Chapter 6
Abstract Interpretation

Nathan Wasser, Reiner Hähnle and Richard Bubel

6.1 Introduction

The previous chapters focused on the development of a *faithful* and *relatively complete* program logic for sequential Java. Consequently we obtain a formal language of high expressivity and a powerful calculus to prove the stated properties. Nevertheless expressivity comes with a cost, namely, an unpredictable degree of automation. This does not necessarily mean interaction with the theorem prover as such, but also the necessity to provide hand-crafted specifications like loop invariants or method contracts. The latter often needs to take idiosyncrasies of the theorem prover into account, at least, in regard of automation. This is also true in cases for which one is only interested in establishing simple properties like "No `NullPointerExceptions` are thrown."

Other techniques from static program analysis, like abstract interpretation as introduced by Cousot and Cousot [1977], utilize program abstraction to achieve full automation. But they pay with loss of precision that manifests itself in reduced expressivity (often only predefined properties can be expressed and ensured) and false positives.

In this chapter we show how to integrate abstract interpretation in JavaDL to achieve high automation while also maintaining high precision. Our approach has two main characteristics: i) abstraction of the state representation instead of the program, and ii) full precision until a loop or (recursive) method call is encountered. Only at those program points is abstraction applied and then only on the state region which might be modified by the loop or method. All other object fields or local variables keep their exact symbolic value.

6.2 Integrating Abstract Interpretation

6.2.1 Abstract Domains

We introduce notions commonly used in abstract interpretation [Cousot and Cousot, 1977]. The core of abstract interpretation is abstract domains for the types occurring within the program. Each abstract domain forms a lattice and there is a mapping between each concrete domain D^T (i.e., the externalization of a concrete program type) and its corresponding abstract domain A^T. Their relationship is established by two total functions:

$$\alpha : 2^{D^T} \to A^T \qquad (\textit{abstraction function})$$
$$\gamma : A^T \to 2^{D^T} \qquad (\textit{concretization function})$$

The abstraction function maps a set of concrete domain elements onto an abstract domain element and the concretization function maps each abstract domain element onto a set of concrete domain elements, such that $\alpha(\gamma(a)) = a$ and $C \subseteq \gamma(\alpha(C))$ holds. A pair of functions with the latter two properties is a special case of a Galois connection called Galois insertion. Figure 6.1 illustrates such a mapping. The arrows represent the concretization (from wheel to vehicle) and abstraction function (from vehicle to wheel).

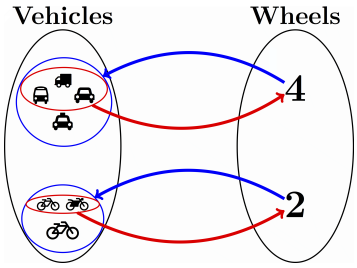


Figure 6.1 An example abstract domain: The concrete domain of vehicles is abstracted w.r.t. the number of wheels

We can now summarize the above into a formal definition of an abstract domain.

Definition 6.1 (Abstract Domain). Let D be a *concrete domain* (e.g., from a first-order structure). An *abstract domain* A is a complete lattice with partial order $\sqsubseteq$ and join operator $\sqcup$. It is connected to D with an *abstraction function* $\alpha : 2^D \to A$ and a *concretization function* $\gamma : A \to 2^D$ which form a Galois insertion [Cousot and Cousot, 1977], i.e. $\alpha(\gamma(a)) = a$ and $C \subseteq \gamma(\alpha(C))$. In this chapter we only deal with countable abstract domains.

Let $f : A \to A$ be any function. The monotonic function $f' : A \to A$ is defined as $f'(a) = a \sqcup f(a)$. If $\mathscr{A}$ satisfies the ascending chain condition (trivially the case if

$\mathscr{A}$ has finite height), then starting with any initial input $x \in A$ a least fixed point for f' on this input can be found by locating the stationary limit of the sequence $\langle x'_i \rangle$, where $x'_0 = x$ and $x'_{n+1} = f'(x'_n)$.

Abstract interpretation makes use of this when analyzing a program. Let p be a loop, x the only variable in p and $a \in A$ the abstract value of x before execution of the loop. Then we can see f as the abstract semantic function of a single loop iteration on the variable x. The fixed point for f' is an abstract value expressing an overapproximation of the set of all values of x before and after *each* iteration. Therefore it is sound to replace the loop with the assignment $\mathtt{x} = a$.

If $\mathscr{A}$ does not satisfy the ascending chain condition, there may not be a stationary limit for $\langle x'_i \rangle$. In these cases a *widening operator* is required.

Definition 6.2 (Widening Operator $\cdot\nabla\cdot$). A widening operator for an abstract domain $\mathscr{A}$ is a function $\nabla : A \times A \to A$, where

1. $\forall a, b \in A.\ a \sqsubseteq a \nabla b$
2. $\forall a, b \in A.\ b \sqsubseteq a \nabla b$
3. for any sequence $\langle y'_n \rangle$ and initial value for x'_0 the sequence $\langle x'_n \rangle$ is ultimately stationary, where $x'_{n+1} = x'_n \nabla y'_n$.

If $\mathscr{A}$ has a least element $\bot$, it suffices to use this as the initial value for x'_0, rather than proving the property for all possible initial values.

Abstract domains come traditionally in two flavors *relational* and *nonrelational*. Advantages with relational abstract domains are expressiveness and the abilities to easily formulate often-occurring and helpful abstract notions such as $i \leq a.\mathtt{length}$. The advantage of nonrelational abstract domains is their ease of use from an implementation standpoint, as nonrelational abstract domains care only about the actual variable being updated, rather than having potential to change multiple values at once. We choose a third path: using nonrelational abstract domains but including invariant suggestions which can model certain relational-style expressions such as the example $i \leq a.\mathtt{length}$.

To achieve a seamless integration of abstract domains within JavaDL, we refrain from the introduction of abstract elements as first-class members. Instead we use a different approach to refer to the element of an abstract domain:

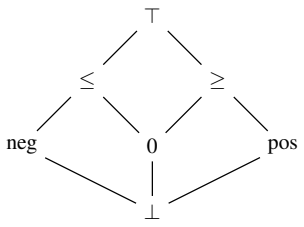
Definition 6.3 ($\gamma_{a,\mathbb{Z}}$-symbols). Given a countable abstract domain $A = \{a_1, a_2, \ldots\}$. For each abstract element $a_i \in A - \{\bot\}$ there

- are infinitely many constant symbols $\gamma_{a,j} \in \mathrm{FSym},\ j \in \mathbb{N}$ and $\gamma^{\mathscr{M}}_{a_i,j} \in \gamma(a_i)$,
- is a unary predicate χ_{a_i} where $\chi^{\mathscr{M}}_{a_i}$ is the characteristic predicate of set $\gamma(a_i)$.

The interpretation of a symbol $\gamma_{a_i,j}$ is restricted to one of the concrete domain elements represented by a_i, but otherwise not fixed. In other words, the only guarantee about (restriction on) the actual value of $\gamma_{a_i,j}$ is to be an element of $\alpha(a_i)$.

6.2.2 Abstractions for Integers

In this subsection, we introduce a simple abstract domain for integers, which we use to illustrate our approach. This abstract domain is called *Sign Domain* and shown in Figure 6.2. As its naming suggests, it abstracts from the actual integer values and



$$\begin{aligned}
\gamma(\top) &= \mathbb{Z} \\
\gamma(\leq) &= \{i \in \mathbb{Z} \mid i \leq 0\} \\
\gamma(\geq) &= \{i \in \mathbb{Z} \mid i \geq 0\} \\
\gamma(\text{neg}) &= \{i \in \mathbb{Z} \mid i < 0\} \\
\gamma(\text{pos}) &= \{i \in \mathbb{Z} \mid i > 0\} \\
\gamma(0) &= \{0\} \\
\gamma(\bot) &= \emptyset
\end{aligned}$$

Figure 6.2 Sign Domain: An abstract domain for integers

distinguishes them only w.r.t. their sign. The associated abstraction and concretization function obviously form a Galois connection.

