Constancy Analysis

Samir Genaim and Fausto Spoto

[1] CLIP, Technical University of Madrid (UPM), Spain
[2] Università di Verona, Italy
samir@clip.dia.fi.upm.es, fausto.spoto@univr.it

Abstract. A reference variable x is constant in a piece of code C if the execution of C does not modify the heap structure reachable from x. This information lets us infer purity of method arguments, an important ingredient during the analysis of programs dealing with dynamically allocated data structures. We define here an abstract domain expressing constancy as an abstract interpretation of concrete denotations. Then we define the induced abstract denotational semantics for Java-like programs and show how constancy information improves the precision of existing static analyses such as sharing, cyclicity and path-length.

1 Introduction

A major difference between pure functional/logic programming and imperative programming is that the latter uses destructive updates. That is, data structures are *mutable*: they are built *and later modified.* This can be both recognized as a superiority of imperative programming, since it allows one to write faster and simpler code, and as a drawback, since if two variables share a data structure then a destructive update to the data reachable from one variable may affect the data reachable from the other. This often leads to subtle programming bugs.

It is hence important to control what a method invocation modifies. Some methods do not modify the data structures reachable from their parameters. Others only modify those reachable from some but not all parameters. Namely, some parameters are *constant* or *read-only*, others may be modified. If all parameters of a method are constant, the method is *pure* [10]. Knowledge about purity is important since pure methods can be invoked in any order, which lets compilers apply aggressive optimizations; pure methods can be used in program assertions [7]; they can be skipped during many static analyses or more precisely approximated than other methods. This results in more efficient and more precise analyses. For instance, sharing analysis [11] can safely assume that sharing is not introduced during the execution of a pure method. In general, all static analyses tracking properties of the heap benefit from information about purity.

For these reasons, software specification has found ways of expressing purity of methods and constant parameters. The notable example is the Java Modeling Language [7], which uses the `assignable` clause to specify those heap positions that might be mutated during the execution of a method. Those clauses are manually provided and used by many static analyzers, such as ESC/Java [6] and ChAsE [4]. However, those tools do not verify the correctness of the user-provided `assignable` clauses, or use potentially incorrect verification techniques. A formally correct verification technique is defined in [14], but has never been implemented. In [10] a formally correct analysis for purity is presented, it is based on a preliminary points-to and escape analysis, and an implementation exists and has been applied to some small size examples. In [8] a correct and precise algorithm for statically inferring the reference immutability qualifiers of the Javari language has been presented. The algorithm has been implemented in the Javarifier tool.

In this paper, we investigate an alternative technique aiming at determining which parameters of a method are constant. We use abstract interpretation [5] and perform a static analysis over the reduced product of the sharing domain in [11] (the *sharing component*) and a new abstract domain expressing the set of variables bound to data structures mutated during the execution of a piece of code (the *purity component*). The use of reduced product is justified since the sharing component helps the purity component during a destructive update, by identifying which variables share the updated data structure and hence lose their purity; conversely, the sharing component uses the purity component during method calls, since only variables sharing with non-pure parameters of a method m can be made to share during the execution of m.

Our technique is sometimes less precise than [10], since it does not use the field names (i.e., we do not keep information on which field has been updated, but rather that a field has been updated). However, it is implemented in a completely flow-sensitive and context-sensitive fashion, which improves its precision. Moreover, it is expressed in terms of Boolean formulas implemented through binary decision diagrams, resulting in fast analyses scaling to quite big programs. Our contributions are hence: (1) a definition of the reduced product of sharing and purity; (2) its application to large programs; (3) a comparison of the precision of sharing analysis alone with that of sharing analysis in reduced product with purity; and (4) an evaluation of the extra precision induced by the purity information during static analyses tracking properties of the heap, namely, possible cyclicity of data structures [9] and path-length of data structures [13].

The paper is organized as follows: Section 2 defines the syntax and semantics of a simple Object-Oriented language; Section 3 develops our constancy analysis for that language; Section 4 provides an experimental evaluation.

