
[5] C. Chambers, J. Dean, and D. Grove. Whole-Program Optimization of Object-Oriented Languages. Technical report, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, Washington 98195–2350 USA.
[6] D. R. Chase, 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, June 1990.
[7] A. Cortesi, 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, January 1994.
[8] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 23(3):237–278, June 1995.
[9] P. Cousot 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)*, pages 238–252, Los Angeles, California, January 1977.
[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of Sixth ACM Symposium on Programming Languages (POPL'79)*, pages 269–282, Los Angeles, California, January 1979.
[11] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, Oakland, California, April 1992. IEEE Computer Society Press, Los Alamitos, California.
[12] A. Deutsch. Interprocedural May-Alias Analysis for Pointers : Beyond k-Limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language and Implementation*, pages 230–241, Orlando, Florida, June 1994.
[13] A. Deutsch. Semantic models and abstract interpretation for inductive data structures and pointers. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based program Manipulation(PEPM'95)*, pages 226–228, New-York, June 1995.
[14] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proceedings of the Seventh International Symposium on Static Analysis (SAS'2000)*, LNCS. Springer-Verlag, September 2000.
[15] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic abstract interpretation algorithms for prolog: Two optimization techniques and their experimental evaluation. *Software Practice and Experience*, 23(4):419–459, April 1993.
[16] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language and Implementation*, 1996.
[17] P. Fradet, R. Gaugne, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proceedings of the European Symposium on Programming (ESOP'96)*, LNCS, 1996.
[18] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
[19] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.

[20] K. Musumbu. *Interprétation Abstraite de Programmes Prolog.* PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990. In French.

[21] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[22] I. Pollet. Sémantiques opérationnelles et domaines abstraits pour l'analyse statique de Java. DEA thesis, University of Namur, Belgium, September 1999.

[23] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. Research Paper RP-01-003, Institute of Computer Science, University of Namur, Belgium, 2001.

[24] M. Sagiv, 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, January 1999.

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

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

[27] J. Stransky. A Lattice for Abstract Interpretation of Dynamic (lisp-like) Structure. *Information and Computation*, 101:70–102, 1992.

[28] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of *Prop*. *Journal of Logic Programming*, 23(3):237–278, June 1995.

[29] E. Yahav. Verifying Safety Properties of Concurrent Java Programs Using 3-Valued Logic. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL'2001)*, January 2001.