The abstract domain is integrated into JavaDL by adding $\gamma_{a,\mathbb{Z}}$ symbols and their characteristic predicates χ_a for $a \in \{\text{neg}, 0, \text{pos}, \leq, \geq, \top\}$. The characteristic predicates are defined as follows:

$$\begin{aligned}
\forall \mathit{int}\ x; (\chi_0(x) &\leftrightarrow x \doteq 0) & \forall \mathit{int}\ x; (\chi_\top(x) &\leftrightarrow \text{true}) \\
\forall \mathit{int}\ x; (\chi_{\text{neg}}(x) &\leftrightarrow x < 0) & \forall \mathit{int}\ x; (\chi_{\text{pos}}(x) &\leftrightarrow x > 0) \\
\forall \mathit{int}\ x; (\chi_\leq(x) &\leftrightarrow x \leq 0) & \forall \mathit{int}\ x; (\chi_\geq(x) &\leftrightarrow x \geq 0)
\end{aligned}$$

6.2.3 Abstracting States

We now have all the parts together to explain how to abstract a given program state. We go further and embed the approach in a general notion of weakening, which provides us also with a natural soundness notion.

Given the following sequent:

$$c \geq 5 \Longrightarrow \{\texttt{i} := c+1\}\,[\texttt{i++;}]\texttt{i} > 0$$

The idea is to provide an *abstraction* rule that rewrites the above rule into:

$$c \geq 5 \Longrightarrow \{\texttt{i} := \gamma_{pos,1}\}\,[\texttt{i++;}]\texttt{i} > 0$$

where we replaced the 'more' complicated expression $c+1$ on the right hand side of the update to $\mathtt{i}$ by a simpler gamma symbol. The latter sequent preserves the knowledge about the sign of $\mathtt{i}$ under which the box formula is evaluated (namely that it is strictly greater than 0), but we lose all knowledge about $\mathtt{i}$'s actual value. More formally, if the latter sequent is valid then also the first one is valid. We call the update $\mathtt{i} := \gamma_{pos,1}$ *weaker* than $\mathtt{i} := c+1$ as the first one allows more reachable states: For the sequent to be true, the formula behind $\mathtt{i} := \gamma_{pos,1}$ must be true in all Kripke structures $\mathscr{K}$, i.e., for any positive value of i as $\gamma_{pos,1}$ does not occur anywhere else in the sequent. The original sequent only requires the formula behind the update to be true for all values strictly greater than 5.

We can formalize the weakening notion by introducing the update weakening rule from [Bubel et al., 2009]:

$$\mathsf{weakenUpdate}\ \frac{\Gamma, \mathscr{U}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\mathscr{U}'(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \qquad \Gamma \Longrightarrow \mathscr{U}'\varphi, \Delta}{\Gamma \Longrightarrow \mathscr{U}\varphi, \Delta}$$

where $\bar{\mathtt{x}}$ denotes a sequence of all program variables occurring as left-hand sides in $\mathscr{U}$ and $\bar{\mathtt{c}}$ are fresh Skolem constants used to store the values of the variables $\bar{\mathtt{x}}$ under update $\mathscr{U}$. The formula $\exists\bar{\gamma}.\varphi$ is a shortcut for $\exists\bar{y}.(\chi_{\hat{a}}(\bar{y}) \wedge \psi[\bar{\gamma}/\bar{y}])$, where $\bar{y} = (y_1, \ldots, y_m)$ is a list of fresh first-order variables of the same length as $\bar{\gamma}$, and where $\psi[\bar{\gamma}/\bar{y}]$ stands for the formula obtained from ψ by replacing all occurrences of a symbol in $\bar{\gamma}$ with its counterpart in $\bar{y}$. This rule allows us to abstract any part of the state with a location-wise granularity.

Performing value-based abstraction becomes thus simply the replacement of an update by a weaker update. In particular, we do not perform abstraction on the program level, but on the *symbolic state* level. Thus abstraction needs to be defined only on symbolic states (updates) and not on programs.

6.3 Loop Invariant Generation

In this section we describe how to use update weakening to automatically infer loop invariants that allow us to verify unbounded loops without the need for a user provided loop invariant. To describe the approach we restrict ourselves to simple program variables of integer type. We discuss extensions for objects and in particular arrays in a later section.

As indicated earlier we intend to perform abstraction on demand when reaching a loop (or recursive method call), as those cases require user interaction in the form of loop specification or method contracts. Our aim is to avoid this tedious work. We solve this by two steps (i) an adapted loop invariant rule which allows one to integrate value abstraction as part of the anonymizing update, and (ii) a method to compute the abstracted state.

Assume a proof situation in which we encounter a loop. To reason about the loop's effect, we use the following rule:

$$\textsf{invariantUpdate}\quad \frac{\begin{array}{l}\Gamma, \mathscr{U}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\mathscr{U}'_{mod}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \mathscr{U}'_{mod} g,\ \mathscr{U}'_{mod}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\mathscr{U}'_{mod}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \mathscr{U}'_{mod} \neg g \Longrightarrow \mathscr{U}'_{mod}[r]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \mathscr{U}[\texttt{while (}g\texttt{) \{}p\texttt{\}; }r]\varphi, \Delta}$$

With its three premisses and its basic structure, the invariantUpdate rule still resembles the classical loop invariant rule. The role of the loop invariant is taken over by the anonymizing update $\mathscr{U}'_{mod}$. The idea is to be smarter when anonymizing locations that might possibly be changed by the loop. Instead of anonymizing these locations by fresh Skolem constants, losing all information about their value and retroactively adding some knowledge back using a loop invariant, we use fresh $\gamma_{a,\mathbb{Z}}$ symbols. This way we keep some information about the value of these locations, namely, that their value remains within the concretization $\gamma(a)$ of the abstract element a. As we no longer have a traditional loop invariant, the lower bound of precision loss for the anonymized locations is given by the granularity of our abstract domain.

The rule's third premiss is the use case and represents the case where we have just exited the loop. The reachable states after the loop are contained in the set of all states reachable by update $\mathscr{U}'_{mod}$ strengthened by the fact that only those states need to be considered where the loop's guard evaluates to false. For all those states we have to show that after symbolic execution of the remaining program the property to prove holds. The second premise ensures that $\mathscr{U}'_{mod}$ is a sound approximation of all states reachable by any loop iteration. The first premiss ensures that the entry state is also contained in $\mathscr{U}'_{mod}$.

Example 6.4. Given the following sequent

$$\mathtt{i} \geq 0 \Longrightarrow \{\mathtt{n} := 0\}\,[\texttt{while (i>0) \{i--; n++;\}}](\mathtt{i} \doteq 0 \wedge \mathtt{n} \geq 0)$$

To apply rule updateInvariant we need to provide $\mathscr{U}'_{mod}$. Intuitively, we see that the loop modifies both variables i and n. About n we know that it is initially 0 and afterwards only increased. In case of i we know that its initial value is nonnegative and decreased by one with each loop iteration. Hence, we can conclude that both i and n are always covered by abstraction $\geq$. This gives us the following anonymizing update $\mathscr{U}'_{mod}$

$$\mathtt{i} := \gamma_{\geq,1} \,||\, \mathtt{n} := \gamma_{\geq,2}$$

The resulting proof goals after the rule application are: For the initial case

$$\begin{array}{l}\mathtt{i} \geq 0, \{\mathtt{n} := 0\}(\mathtt{i} \doteq c_1 \wedge \mathtt{n} \doteq c_2) \Longrightarrow \\ \quad \exists y_1, y_2.\big(\chi_{\geq}(y_1) \wedge \chi_{\geq}(y_2) \wedge \{\mathtt{i} := y_1 \,||\, \mathtt{n} := y_2\}(\mathtt{i} \doteq c_1 \wedge \mathtt{n} \doteq c_2)\big)\end{array}$$

which can easily be proven valid by choosing i for y_1 and 0 for y_2 as instantiations of the existential formula on the right sequent side. The second branch proving that the update describes all possible values of i and n after any loop iteration is

$$\begin{array}{l}\mathtt{i} \geq 0, \{\mathtt{i} := \gamma_{\geq,1} \,||\, \mathtt{n} := \gamma_{\geq,2}\}(\mathtt{i} > 0), \\ \quad \{\mathtt{i} := \gamma_{\geq,1} \,||\, \mathtt{n} := \gamma_{\geq,2}\}[\mathtt{i{-}{-};n{+}{+};}](\mathtt{i} \doteq c_1 \wedge \mathtt{n} \doteq c_2) \Longrightarrow \\ \quad\quad \exists y_1, y_2.\big(\chi_{\geq}(y_1) \wedge \chi_{\geq}(y_2) \wedge \{\mathtt{i} := y_1 \,||\, \mathtt{n} := y_2\}(\mathtt{i} \doteq c_1 \wedge \mathtt{n} \doteq c_2)\big)\end{array} .$$

This sequent can also be proven directly. After executing the loop body, applying the updates and some simplifications the above sequent becomes

$$\begin{array}{l}\mathtt{i} \geq 0,\ \gamma_{\geq,1} > 0,\ \gamma_{\geq,1} - 1 \doteq c_1,\ \gamma_{\geq,2} + 1 \doteq c_2 \Longrightarrow \\ \quad\quad \exists y_1, y_2.\big(\chi_{\geq}(y_1) \wedge \chi_{\geq}(y_2) \wedge \{\mathtt{i} := y_1 \,||\, \mathtt{n} := y_2\}(\mathtt{i} \doteq c_1 \wedge \mathtt{n} \doteq c_2)\big)\end{array}$$

by choosing $\gamma_{\geq,1} - 1$ for y_1 and $\gamma_{\geq,2} + 1$ for y_2 we can prove the sequent as $\chi_{\geq}(\gamma_{\geq,2} + 1)$ is obviously true and the truth of $\chi_{\geq}(\gamma_{\geq,1} - 1)$ follows from the formula $\gamma_{\geq,1} > 0$ which is part of the antecedent (obtained from the knowledge that the loop guard is true).