2 Our Simple Object-Oriented Language

This section presents syntax and denotational semantics of a simple Object-Oriented language that we use through the paper. Its commands are normalized versions of corresponding Java commands: the language supports reference and integer types; in method calls, only syntactically distinct variables can be actual parameters, which is a form of normalization and does not prevent them from being bound to shared data-structures at run-time; in assignments, the left hand side is either a variable or the field of a variable; Boolean conditions are kept generic, they are conditions that are evaluated to either $true$ or $false$; iterative constructs, such as the `while` loop, are not supported since they can be implemented through recursion. These assumptions are only for the sake of clear and simple presentation and can be relaxed without affecting subsequent results. A program has a set of *variables* $\mathcal{V}$ (including *out* and *this*) and a finite poset of *classes* $\mathbb{K}$. The *commands* of the language are

$$\begin{aligned} com ::= \; & v := c \mid v := w \mid v := \texttt{new}\ \kappa \mid v := w + z \mid v := w.\texttt{f} \mid v.\texttt{f} \ := w \mid \\ & v := v_0.\texttt{m}(v_1, \ldots, v_n) \mid \texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2 \mid com_1;com_2 \end{aligned}$$

$v, w, z, v_0, v_1, \ldots, v_n \in \mathcal{V}$ are distinct variables, $c \in \mathbb{Z} \cup \{\mathsf{null}\}$, $\kappa \in \mathbb{K}$ and e is a Boolean expression. The signature of a method $\kappa.\texttt{m}(t_1, \ldots, t_p){:}t$ refers to a method called `m` expecting p parameters of type $t_1, \ldots, t_p \in \mathbb{K} \cup \{\mathsf{int}\}$, respectively, returning a value of type t and *defined* in class κ with a statement

$$t\ \texttt{m}(w_1{:}t_1, \ldots, w_n{:}t_n)\ \texttt{with}\ \{w_{n+1}{:}t_{n+1}, \ldots, w_{n+m}{:}t_{n+m}\}\ \texttt{is}\ com,$$

where $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ are distinct, not in $\{out, this\}$ and have *type* $t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m} \in \mathbb{K} \cup \{\mathsf{int}\}$, respectively. Variables $w_1, \ldots, w_n$ are the *formal parameters* of the method and $w_{n+1}, \ldots, w_{n+m}$ are its *local variables*. The method also uses a variable *out* of type t to store its *return value*. For a given method signature $m = \kappa.\texttt{m}(t_1, \ldots, t_p) : t$, we define $m^b = com$, $m^i = \{this, w_1, \ldots, w_n\}$, $m^o = \{out\}$, $m^l = \{w_{n+1}, \ldots, w_{n+m}\}$ and $m^s = m^i \cup m^o \cup m^l$. Classes might declare fields of type $t \in \mathbb{K} \cup \{\mathsf{int}\}$.

We use a denotational semantics, hence compositional, in the style of [15]. However, we use a more complex notion of state, which assumes an infinite set of *locations*. Basically, a state is a pair which consists of a frame and a heap, where a frame maps variables to values and a heap maps locations to objects. Note that since we assume a denotational semantics, a state has a single frame, rather than an activation stack of frames as it is required in operational semantics. We let $\mathbb{L}$ denote an infinite set of *locations*, and let $\mathbb{V}$ denotes the set of *values* $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$. A *frame* over a finite set of variables V is a mapping that maps each variable in V into a value from $\mathbb{V}$; a *heap* is a partial map from $\mathbb{L}$ into *objects*. An object is a pair that consists of its class tag κ and a frame that maps its *fields* (identifiers) into values from $\mathbb{V}$; we say that it *belongs to* class κ or *has* class κ. Given a class κ, we assume that $newobj(\kappa)$ return a new object where its fields are initialized to 0 or depending on their types. If ϕ is a frame and $v \in V$, then $\phi(v)$ is the value of variable v. If μ is a heap and $\ell \in \mathbb{L}$, then $\mu(\ell)$ is the object bound in μ to ℓ. If o is an object, then $o.tag$ denotes its class and $o.\phi$ denotes its frame; if f is a field of o, then sometimes we use $o.f$ to refer to (or set) its value instead of going through its frame.