Finally the last proof goal to be shown valid is

$$\{\mathtt{i} := \gamma_{\geq,1} \,||\, \mathtt{n} := \gamma_{\geq,2}\}(\neg \mathtt{i} > 0) \Longrightarrow \{\mathtt{i} := \gamma_{\geq,1} \,||\, \mathtt{n} := \gamma_{\geq,2}\}[](\mathtt{i} \doteq 0 \wedge \mathtt{n} \geq 0)$$

which, once we derive that `i` is 0 from the antecedent, is trivial.

The question remains how to find a good candidate for $\mathscr{U}'_{mod}$ automatically. The solution is to start a side proof which unwinds the loop; once the loop body has been symbolically executed, we join the updates of all open branches by assigning each changed location the smallest abstract domain element that encompasses all of its potential values on the different branches. Repeat unwinding the loop until the update created by the join does not change any longer. The such obtained update is a sound candidate for $\mathscr{U}'_{mod}$.

Definition 6.5 (Joining Updates). The update join operation is defined as

$$\dot{\sqcup} : (2^{\mathrm{Fml}} \times \mathrm{Upd}) \times (2^{\mathrm{Fml}} \times \mathrm{Upd}) \to (2^{\mathrm{Fml}} \times \mathrm{Upd})$$

and is defined by the property: Let $\mathscr{U}_1$ and $\mathscr{U}_2$ be arbitrary updates in a proof P and let C_1, C_2 be formula sets representing constraints on the update values. Then for $(C, \mathscr{U}) = (C_1, \mathscr{U}_1) \dot{\sqcup} (C_2, \mathscr{U}_2)$ the following holds for $i \in \{1,2\}$:

1. $\mathscr{U}$ is (P, C_i)-weaker than $\mathscr{U}_i$, and
2. $C_i \Longrightarrow \{\mathscr{U}_i\} \bigwedge C$

A concrete implementation $\sqcup_{abs}$ of $\dot{\sqcup}$ for values can be computed as follows: For each update $\mathtt{x} := v$ in $\mathscr{U}_1$ or $\mathscr{U}_2$ the generated update is $\mathtt{x} := v$, if $\{\mathscr{U}_1\}x \doteq \{\mathscr{U}_2\}x$. Otherwise it is $\mathtt{x} := \gamma_{\alpha_i,j}$ for some α_i where $\chi_{\alpha_i}(\{\mathscr{U}_1\}x)$ and $\chi_{\alpha_i}(\{\mathscr{U}_2\}x)$.

Example 6.6. We illustrate the described algorithm along the previous example: The sequent to prove was

$$\mathtt{i} \geq 0 \Longrightarrow \{\mathtt{n} := 0\}\ [\mathtt{while\ (i>0)\ \{i{-}{-};\ n{+}{+};\}}](\mathtt{i} \doteq 0 \wedge \mathtt{n} \geq 0)$$

instead of 'guessing' the correct update, we sketch now how to find it automatically: Unrolling the loop once ends in two branches: one where the loop guard does not

hold and the loop is exited (which we can ignore) and the second one where the loop body is executed. After finishing the symbolic execution of the loop body the sequent is

$$\texttt{i} > 0 \Longrightarrow \{\texttt{i} := \texttt{i} - 1 \,||\, \texttt{n} := 1\}\, [\texttt{while (i>0) \{i--; n++;\}}](\texttt{i} \doteq 0 \wedge \texttt{n} \geq 0) \ .$$

We now compare the two sequents and observe that `i` and `n` has changed. Finding the minimal abstract element for `n` which covers both values 0 and 1 returns the abstract element $\geq$. For `i` we know that the previous value was greater-or-equal than the 0, after this iteration we know it has been decreased by one, but we also learned from the loop guard that on this branch the initial value of `i` was actually strictly greater than 0, hence, $\texttt{i} - 1$ is at least 0 and thus the abstract element covering both values is also $\geq$. We continue with the sequent

$$\texttt{i} > 0 \Longrightarrow \{\texttt{i} := \gamma_{\geq,1} \,||\, \texttt{n} := \gamma_{\geq,2}\}\, [\texttt{while (i>0) \{i--; n++;\}}](\texttt{i} \doteq 0 \wedge \texttt{n} \geq 0)$$

where the update has been replaced by the 'abstracted' one. The result of unrolling the loop once more results in the sequent

$$\gamma_{\geq,1} > 0 \Longrightarrow \{\texttt{i} := \gamma_{\geq,1} - 1 \,||\, \texttt{n} := \gamma_{\geq,2} + 1\}\, [\texttt{while (i>0) \{...\}}](\texttt{i} \doteq 0 \wedge \texttt{n} \geq 0) \ .$$

Joining this update with the previous one results in the update $\texttt{i} := \gamma_{\geq,4} \,||\, \texttt{n} := \gamma_{\geq,4}$ which is (except for the numbering) identical to the previous one. This means we have reached a fixed point and we can use this update as the anonymizing update.

The approach of finding the update is sound, but we want to stress that this is actually not essential as the invariantUpdate rule checks the soundness of the provided updated.

6.4 Abstract Domains for Heaps

In KeY the program heap is modeled by the program variable `heap` of type *Heap*. Therefore any changes to the program heap will be expressed as an update to the program variable `heap`. Furthermore, the *program rules*, i.e., those calculus rules for dealing with program fragments, can only ever modify `heap` by *extension*. By this we mean that given an initial update $\texttt{heap} := h$ the application of a program rule can produce an update $\texttt{heap} := h'$ only if h is a subterm of h' and h' extends h only with *anon*, *create*, and/or *store* operations. *Heap simplification rules*, however, may reduce the heap term.

Our intentions for abstraction are to join multiple program states *originating from a single source program state*, such as the program state before execution of a loop, method call, `if`- or `switch`-statement. Therefore we can assume an initial value *old* for the program heap at the originating program point and based on this create the abstract domain as follows:

We define $LS \subset D^{LocSet}$ to contain all object/field pairs not containing the *created* field, i.e.:

$$LS = D^{Object} \times (D^{Field} \setminus \{created^{\mathscr{M}}\})$$

We define the family of abstract domains $\mathscr{A}^{Heap}_{old} = (A^{Heap}_{old}, \sqsubseteq_{old}, \sqcup_{old})$ for all initial well-formed heaps *old* as

$$A^{Heap}_{old} = \{\bot, \top\} \cup 2^{LS}$$

$$x \sqcup_{old} y = \begin{cases} x & \text{, if } y = \bot \\ y & \text{, if } x = \bot \\ \top & \text{, if } y = \top \\ \top & \text{, if } x = \top \\ x \cup y & \text{, otherwise} \end{cases}$$

$$x \sqsubseteq_{old} y = (y = x \sqcup_{old} y)$$

Abstraction and concretization functions are given as:

$$\alpha_{old} : 2^{D^{Heap}} \to A^{Heap}_{old}$$

$$heaps \mapsto \begin{cases} \bot & \text{, if } heaps = \emptyset \\ a & \text{, if } \forall h \in heaps, o \in D^{Object}. \\ & wellFormed^{\mathscr{M}}(h) \wedge (old(o, created^{\mathscr{M}}) \to h(o, created^{\mathscr{M}})) \\ \top & \text{, otherwise} \end{cases}$$

$$\text{where } a = \{(o,f) \in D^{LS} \mid \exists h \in heaps.\ h(o,f) \neq old(o,f)\}$$

$$\gamma_{old} : A^{Heap}_{old} \to 2^{D^{Heap}}$$

$$\bot \mapsto \emptyset$$

$$\top \mapsto D^{Heap}$$

$$ls \subseteq LS \mapsto \{h \mid (\forall o \in D^{Object}.\ old(o, created^{\mathscr{M}}) \to h(o, created^{\mathscr{M}})) \wedge (\forall (o,f) \in D^{LS \setminus ls}.\ h(o,f) = old(o,f)\}$$

As $\mathscr{A}^{Heap}_{old}$ contains infinite ascending chains due to both *Object* and *Field* being infinite, we require either a weakening or a subset of A^{Heap}_{old} for which no infinite ascending chains exist. We could, for example, reduce the set of available location sets from *LS* to the subset thereof for which no location set contains more than n elements, for some $n \in \mathbb{N}$. Anytime a larger location set were required, we would instead return $\top$. This works, but has the distinct disadvantage that the following infinite ascending chain $\langle x_i \rangle$ will wind up overapproximating not only for the cause of the infinite ascension (the field f), but also for all other fields as well:

$$x_0 = \emptyset \tag{6.1}$$
$$x_{i+1} = x_i \cup \{(o_i, f)\} \tag{6.2}$$

In order to keep the overapproximation as localized as possible, we consider the following points:

- *Field* can be separated into array indices $Arr = \{arr^{\mathscr{M}}(x) \mid x \in \mathbb{N}\}$ and non array indices $Field \setminus Arr$.
- For any Java program there is a finite subset $fs \subset (Field \setminus Arr)$ in a closed world determinable a priori which contains all non array index fields modifiable by the program.

We therefore introduce the family of abstract domains $\mathscr{A}^{Heap}_{old,fs,n,m,k}$ for all finite sets $fs \subset (Field \setminus Arr)$ and integers $n,m,k \in \mathbb{N}$, which contain no infinite ascending chains:

$$\mathscr{A}^{Heap}_{old,fs,n,m,k} = (A^{Heap}_{old,fs,n,m,k}, \sqsubseteq_{old,fs,n,m,k}, \sqcup_{old,fs,n,m,k})$$
$$A^{Heap}_{old,fs,n,m,k} = \{\bot, \top\} \cup \{W_{fs,n,m,k}(ls) \mid ls \subseteq LS\}$$
$$x \sqcup_{old,fs,n,m,k} y = \begin{cases} \top & \text{, if } x = \top \text{ or } y = \top \\ x & \text{, if } y = \bot \\ y & \text{, if } x = \bot \\ W_{fs,n,m,k}(x \cup y) & \text{, otherwise} \end{cases}$$
$$x \sqsubseteq_{old,fs,n,m,k} y = (y = x \sqcup_{old,fs,n,m,k} y)$$

where $W_{fs,n,m,k} : 2^{LS} \to 2^{LS}$ is defined as:

$$ls \mapsto \begin{cases} LS & \text{, if } \exists (o', f') \in ls.\ f' \notin (fs \cup Arr) \\ ls \cup W^N_{fs,n}(ls) \cup W^M_m(ls) \cup W^K_k(ls) & \text{, otherwise} \end{cases}$$

with $W^N_{fs,n}, W^M_m, W^K_k$ defined as:

$$W^N_{fs,n}(ls) = \{(o,f) \mid f \in fs \wedge |\{o' \mid (o',f) \in ls\}| > n\}$$
$$W^M_m(ls) = \{(o,f) \mid f \in Arr \wedge |\{f' \in Arr \mid (o,f') \in ls\}| > m\}$$
$$W^K_k(ls) = \begin{cases} D^{Object} \times Arr & \text{, if } |\{o \mid \exists f \in Arr.\ (o,f) \in ls\}| > k \\ \emptyset & \text{, otherwise} \end{cases}$$

The function $W_{fs,n,m,k}$ is the identity on any location set which:

- contains only pairs (o,f) where f is in fs or is an array index,
- contains no more than n pairs (o,f) for any fixed $f \in fs$,
- contains no more than m pairs $(o, arr^{\mathscr{M}}(x))$ for any fixed $o \in D^{Object}$, and
- contains pairs $(o, arr^{\mathscr{M}}(x))$ for no more than k different objects $o \in D^{Object}$.

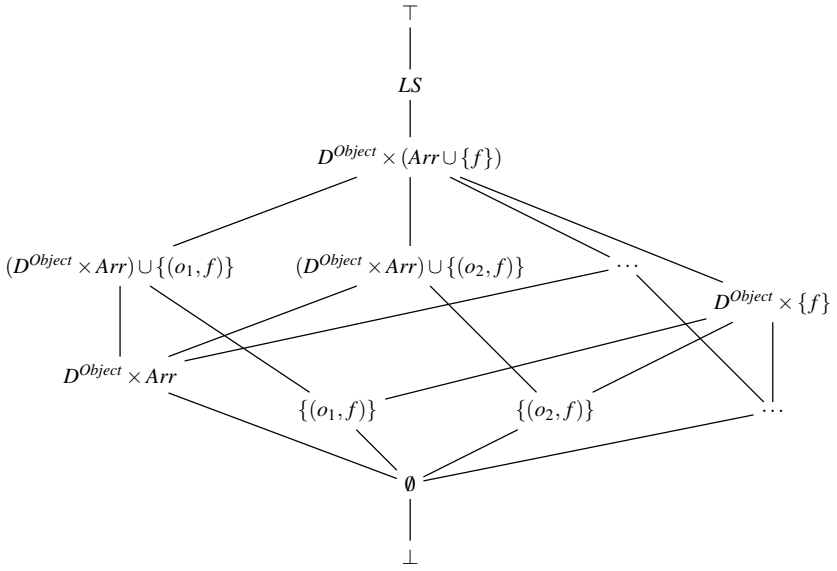


Figure 6.3 Abstract Domain $\mathscr{A}^{Heap}_{\mathtt{heap},\{f\},1,0,0}$

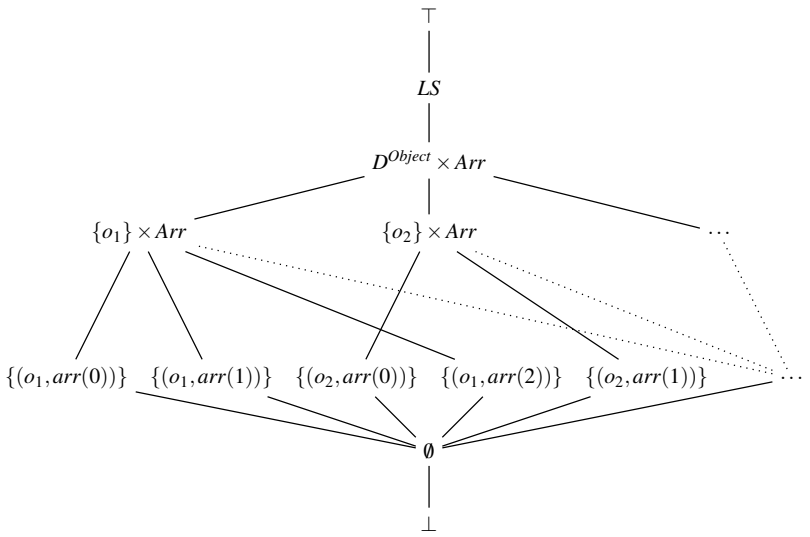


Figure 6.4 Abstract Domain $\mathscr{A}^{Heap}_{\mathtt{heap},\emptyset,0,1,1}$

For location sets outside of this scope, $W_{fs,n,m,k}$ extends the set by completion of those pairs which violate the above rules, i.e.:

- Full extension to *LS* for any location set containing a field not in *fs* or *Arr*.
- Inclusion of all pairs (o, f) for each $f \in fs$ which had more than n occurrences.
- Inclusion of all pairs $(o, arr^{\mathscr{M}}(x))$ for each $o \in D^{Object}$ which had more than m index references.
- Inclusion of all pairs $(o, arr^{\mathscr{M}}(x))$ for all objects and indices if there were more than k different objects containing index references.

The above treatment limits overapproximation, while still ensuring that no infinite ascending chains are possible.