Definition 1 (computional state). *Let V denotes the set of variables in scope at a given program point p. The set of possible* states *at p is*

$$\Sigma_V = \left\{ \langle \phi, \mu \rangle \;\middle|\; \begin{array}{l} 1.\ \phi \textit{ is a frame over } V \textit{ and } \mu \textit{ is a heap} \\ 2.\ \mathrm{rng}(\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \\ 3.\ \forall \ell \in \mathrm{dom}(\mu).\ \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \end{array} \right\}$$

Conditions 2 and 3 guarantee the absence of dangling pointers. Given $\sigma = \langle \phi, \mu \rangle \in \Sigma_V$, we use ϕ_σ and μ_σ to refer to its frame and heap respectively. □

Now we define the notion of *Denotations* which are the input/output semantics of a piece of code. Basically they are mappings from states to states which describe how the input state is changed when the corresponding code is executed. *Interpretations* are a special case of denotations which provide a denotation for each method in terms of its input and output variables.

Definition 2. *A* denotation *δ from V to V' is a partial function from Σ_V to $\Sigma_{V'}$. We often refer to $\delta(\sigma) = \sigma'$ as $(\sigma, \sigma') \in \delta$. The set of denotations from V to V' is $\Delta(V,V')$. An* interpretation *ι maps methods to denotations and is such that $\iota(m) \in \Delta(m^i, m^i \cup m^o)$ for each method $m = \kappa.\mathtt{m}(t_1, \dots, t_p) : t$ in the given program. The set of all possible interpretations is written as $\mathbb{I}$.* □

The denotational semantics associates a denotation to each command of the language. Let V denotes a set of variables. Let $\iota \in \mathbb{I}$. We define the *denotation for commands* $\mathcal{C}^\iota_V[\![\_]\!] : com \mapsto \Delta(V,V)$, as their input/output behaviour:

$$\begin{array}{rl}
\mathcal{C}^\iota_V[\![v{:=}c]\!] = & \{(\sigma, \sigma[\phi_\sigma(v) \mapsto c]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w]\!] = & \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}\mathtt{new}\ \kappa]\!] = & \{(\sigma, \sigma[\mu_\sigma(\ell) \mapsto newobj(\kappa)]) \mid \ \sigma \in \Sigma_V,\ \ell \notin \mathrm{dom}(\mu_\sigma)\} \\
\mathcal{C}^\iota_V[\![v{:=}w + z]\!] = & \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w) + \phi_\sigma(z)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w.f]\!] = & \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \mu_\sigma\phi_\sigma(w).f]) \mid \sigma \in \Sigma_V, \phi_\sigma(w) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V[\![v.f{:=}w]\!] = & \{(\sigma, \sigma[\mu_\sigma\phi_\sigma(v).f \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V, \phi_\sigma(v) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V\left[\!\!\left[\begin{array}{r}\mathtt{if}\ e\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2\end{array}\right]\!\!\right] = & \begin{array}{l}\{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \mid \sigma \models e \approx true\} \cup \\ \{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_2]\!] \mid \sigma \models e \approx false\}\end{array} \\
\mathcal{C}^\iota_V[\![com_1; com_2]\!] = & \{(\sigma, \sigma'') \mid (\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \wedge (\sigma', \sigma'') \in \mathcal{C}^\iota_V[\![com_2]\!]\}
\end{array}$$

The denotation for a method call $\mathcal{C}^\iota_V[\![v{:=}v_0.\mathtt{m}(v_1, \dots, v_p)]\!]$ should consider the denotation $\iota(m)$ (where m is the called method) and extend it to fit in the calling scope and update the variable v. Assume the method signature is $\mathtt{m}(t_1, \dots, t_p){:}t$, and that we have a lookup procedure $\mathcal{L}$ that, for any given $\sigma \in \Sigma_V$, fetches the actual method that is called depending on the run-time class of v_0. Then the method call denotation is defined as follows:

$$\left\{ (\sigma, \langle \phi_\sigma[v \mapsto \phi''_\sigma(out)], \mu''_\sigma \rangle) \;\middle\|\; \begin{array}{l} 1.\ \sigma \in \Sigma_V, \phi_\sigma(v_0) \in \mathrm{dom}(\mu_\sigma); \\ 2.\ m = \mathcal{L}(v_0, \sigma, \mathtt{m}(t_1, \dots, t_p){:}t); \\ 3.\ (\sigma', \sigma'') \in \iota(m); \\ 4.\ \mu_\sigma \equiv \mu'_\sigma, \forall 0{\le}i{\le}p.\ \phi_\sigma(v_i) = \phi'_\sigma(w_i) \end{array} \right\}$$

The concrete denotational semantics of a program is the least fixpoint of the following transformer of interpretations [3].

Definition 3 (Denotational semantics). *The denotational semantics of a program P is defined as $\bigcup_{i \geq 0} T^i_P(\iota_0)$, i.e. the* least fixed point *of T_P where T_P is:*

$$T_P(\iota) = \left\{ (m, X) \left\| \begin{array}{l} 1.\ m \in P \\ 2.\ \sigma \in \Sigma_{m^s}, \forall v \in m^l.\ \phi_\sigma(v) = 0 \ \mathit{or}\ \phi_\sigma(v) = \mathtt{null} \\ 3.\ X = \{(\sigma|_{m^i}, \sigma'|_{m^i \cup m^o}) \mid (\sigma, \sigma') \in \mathcal{C}^\iota_{m^s}[\![m^b]\!]\} \end{array} \right. \right\}$$

and $\iota_0 = \{(m, \emptyset) \mid m \in P\}$ *and* $\forall \iota_1, \iota_2 \in \mathbb{I}$ *the union* $\iota_1 \cup \iota_2$ *is defined as* $\{(m, X_1 \cup X_2) \mid m \in P, (m, X_1) \in \iota_1, (m, X_2) \in \iota_2\}$ □

3 Constancy Analysis

We want to design an analysis to infer definite information about constant data structures. This can be done by tracking data structures that are not modified (definite information), or by tracking data structures that might be modified (may information). We follow the latter approach as we believe it easier. In addition, we want to analyze methods in a context independent way, and later adapt the result to any calling context.

Example 1. Consider the following method:

```
A m(x:A, y:A) with {} is y:=y.next; x.next:=y; out:=y;
```

The only command that might modify the heap structure is "`x.next:=y`". Note that "`y:=y.next`" does not affect the heap structure but rather changes the heap location stored in `y`. This method might be called in different contexts where the actual parameters: (1) do not have any common data structure; or (2) have a common data structure. In the first case, "`x.next:=y`" might modify only the data structure pointed by the first argument. In the second case, it might modify a common data structure for `x` and `y`, and therefore we say that both arguments might be modified. We describe this behaviour by the Boolean formula $\check{x} \wedge (\check{y} \leftrightarrow x\check{\cdot}y)$, which is interpreted as: (1) in any calling context, the data structure the first argument points to when the method is called might be modified by the method (expressed by $\check{x}$); and (2) the data structure that the second argument points to when the method is called, might be modified by the method (expressed by $\check{y}$) iff x and y might share a data structure when the method is called (expressed by $x\check{\cdot}y$).

□

We define now the set of reachable heap locations from a given reference variable, which we need to define the notion of *constant heap structure.*

Definition 4 (reachable heap locations). *Let* μ *be a heap. The set of* locations reachable from $\ell \in \mathrm{dom}(\mu)$ *is* $L(\mu, \ell) = \cup\{L^i(\mu, \ell) \mid i \geq 0\}$ *where* $L^0(\mu, \ell) = \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L}$ *and* $L^{i+1}(\mu, \ell) = \cup\{\mathrm{rng}(\mu(\ell')) \cap \mathbb{L} \mid \ell' \in L^i(\mu, \ell)\}$. *The set of reachable heap locations from* v *in* $\sigma \in \Sigma_V$*, denoted* $L_V(\sigma, v)$*, is* $\{\phi_\sigma(v)\} \cup L(\mu_\sigma, \phi_\sigma(v))$ *if* $\phi_\sigma(v) \in dom(\mu_\sigma)$*; and the empty set otherwise.* □