Example 6.7 (Two Small Heap Abstractions). To demonstrate these finite height abstract domains for heaps, we look at the two heap abstractions $\mathscr{A}^{Heap}_{\texttt{heap},\{f\},1,0,0}$ and $\mathscr{A}^{Heap}_{\texttt{heap},\emptyset,0,1,1}$ in Figure 6.3 and Figure 6.4. Here the nodes marked as "..." represent an infinite number of nodes, full lines from or to such infinite nodes represent edges from or to each actual node, while dotted lines from such nodes represent edges from an infinite subset of these nodes to the corresponding connecting node. In general, of course, there will be many more available fields in *fs*, as well as n, m and k being greater than 1.

6.5 Abstract Domains for Objects

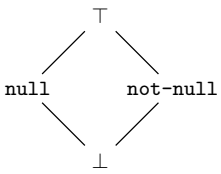
In addition to an abstraction for the `heap` variable, there also exist local variables for objects which must be abstracted as well, therefore we require an abstract domain for D^{Object}. Most of the information about an object is actually only representable if the heap on which the object resides is also known. From an abstract domain point of view this would require a relational abstract domain linking objects and heaps. As our approach does not use relational abstract domains (at least not directly), our abstract domain for objects can only express the knowledge directly obtainable from only the object itself. The following attributes of an object can be obtained without knowledge of the heap:

- Reference equality of this object with any other object, in particular `null`.
- The `length` of this object (used only by arrays).
- The type of this object.

We therefore first introduce abstract domains for objects based on each of these points separately and can then combine them into one abstract domain for objects $\mathscr{A}^{Object}$.

6.5.1 Null/Not-null Abstract Domain

The abstract domain $\mathscr{A}^{Object}_{\texttt{null}}$ for objects based on reference equality to `null` is quite simple and at the same time incredibly useful, in that it can be used to check for possible `NullPointerExceptions` or prove the lack thereof in a piece of Java code. $\mathscr{A}^{Object}_{\texttt{null}}$ is shown in Figure 6.5 with abstraction and concretization functions.



$$\alpha_{\mathscr{A}^{Object}_{\texttt{null}}}(X) = \begin{cases} \bot & \text{, if } X = \emptyset \\ \texttt{null} & \text{, if } X = \{\texttt{null}^{\mathscr{M}}\} \\ \texttt{not-null} & \text{, if } \texttt{null}^{\mathscr{M}} \notin X \\ \top & \text{, otherwise} \end{cases}$$

$$\gamma_{\mathscr{A}^{Object}_{\texttt{null}}}(x) = \begin{cases} \emptyset & \text{, if } x = \bot \\ \{null\} & \text{, if } x = \texttt{null} \\ D^{Object} \setminus \{null\} & \text{, if } x = \texttt{not-null} \\ D^{Object} & \text{, if } x = \top \end{cases}$$

Figure 6.5 Abstract Domain $\mathscr{A}^{Object}_{\texttt{null}}$

6.5.2 Length Abstract Domain

An abstract domain for objects based on their *length* is useful only for arrays. For all other object types the length is some arbitrary number which has no meaning. For arrays, however, abstracting these to their length can be quite helpful, for example one could conclude based on this abstraction whether a loop iterating over an array should be unrolled completely or a loop invariant generated for it.

We require an abstract domain $\mathscr{A}^{\mathbb{Z}}$ for the concrete domain $\mathbb{Z}$ and map each object's length to said abstract domain. We can then define the abstract domain for objects based on their *length* as $\mathscr{A}^{Object}_{\texttt{length}} := \mathscr{A}^{\mathbb{Z}}$ with abstraction and concretization functions as follows:

$$\alpha_{\mathscr{A}^{Object}_{\texttt{length}}}(X) = \alpha_{\mathscr{A}^{\mathbb{Z}}}(\{length^{\mathscr{M}}(x) \mid x \in X\})$$
$$\gamma_{\mathscr{A}^{Object}_{\texttt{length}}}(x) = \{o \in D^{Object} \mid length^{\mathscr{M}}(o) \in \gamma_{\mathscr{A}^{\mathbb{Z}}}(x)\}$$

We can use any abstract domain for $\mathbb{Z}$, for example the simple sign domain in Figure 6.2. However, using this abstract domain would not be very clever as the abstract elements neg and $\leq$ will never abstract valid array lengths, while the abstraction of both 1 and 10000 to the same abstract element pos is not very helpful. Instead, let us consider the following:

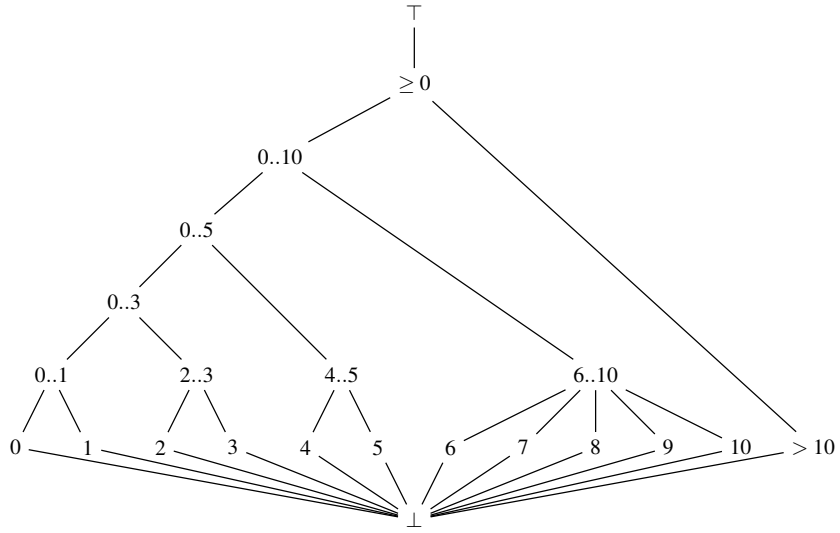


Figure 6.6 An Abstract Domain $\mathscr{A}^{Object}_{\texttt{length}}$

1. Iterating over an array of length 0 is trivial (do not enter loop) and therefore full precision should be kept, rather than abstracting by applying a loop invariant.
2. Iterating over an array of length 1 is similarly trivial (execute the loop body once) and therefore full precision should also be kept here by unrolling the loop, rather than applying a loop invariant. The loop invariant rule must still prove that the loop body preserves the invariant, thus execution of the loop body is always required once, even when applying a loop invariant.
3. Iterating over an array of length 2 or 3 can usually be done reasonably quickly by unrolling the loop a sufficient number of times, therefore unrolling should be favored over applying a loop invariant except in cases where symbolic execution of the loop body is extremely costly.
4. Iterating over an array of length 4 or 5 can often be done reasonably quickly by loop unrolling, therefore applying a loop invariant should only be done for somewhat complex loop bodies.
5. Iterating over an array of length 6 to 10, applying a loop invariant should be favored, except in cases where the loop body is trivial.
6. Iterating over an array of length greater than 10 should almost always be solved by applying a loop invariant.

The above are reasonable guidelines (or in the case of lengths 0 and 1 simple fact), such that we can present the abstract domain in Figure 6.6 for the concrete domain $\mathbb{Z}$ and therefore also for objects based on their length.

6.5.2.1 Type Abstract Domain

Abstracting on object type requires knowledge of the type hierarchy. However, due to logical consequence of a formula requiring that the formula hold in all extensions of the type hierarchy, we must in essence create an abstract domain based on not just the type hierarchy given directly by the program, but any extension thereof.

For a set of objects X we offer abstractions for their types based on which exact types are present in X, i.e., a set of types such that each element in X is an exact instance of one of those types.

For any given type hierarchy $\mathscr{T}$ we must create an abstract domain, such that there exist abstraction and concretization functions for all type hierarchies $\mathscr{T}'$ which extend $\mathscr{T}$.