Definition 5 (constant reference variable). *A reference variables $v \in V$ is constant with respect to a denotation δ, denoted $\mathsf{c}(v, \delta)$, iff for any $(\sigma_1, \sigma_2) \in \delta$ all locations in $L_V(\sigma_1, v)$ are constant with across δ, namely $\forall \ell \in L_V(\sigma_1, v)$, $\mu_{\sigma_1}(\ell)$ and $\mu_{\sigma_2}(\ell)$ have the same class tag and agree on their* reference *field values.* $\square$

The definition above considers modifications of fields of reference type only. The reason for concentrating on reference fields is that we have developed this analysis for a specific need which requires tracking updates only in the shape of the data structure (see Section 4). Tracking updates of integer fields can simply done by modifying the above definition to consider those updates. In what follows, a modification of a variable stands for a modification of the shape of the heap structure reachable from that variable.

Definition 6 (common heap location). *$x, y \in V$ have a common heap location (share) in a state $\sigma \in \Sigma_V$ if and only if $L_V(\sigma, x) \cap L_V(\sigma, y) \neq \emptyset$* $\square$

We define now an abstract domain which captures a set of variables that *might* be modified by a concrete denotation.

Definition 7 (update abstract domain). *The update abstract domain U_V is a partial order $\langle \wp(V), \subseteq \rangle$. Its concretization function $\gamma_V{:}\mathsf{U}_V \to \Delta(V,V')$ is defined as $\gamma_V(X) = \{\delta \mid \forall v \in V.\ \neg\mathsf{c}(v, \delta) \to (v \in X)\}$.* $\square$

As we have seen in Example 1, information about possible sharing between variables is important for a precise constancy analysis. There are many ways for inferring such information. Here, we use the pair-sharing domain [11]. Moreover, constancy information improves the precision of method calls in pair sharing analysis. This is because the execution of a method m can introduce sharing between non-constant parameters only. Hence we design an analysis over the (reduced) product of the update domain U_V and of the pair-sharing domain SH_V, denoted by $\mathsf{SH}{\times}\mathsf{U}_V$. Informally, the pair sharing domain abstracts an element $s \in \wp(\Sigma_V)$ to a set sh of symmetric pairs of the form (x, y) where $x, y \in V$. If $(x, y) \in sh$ then x and y *might* share in s, and if $(x, y) \notin sh$ then they cannot share, so that if $(x, x) \notin sh$ then x must be `null` in s. In what follows, instead of saying *might share* we simply say share.

Figure 1 defines abstract denotations for our simple language over $\mathsf{SH}{\times}\mathsf{U}_V$. They are Boolean functions corresponding to the elements of $\mathsf{SH}{\times}\mathsf{U}_V$. For a piece of code C, the Boolean variables:

– $x\check{\cdot}y$ and $x\hat{\cdot}y$ indicate if x and y *share* before and after executing C, respectively. Since pair sharing is symmetric, $x\check{\cdot}y$ and $y\check{\cdot}x$ are equivalent Boolean variables; and
– $\check{x}$ and $\hat{x}$ indicate if x is modified with respect to its value before and after C (by the program execution), respectively.

Each abstract denotation is defined in terms of a Boolean function $\varphi \wedge \psi$, where φ propagates (forward) *sharing* information and ψ propagates (backwards) *update* information. In what follows we explain the meaning of each abstract denotation:

$$
\begin{aligned}
\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!] &= \varphi \wedge \psi \\
&-\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \\
&-\varphi_1 = (\wedge\{\neg x\hat{\cdot}v \mid x \in V\}) \\
&-\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!] &= \varphi \wedge \psi \\
&-\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \wedge \varphi_2 \\
&-\varphi_1 = \wedge\{x\hat{\cdot}v \leftrightarrow x\check{\cdot}w \mid x \in V\backslash\{v\}\} \\
&-\varphi_2 = w\check{\cdot}w \leftrightarrow v\hat{\cdot}v \\
&-\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{new}\ \kappa]\!] &= \varphi \wedge \psi \\
&-\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge v\hat{\cdot}v \wedge \varphi_1 \\
&-\varphi_1 = (\wedge\{\neg x\hat{\cdot}v \mid x \in V \setminus \{v\}\}) \\
&-\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}w.f]\!] &= \mathcal{A}^{\iota}_{V}[\![v{:=}w]\!] \\
\mathcal{A}^{\iota}_{V}[\![v.f{:=}w]\!] &= \varphi \wedge \psi \\
&-\varphi = \wedge\{x\hat{\cdot}y \leftrightarrow x\check{\cdot}y \vee (x\check{\cdot}w \wedge y\check{\cdot}v) \mid x, y \in V\} \\
&-\psi = \{\check{x} \leftrightarrow v\check{\cdot}x \vee \hat{x} \mid x \in V\} \\
\mathcal{A}^{\iota}_{V}[\![\texttt{if}\ e\ \ldots]\!] &= \mathcal{A}^{\iota}_{V}[\![c_1]\!] \vee \mathcal{A}^{\iota}_{V}[\![c_2]\!] \\
\mathcal{A}^{\iota}_{V}[\![c_1; c_2]\!] &= \mathcal{A}^{\iota}_{V}[\![c_1]\!] \circ \mathcal{A}^{\iota}_{V}[\![c_2]\!] \\
\mathcal{A}^{\iota}_{V}[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!] &= \phi \wedge \varphi \wedge \psi \\
&\phi_m = \vee\{\iota(m) \mid m \text{ might be called }\} \\
&\phi = \phi_m[s_i \mapsto v_i, \mathit{out} \mapsto v, \mathit{this} \mapsto v_0] \\
&\varphi = \wedge\{x\hat{\cdot}y \leftrightarrow x\check{\cdot}y \vee \varphi_1 \mid x, y \in V\backslash\{v_0, \ldots, v_p\}\} \\
&\varphi_1 = \vee\{(x\check{\cdot}v_i \wedge y\check{\cdot}v_j \wedge v_i\hat{\cdot}v_j \wedge (\check{v}_i \vee \check{v}_j)) \mid i, j \in \{0, \ldots, p\}\} \\
&\psi = \psi_1 \wedge (\check{v} \leftrightarrow \psi_3 \vee \psi_2(v)) \\
&\psi_1 = \wedge\{\check{x} \leftrightarrow \hat{x} \vee \psi_2(x) \mid x \in V\backslash\{v, v_0, \ldots, v_p\}\} \\
&\psi_2(x) = \vee\{(x\check{\cdot}v_i \wedge \check{v}_i) \mid i \in \{0, \ldots, p\}\} \\
&\psi_3 = \{x\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}
\end{aligned}
$$

Fig. 1. Abstract Denotations over $\mathsf{SH}\times\mathsf{U}_V$

- $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); and nothing can share with v after C (φ_1). (**U**) $x \in V\backslash\{v\}$ is modified before C iff it is modified after C, and v is modified before C iff it shares with some y before C and y is modified after C.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); since v becomes an alias for w then v can share with $x \in V\backslash\{v\}$ after C iff x shares with w before C (φ_1); and v can share with itself after C (i.e., not `null`) iff w shares with itself before C (φ_2). (**U**) the same as for "$v{:=}\texttt{null}$".
- $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{new}\ \kappa]\!]$: the same as $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!]$ except that v shares with itself after executing the statement.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}w.f]\!]$: the same as $\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!]$ since the analysis is field insensitive.
- $\mathcal{A}^{\iota}_{V}[\![v.f{:=}w]\!]$: (**SH**) $x, y \in V$ share after C iff before C, they shared or x shared with w and y with v; (**U**) $x \in V$ is modified before C, iff it shares with v before C or x is modified after C.
- $\mathcal{A}^{\iota}_{V}[\![\texttt{if}\ e\ \ldots]\!]$: combines the branches through logical or.