For a given type hierarchy $\mathscr{T} = (TSym, \sqsubseteq)$ we first define the set of all dynamic object types $O_d = \{d \in TSym \mid d \sqsubseteq Object \text{ and } d \text{ is not marked abstract}\}$ and based on this define the abstract domain $\mathscr{A}_{O_d}^{Object}$, as shown in Figure 6.7. Then for any type hierarchy extension $\mathscr{T}' = (TSym', \sqsubseteq')$ of $\mathscr{T}$ the abstraction and concretization functions are given in Figure 6.8.

$$
\begin{aligned}
\mathscr{A}_{O_d}^{Object} &= (A_{O_d}^{Object}, \sqsubseteq_{O_d}^{Object}, \sqcup_{O_d}^{Object}) \\
A_{O_d}^{Object} &= \{\top\} \cup (2^{O_d} \setminus O_d) \\
X \sqcup_{O_d}^{Object} Y &= \begin{cases} \top & \text{, if } X = \top \text{ or } Y = \top \text{ or } X \cup Y = O_d \\ X \cup Y & \text{, otherwise} \end{cases} \\
X \sqsubseteq_{\mathscr{T}}^{Object} Y &= \begin{cases} tt & \text{, if } Y = \top \\ ff & \text{, if } X = \top \text{ and } Y \neq \top \\ X \subseteq Y & \text{, otherwise} \end{cases}
\end{aligned}
$$

Figure 6.7 Family of Abstract Domains $\mathscr{A}_{O_d}^{Object}$

$$
\begin{aligned}
\alpha_{O_d,\mathscr{T}'}^{Object}(X) &= \begin{cases} \top & \text{, if } \exists x \in X.\ \delta'(x) \notin O_d \\ & \text{ or } \{d \in T_d \mid \exists x \in X.\ \delta'(x) = d\} = O_d \\ \{d \in T_d \mid \exists x \in X.\ \delta'(x) = d\} & \text{, otherwise} \end{cases} \\
\gamma_{O_d,\mathscr{T}'}^{Object}(X) &= \begin{cases} \{o \in D' \mid \delta'(o) \sqsubseteq' Object\} & \text{, if } X = \top \\ \{o \in D' \mid \bigvee_{d \in X} \delta'(o) = d\} & \text{, otherwise} \end{cases}
\end{aligned}
$$

Figure 6.8 Galois connection between $\mathscr{A}_{O_d}^{Object}$ and the Concrete Object Domain from $\mathscr{T}'$

Example 6.8. We can consider a simplified Java program containing only the types declared in Listing 6.1. Based on this we have the set of concrete object types

```
class Object {...}
abstract class A {...}
interface I {...}
class B extends A implements I {...}
```

Listing 6.1 Type declarations

$\{Object, B, Null\}$ and the abstract domain $\mathscr{A}^{Object}_{\{Object,B,Null\}}$ as shown in Figure 6.9.

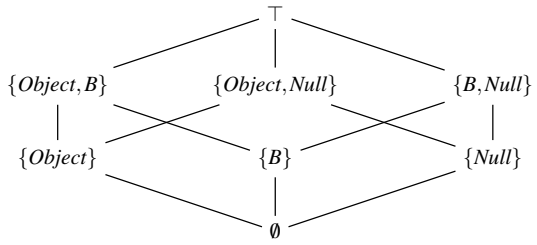


Figure 6.9 Abstract Domain $\mathscr{A}^{Object}_{\{Object,B,Null\}}$

The abstract domains $\mathscr{A}^{Object}_{O_d}$ can be used, for example, to:

- prove that casting of an object does not cause a `ClassCastException` to be thrown,
- prove that no `ArrayStoreException` is thrown when inserting an object into an array,
- prove that an `instanceof` check will be successful,
- prove that an `instanceof` check will be unsuccessful, and/or
- narrow the list of possible method body instantiations down which is created when unfolding a method call.

It is important to point out that although $\mathscr{A}^{Object}_{O_d}$ always has abstract elements $\{Null\}$ and $O_d \setminus \{Null\}$, it is not inherently stronger than the abstract domain $\mathscr{A}^{Object}_{\texttt{null}}$. This is because given a type hierarchy extension $\mathscr{T}'$ which introduces a new dynamic type $d' \sqsubset Object$, for which it holds for some o that $\delta'(o) = d'$ the following abstractions exist:

$$\alpha^{Object}_{O_d,\mathscr{T}'}(\{o\}) = \top$$
$$\alpha^{Object}_{\texttt{null}}(\{o\}) = \texttt{not-null}$$

6.5.2.2 Combining the Object Abstract Domains Into One

Of course, we would like just one abstract domain for objects encompassing all of the abstractions discussed in the previous subsections. The abstract domain $\mathscr{A}^{Object}$ for a given type hierarchy $\mathscr{T}$ is a partial cartesian product of the abstract domains $\mathscr{A}^{Object}_{\texttt{null}}$, $\mathscr{A}^{Object}_{\texttt{length}}$ and $\mathscr{A}^{Object}_{O_d}$ such that the abstraction and concretization functions for any type hierarchy extension $\mathscr{T}'$ can be given as in Figure 6.10. The reason why only a subset of the cartesian product is required is due to the following: As it must hold that $\alpha(\gamma(a)) = a$ for all abstract elements a, there can never be more than one abstract element representing the same set. We therefore cannot have both $(\bot, y, z)$ and $(x, \bot, z)$ as separate abstract elements, as intuitively both of these would have to represent the empty set. Additionally, while the abstraction for `length` is orthogonal to the abstractions for `null` and exact type (due to the function *length* being defined for all objects, including `null` and nonarray types), the abstractions for `null` and exact type are not. While it is true that in one abstraction we may know that `null` does not appear, while in the other abstraction we do not, it is nonetheless impossible for certain abstract elements to be combined without representing the empty set, for example the abstract elements `null` and $\{Object\}$.

The abstract domain $\mathscr{A}^{Object}$ is defined in Figure 6.11.

$$\alpha^{Object}(X) = (\alpha^{Object}_{\texttt{null}}(X), \alpha^{Object}_{\texttt{length}}(X), \alpha^{Object}_{O_d,\mathscr{T}'}(X))$$
$$\gamma^{Object}((a,b,c)) = \gamma^{Object}_{\texttt{null}}(a) \cap \gamma^{Object}_{\texttt{length}}(b) \cap \gamma^{Object}_{O_d,\mathscr{T}'}(c)$$

Figure 6.10 Abstraction and concretization Functions between $\mathscr{A}^{Object}$ and concrete objects in $\mathscr{T}'$

$$\mathscr{A}^{Object} = (A^{Object}, \sqsubseteq^{Object}, \sqcup^{Object})$$
$$A^{Object} \subset A^{Object}_{\texttt{null}} \times A^{Object}_{\texttt{length}} \times A^{Object}_{O_d}$$
$$(a,b,c) \sqsubseteq^{Object} (x,y,z) = a \sqsubseteq^{Object}_{\texttt{null}} x \wedge b \sqsubseteq^{Object}_{\texttt{length}} y \wedge c \sqsubseteq^{Object}_{O_d} z$$
$$(a,b,c) \sqcup^{Object} (x,y,z) = (a \sqcup^{Object}_{\texttt{null}} x, b \sqcup^{Object}_{\texttt{length}} y, c \sqcup^{Object}_{O_d} z)$$

Figure 6.11 Abstract Domain $\mathscr{A}^{Object}$

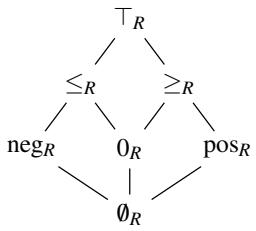
6.6 Extensions

In this section we briefly sketch how to add additional precision for arrays while staying fully automatic. For sake of presentation we use a simplified abstract domain for arrays (but which is included in the abstraction given in Section 6.4) and define a more specific notion to join heap values. Based on this we can then sketch how to automatically generate loop invariants for arrays that maintain a reasonable level of precision for many use cases. This section is basically a shortened version of [Hähnle et al., 2016] to which we refer the reader for details.

6.6.1 Abstractions for Arrays

We extend the abstract domain of the array elements to a range within the array. Given a set of indexes R, an abstract domain A for array elements can be extended to an abstract domain A_R for arrays by copying the structure of A and renaming each α_i to $\alpha_{R,i}$. The $\alpha_{R,i}$ are such that $\gamma_{\alpha_{R,i},j} \in \{arrObj \in \texttt{int[]} \mid \forall k \in R.\chi_{\alpha_i}(arrObj[k])\}$.

Example 6.9. As abstract domain A we use the sign domain for integers, producing for each $R \subseteq \mathbb{N}$ an abstract domain A_R:



$$\begin{aligned}
\gamma(\top_R) &= \texttt{int[]} \\
\gamma(\leq_R) &= \{arrObj \in \texttt{int[]} \mid \forall k \in R.arrObj[k] \leq 0\} \\
\gamma(\geq_R) &= \{arrObj \in \texttt{int[]} \mid \forall k \in R.arrObj[k] \geq 0\} \\
\gamma(\text{neg}_R) &= \{arrObj \in \texttt{int[]} \mid \forall k \in R.arrObj[k] < 0\} \\
\gamma(\text{pos}_R) &= \{arrObj \in \texttt{int[]} \mid \forall k \in R.arrObj[k] > 0\} \\
\gamma(0_R) &= \{arrObj \in \texttt{int[]} \mid \forall k \in R.arrObj[k] \doteq 0\} \\
\gamma(\emptyset_R) &= \{\}
\end{aligned}$$

With $R = \{0,2\}$, we get $\gamma(\geq_{\{0,2\}}) = \{arrObj \in \texttt{int[]} \mid arrObj[0] \geq 0 \wedge arrObj[2] \geq 0\}$. Importantly, the array length itself is irrelevant, provided $arrObj[0]$ and $arrObj[2]$ have the required values. Therefore the arrays (we deviate from Java's array literal syntax for clarity) $[0,3,6,9]$ and $[5,-5,0]$ are both elements of $\gamma(\geq_{\{0,2\}})$.

Of particular interest are the ranges containing all elements modified within a loop. One such range is $[0..arrObj.\texttt{length})$. This range can always be taken as a fallback option if no more precise range can be found.

6.6.2 Loop Invariant Rule with Value and Array Abstraction

To be able to deal with arrays we extend the updateInvariant rule:

invariantUpdate

$$\frac{\begin{array}{l}\Gamma, \mathscr{U}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \mathtt{old} \doteq \mathscr{U}\,\mathtt{heap} \Longrightarrow \mathscr{U}\,Inv, \Delta \\ \Gamma, \mathtt{old} \doteq \mathscr{U}\,\mathtt{heap},\ \mathscr{U}'_{mod}(g \wedge Inv),\ \mathscr{U}'_{mod}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \\ \qquad\qquad \exists\ \bar{\gamma}; \mathscr{U}'_{mod}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \mathtt{old} \doteq \mathscr{U}\,\mathtt{heap},\ \mathscr{U}'_{mod}(g \wedge Inv) \Longrightarrow \mathscr{U}'_{mod}[\mathtt{p}]Inv, \Delta \\ \Gamma, \mathtt{old} \doteq \mathscr{U}\,\mathtt{heap},\ \mathscr{U}'_{mod}(\neg g \wedge Inv) \Longrightarrow \mathscr{U}'_{mod}[r]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \mathscr{U}[\mathtt{while\ (}g\mathtt{)\ \{}p\mathtt{\};}\ r]\varphi, \Delta}$$

where $\mathscr{U}'_{mod} := (\mathscr{U}' \parallel \mathscr{V}^{heap}_{mod})$ with $\mathscr{U}'$ being the $\mathscr{U}'_{mod}$ from the previous sections and $\mathscr{V}^{heap}_{mod}$ denotes the abstraction of the heap stored in program variable `heap`. The $\bar{\mathtt{x}}, \bar{c}, \bar{\gamma}$ and $\exists\ \bar{\gamma}; \varphi$ are defined as previously. In addition to heap abstraction, we reintroduce the loop invariant formula *Inv*, which is subsequently used to express properties about the content of the heap. This includes explicit heap invariants of the form $\forall i \in S.\ C \rightarrow P(select_{int}(\mathtt{heap}, \mathtt{arrObj}, arr(i)))$ as well as invariants which further specify S or C. The program variable and `old` is a fresh constant used in *Inv* to refer to the heap before loop execution.

```
i = 0; j = 0;
while(i < a.length) {
  if (a[j] > 0) j++;
  b[i] = j;
  c[2*i] = 0;
  i++;
}
```

Listing 6.2 Example program for array abstraction

Most branches serve a similar approach as those in the previous version. The second and the third branch are new ensuring that the loop invariant formula is initially valid as well as preserved by the loop body. Given program `p` in Listing 6.2, applying the assignment rule to $\Gamma \Longrightarrow \mathscr{U}[\mathtt{p}]\varphi, \Delta$ leads to $\Gamma \Longrightarrow \{\mathscr{U} \parallel \mathtt{i} := 0 \parallel \mathtt{j} := 0\}[\mathtt{while...}]\varphi, \Delta$. Now the `invariantUpdate` rule is applied with, e.g., the following values:

$$\mathcal{U}' = (\mathcal{U} \parallel \mathtt{i} := \gamma_{\geq,1} \parallel \mathtt{j} := \gamma_{\geq,2})$$

$$\mathcal{V}^{heap}_{mod} =$$
$$\quad \mathtt{heap} := anon(anon(\mathtt{heap}, \mathtt{b}[0..i], anonHeap_1), \mathtt{c}[0..\mathtt{c.length}], anonHeap_2)$$

$$Inv = \big(\forall k \in [0..\mathtt{j});\ \chi_{>}(select_{int}(\mathtt{heap}, \mathtt{a}, arr(k)))\big)$$
$$\quad \wedge \big(\forall k \in [0..\mathtt{i});\ \chi_{\geq}(select_{int}(\mathtt{heap}, \mathtt{b}, k))\big)$$
$$\quad \wedge (\forall m \in [0..\mathtt{c.length});$$
$$\qquad (m < 2*i \wedge m\%2 \doteq 0) \rightarrow \chi_0(select_{int}(\mathtt{heap}, \mathtt{c}, arr(2*m))))$$
$$\quad \wedge \big(\forall m \in [0..\mathtt{c.length}); \neg(m < 2*i \wedge m\%2 \doteq 0)$$
$$\qquad \rightarrow (select_{int}(\mathtt{heap}, \mathtt{c}, arr(m)) \doteq select_{int}(\mathtt{old}, \mathtt{c}, arr(m)))\big)$$

The update $\mathcal{U}'_{mod}$ is equal to the original update $\mathcal{U}$ except for the values of `i` and `j` which can both be any nonnegative number. The arrays `b` and `c` have (partial) ranges anonymized. We use $arrObj[lower..upper]$ to express the set of locations consisting of all array elements of array $arrObj$ from $lower$ (included) to $upper$ (excluded).

Array `a` is not changed by the loop and thus not anonymized. The invariants in Inv express that

1. `a` contains positive values at all positions prior to the current value of `j`,
2. the anonymized values in `b` (cf. $\mathcal{V}^{heap}_{mod}$) are all nonnegative, and
3. the anonymized values in `c` are equal to their original values (if the loop does not or has not yet modified them) or are equal to 0.

6.6.3 Computation of the Abstract Update and Invariants

We generate $\mathcal{U}'$, $\mathcal{V}^{heap}_{mod}$ and Inv automatically in a side proof, by symbolic execution of single loop iterations until a fixed-point is found. The computation of $\mathcal{U}'$ is as in Section 6.3, but ignores the heap variable `heap`. We generate $\mathcal{V}^{heap}_{mod}$ and Inv by examining each array modification[1] and anonymizing the entire range within the array (expressed in $\mathcal{V}^{heap}_{mod}$) while adding a partial invariant to the set Inv. Once a fixed-point for $\mathcal{U}'$ is reached, we can refine $\mathcal{V}^{heap}_{mod}$ and Inv by performing in essence a second fixed-point iteration, this time anonymizing possibly smaller ranges and potentially adding more invariants.

To perform this we need to generalize our notion of joining updates to include heaps.

Definition 6.10 (Joining Heaps). Any operator with the signature

$$\hat{\sqcup} : (2^{\text{Fml}} \times \text{DLTrm}_{Heap}) \times (2^{\text{Fml}} \times \text{DLTrm}_{Heap}) \rightarrow (2^{\text{Fml}} \times \text{DLTrm}_{Heap})$$

[1] Later we also examine each array access (read or write) in `if`-conditions to gain invariants such as $\forall k \in [0..\mathtt{j}).\ \chi_{>}(\mathtt{select}(\mathtt{heap}, \mathtt{a}, arr(k)))$ in the example above.

is a *heap join operator* if it satisfies the properties: Let h_1, h_2 be arbitrary heaps in a proof P, C_1, C_2 be formula sets representing constraints on the heaps (and possibly also on other update values) and let $\mathscr{U}$ be an arbitrary update. Then for $(C,h) = (C_1,h_1) \hat{\sqcup} (C_2,h_2)$ the following holds for $i \in \{1,2\}$:

1. $(\mathscr{U} \parallel \texttt{heap} := h)$ is (P, C_i)-weaker than $(\mathscr{U} \parallel \texttt{heap} := h_i)$,
2. $C_i \Longrightarrow \{\mathscr{U} \parallel \texttt{heap} := h_i\} \bigwedge C$, and
3. $\hat{\sqcup}$ is associative and commutative up to first-order reasoning.