- $\mathcal{A}^{\iota}_{V}[\![c_1; c_2]\!]$: combines $\mathcal{A}^{\iota}_{V}[\![c_1]\!]$ and $\mathcal{A}^{\iota}_{V}[\![c_2]\!]$. This is simply done by matching the output variables of the first denotation with the input variables of the second denotation.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!]$: (1) First we fetch the abstract denotations of all methods that might be called, and we combine them through logical or into ϕ_m; (2) Assuming that the method denotations use $s_i \neq v_i$ for the i-th formal parameter, we rename all sharing information by changing each s_i into v_i and *out* into v. We get ϕ. (3) We add sharing information for variables which are not in $V \backslash \{v, v_0, \ldots, v_p\}$. The sharing component φ states that x and y might share after the call iff they shared before (i.e. $x \check{\cdot} y$) or they shared with arguments v_i and v_j where v_i and v_j share after the call, and either v_i or v_j has been modified (expressed by φ_1); (4) We add the constancy information which states that $x \in V \backslash \{v\}$ is modified before iff it is modified after, or if it shares with a variable that is modified by the method. For v it is a bit different since we exclude the case that if v is modified after then it is modified before, since we possibly assign to it a new reference.

The abstract denotation for a method:

$$t\ \mathtt{m}(w_1{:}t_1, \ldots, w_n{:}t_n)\ \mathtt{with}\ w_{n+1}{:}t_{n+1}, \ldots, w_{n+m}{:}t_{n+m}\ \mathtt{is}\ \mathit{com},$$

is then defined as $\phi_m = \exists V'.\ \mathcal{A}^{\iota}_{V}[\![\mathit{com}]\!] \wedge \varphi_1 \wedge \varphi_2$ where:

- $S = \{s_1, \ldots, s_n\}$ such that $S \cap m^s = \emptyset$, and $V = m^s \cup S$
- $\varphi_1 = \{\neg x \check{\cdot} y \mid x \in m^l \cup \{\mathit{out}\}, y \in m^s\}$
- $\varphi_2 = \{s_i \check{\cdot} x \leftrightarrow w_i \check{\cdot} x \mid 1 \leq i \leq n, x \in m^i\}$
- $V' = \{x \check{\cdot} y, x \hat{\cdot} y, \check{x}, \hat{x} \mid x \notin S \cup \{\mathit{this}, \mathit{out}\}, y \in V\} \cup \{\check{\mathit{out}}\}$

The idea is that we: (1) extend m^l to V in order to include shallow variable s_i for each method argument w_i; (2) compute $\mathcal{A}^{\iota}_{V}[\![\mathit{com}]\!]$; (3) add φ_1 which indicates that local variables are initialized to `null`; (4) add φ_2 which creates the connection between the shallow variables and the actual parameters; (5) eliminate all local information by removing the Boolean variables V'. The abstract denotational semantics can be then defined similar to the concrete one in Definition 3, where the initial method summaries are false ans summaries are combined (during the fixpoint iterations) using the logical or $\vee$.

Example 2. Applying the above abstract semantics to the method defined in Example 1 results in a Boolean formula whose constancy component is $(\check{\mathit{this}} \leftrightarrow \hat{\mathit{this}}) \wedge \check{x} \wedge (\check{y} \leftrightarrow (x \check{\cdot} y \vee \hat{y}))$. For simplicity we ignore the part of ϕ_m that talks about sharing.

4 Experiments

We show here some experiments with our domain for sharing and constancy analysis. They have been performed with the Julia analyzer [12] on a Linux

Program	M	Sharing				Non-Cyclicity			
		T		P		T		P	
JLex	446	1595	(2324)	34.30%	(34.84%)	506	(415)	34.03%	(35.21%)
JavaCup	933	5707	(6486)	22.24%	(23.76%)	853	(953)	59.23%	(76.13%)
Kitten	2131	20976	(27824)	17.90%	(19.11%)	2538	(3177)	36.34%	(41.13%)
jEdit	3206	47408	(49356)	21.12%	(21.28%)	4969	(5963)	43.49%	(47.50%)
Julia	4028	79199	(129562)	9.71%	(10.25%)	8014	(12018)	33.40%	(38.17%)

Fig. 2. The effect of the purity component on Sharing and Non-Cyclicity. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

machine based on a 64 bits dual core AMD Opteron processor 280 running at 2.4Ghz, with 2 gigabytes of RAM and 1 megabyte of cache, by using Sun Java Development Kit version 1.5. All programs have been analyzed including all library methods that they use inside the `java.lang.*` and `java.util.*` hierarchies.