We define the set of *normal form heaps* $Heap_{NF} \subset \mathrm{DLTrm}_{\texttt{heap}}$ to be those heap terms that extend `heap` with an arbitrary number of preceding stores or anonymizations. For a heap term $h \in Heap_{NF}$ we define

$$writes(h) := \begin{cases} \emptyset & \text{if } h = \texttt{heap} \\ \{h\} \cup writes(h') & \text{if } h = store(h', a, arr(idx), v) \text{ or} \\ & \quad h = anon(h', a[l..r], h'') \end{cases}$$

A concrete implementation $\hat{\sqcup}_{heap}$ of $\hat{\sqcup}$ is given as follows: We reduce the signature to $\hat{\sqcup}_{heap} : (2^{\mathrm{Fml}} \times Heap_{NF}) \times (2^{\mathrm{Fml}} \times Heap_{NF}) \to (2^{\mathrm{Fml}} \times Heap_{NF})$. This ensures that all heaps we examine are based on `heap` and is a valid assumption when taking the program rules into account, as these maintain this normal form. As both heaps are in normal form, they must share a common subheap (at least `heap`). The largest common subheap of h_1, h_2 is defined as $lcs(h_1, h_2)$ and all writes performed on this subheap can be given as $writes_{lcs}(h_1,h_2) := writes(h_1) \cup writes(h_2) \setminus (writes(h_1) \cap writes(h_2))$. For the interested reader, the actual algorithms to compute the update abstractions are shown in [Hähnle et al., 2016].

6.6.4 Symbolic Pivots

Finally, we sketch briefly how to generate the loop invariant formula *Inv* capturing knowledge about the modified content of an array. In the previous section we computed the update $\mathscr{U}_{mod}$, which provides us the abstraction for primitive types as well as the heap, in particular, arrays. For the latter this information is relatively weak as it assumes any update to an array element could cause a change at any array index. With the generated $\mathscr{U}'$, however, we can now refine $\mathscr{V}_{mod}^{heap}$ and *Inv*. We try to keep the anonymizations in $\mathscr{V}_{mod}^{heap}$ to a minimum, while producing stronger invariants *Inv*.

Consider the starting sequent

$$\Gamma \Longrightarrow \mathscr{U}[\texttt{while (}g\texttt{) \{}p\texttt{\}; }r]\varphi, \Delta \ .$$

As $\mathscr{U}'$ is weaker than $\mathscr{U}$, the update $(\mathscr{U}' \parallel \texttt{heap} := \mathscr{U}\texttt{heap})$ remains weaker than $\mathscr{U}$. For the sequent

$$\Gamma \Longrightarrow \{\mathscr{U}' \parallel \texttt{heap} := \mathscr{U}\texttt{heap}\}[\texttt{while (}g\texttt{) \{}p\texttt{\}; }r]\varphi, \Delta$$

while computing the heap join (by unrolling the loop) we reach open branches

$$\Gamma_i \Longrightarrow \{\mathscr{U}_i\}[\texttt{while (}g\texttt{) \{}p\texttt{\}; }r]\varphi, \Delta_i \ .$$

Aside from the values for `heap`, $\mathscr{U}'$ is weaker than $\mathscr{U}_i$, as $\mathscr{U}'$ is a fixed-point. We therefore do not have to join any nonheap variables when computing $(\mathscr{U}^*, Inv)$.

When joining constraint/heap pairs we distinguish between three types of $f_{m,n}$:

1. anonymizations, which are kept, as well as any invariants generated for them occurring in the constraints,
2. stores to concrete indexes, for which we create a store to the index either of the explicit value (if equal in both heaps) or of a fresh $\gamma_{i,j}$ of appropriate type, and
3. stores to variable indexes, for which we anonymize a (partial) range in the array and give stronger invariants.

Given a store to a variable index $store(h, a, arr(idx), v)$, the index idx is expressible as a function $index(\gamma_{i_0,j_0}, \ldots, \gamma_{i_n,j_n})$. These γ_{i_x,j_x} can be linked to program variables in the update $\mathscr{U}'$, which contains updates $\texttt{pv}_x := \gamma_{i_x,j_x}$.We can therefore express idx as the function $sp(\ldots \texttt{pv}_x \ldots)$.

We call $idx = sp(\ldots \texttt{pv}_x \ldots)$ a *symbolic pivot*, as it expresses what elements of the array can be changed based on which program variables and allows us to partition the array similar to pivot elements in array algorithms. Symbolic pivots split the array into an already modified partition and an unmodified partition, where (parts of) the unmodified partition may yet be modified in later iterations of the loop.

If $P(\mathscr{W}) = \forall k \in [\mathscr{U} sp..\mathscr{W} sp).\ \mathscr{W} \chi_{\alpha_j}(select_{int}(\texttt{heap}, arrObj, arr(k))$, for a symbolic pivot sp, $P(\mathscr{U})$ is trivially true, as we are quantifying over an empty set. Likewise, it is easy to show that the instance $Q(\mathscr{U})$ of the following is valid:

$$\begin{aligned} &Q(\mathscr{W}) = \\ &\quad \forall k \notin [\mathscr{U} sp..\mathscr{W} sp); \\ &\qquad select_{int}(\mathscr{W}\texttt{heap}, \mathscr{W} arrObj, arr(k)) \doteq \\ &\qquad\qquad select_{int}(\mathscr{U}\texttt{heap}, \mathscr{W} arrObj, arr(k)) \end{aligned}$$

Therefore, anonymizing an array $arrObj$ with

$$anon(h, arrObj[0..arrObj.\texttt{length}], anonHeap)$$

and adding invariants $P(\mathscr{U}^*)$ and $Q(\mathscr{U}^*)$ for the contiguous range $[\mathscr{U} sp..\{\mathscr{U}^*\}sp)$ is inductively sound, if $P(\mathscr{U}') \Longrightarrow P(\mathscr{U}_i)$ and $Q(\mathscr{U}') \Longrightarrow Q(\mathscr{U}_i)$.

Definition 6.11 (Iteration affine). Given a sequent $\Gamma \Longrightarrow \mathscr{U}[\mathrm{p}]\varphi, \Delta$ where p starts with `while`, a term t is *iteration affine*, if there exists some $step \in \mathbb{Z}$ such that for any $n \in \mathbb{N}$, if we unroll and symbolically execute the loop n times, for each branch with sequent $\Gamma_i \Longrightarrow \mathscr{U}_i[\mathrm{p}]\varphi, \Delta_i$ it holds that there is some value v, such that $\Gamma_i \cup !\Delta_i \Longrightarrow \mathscr{U}_i t \doteq v$ and $\Gamma \cup !\Delta \Longrightarrow \mathscr{U} t + n * step \doteq v$.

If the symbolic pivot is iteration affine, we know the exact elements that may be modified. We could anonymize only this range. However, as expressing the affine

range as a location set is nontrivial, we anonymize the entire array and create the following invariants for the modified and unmodified partitions (using the symbols of Definition 6.11):

$$\forall k \in [0..arrObj.\texttt{length}).\ (k \geq \mathscr{U}\, sp \wedge k < sp \wedge (k - \mathscr{U}\, sp)\%step \doteq 0) \rightarrow P(k),$$
$$\text{and } \forall k \in [0..arrObj.\texttt{length}).\ \neg(k \geq \mathscr{U}\, sp \wedge k < sp \wedge (k - \mathscr{U}\, sp)\%step \doteq 0)$$
$$\rightarrow select_{int}(\texttt{heap}, arrObj, arr(k)) \doteq select_{int}(\mathscr{U}\,\texttt{heap}, arrObj, arr(k))$$

Finally, we can also add invariants (without anonymizations) for array accesses which influence control flow. For each open branch with a condition $C(select_{int}(h, arrObj, arr(idx)))$ not already present in the sequent leading to it, we determine the symbolic pivot for idx and create an iteration affine or contiguous invariant for it.

6.7 Conclusions

In this section we outlined how to integrate abstraction into JavaDL. We looked first into cases without a heap to explain the basic idea. We sketched then our approach to extend the approach to arrays. We explained the necessary extensions to maintain a reasonable amount of precision when abstracting arrays. The presented approach has also been used to cover method contracts and recursion [Wasser, 2015]. It has been applied in an eVoting case study [Do et al., 2016] to achieve full automation for the purpose of detecting information leaks.