Figure 2 compares sharing analysis alone with sharing analysis in reduced product with constancy (Section 3), and its effect on non-cyclicity analysis [9]. In each column, numbers in parentheses correspond to the analysis using the reduced product. For each program, it reports the number of methods analyzed, including the libraries, and time and precision of the corresponding analysis with and without constancy. For sharing, the precision is the amount of pairs of variables of reference type that are proved not to share at the program points preceding the update of an instance field, the update of an array element or a method call. This is sensible since there is where sharing analysis is used by subsequent analyses. That figure suggests that the constancy component slightly improves the precision of sharing analysis. However, the importance of constancy is shown when we consider its effects on a static analysis that uses constancy information. This is the case of non-cyclicity analysis, which finds variables bound to non-cyclical data structures [9]. Figure 2 shows that the computation of cyclicity analysis after a simple sharing analysis leads to less precise results than the same computation after a sharing and constancy analysis. Here, precision is the number of field accesses that read the field of a non-cyclical object. This is sensible since there is where non-cyclicity is typically used.

The importance of constancy analysis becomes more apparent when it supports a static analysis that uses constancy, sharing and cyclicity information. This is the case of *path-length* [13]. It approximates the length of the maximal path of pointers one can follow from each variable. This information is the basis of a termination [1] and resource bound analyses [2] for programs dealing with dynamic data structures. Figure 3 shows the effects of constancy on path-length and termination analysis (available in [12]) of a set of small programs that do not use libraries except for `java.lang.Object`. Times are in milliseconds and precision is the number of methods proved to terminate. Constancy information

Program	M	T	P
`Init`	10	102 (140)	8 (8)
`List`	11	624 (512)	6 (11)
`Diff`	5	6668 (9040)	5 (5)
`Hanoi`	7	548 (868)	7 (7)
`BTree`	7	306 (415)	6 (7)
`BSTree`	10	234 (273)	9 (10)
`Virtual`	11	357 (418)	10 (11)
`ListInt`	11	767 (507)	6 (11)

Program	M	T	P
`Nested`	4	324 (447)	4 (4)
`Double`	5	270 (268)	5 (5)
`FactSum`	6	169 (178)	6 (6)
`Sharing`	7	309 (501)	6 (7)
`Factorial`	5	102 (196)	5 (5)
`Ackermann`	5	1308 (1732)	5 (5)
`BubbleSort`	5	871 (951)	5 (5)
`FactSumList`	8	278 (703)	7 (8)

Fig. 3. The effect of the purity information on Termination analysis. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

results in proving that all terminating methods terminate (only 2 methods of `Init` are not proved to terminate: they actually diverge). Without constancy information, many terminating methods are not proved to terminate.

These experiments suggest that constancy information contributes to the precision of sharing, cyclicity, path-length and hence termination analysis. Computing constancy information with sharing requires more time than computing sharing alone (Figure 2). Performing other analyses by using the constancy information increases the times further (Figures 2 and 3). Nevertheless, this is justified by the extra precision of the results.

5 Acknowledgments

This work of Samir Genaim was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project, and a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science, Oslo, Norway, June 2008. Springer-Verlag, Berlin. To appear.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19-20:149–197, 1994.
4. N. Cataño and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2003.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center, 1998.
7. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML. Technical Report 96-06p, Iowa State University, 2001.
8. Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.
9. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006.
10. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In R. Cousot, editor, *Proc. of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215, Paris, France, 2005. Springer.
11. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. of Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005.
12. F. Spoto. The JULIA Static Analyser. `profs.sci.univr.it/∼spoto/julia`, 2008.
13. F. Spoto, P. M. Hill, and E. Payet. Path-Length Analysis for Object-Oriented Programs. In *Proc. of Emerging Applications of Abstract Interpretation*, Vienna, Austria, March 2006. `profs.sci.univr.it/∼spoto/papers.html`.
14. F. Spoto and E. Poll. Static Analysis for JML's assignable Clauses. In G. Ghelli, editor, *Proc. of FOOL-10, the 10th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New Orleans, Louisiana, USA, January 2003. ACM Press. Available at `www.sci.univr.it/∼spoto/papers.html`.
15. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.