Path-length analysis for object-oriented programs
Fausto Spoto, Patricia M Hill, Etienne Payet

To cite this version:
Fausto Spoto, Patricia M Hill, Etienne Payet. Path-length analysis for object-oriented programs. First International Workshop on Emerging Applications of Abstract Interpretation (EAAI’06), Mar 2006, Vienne, Austria. hal-01915777

HAL Id: hal-01915777
https://hal.univ-reunion.fr/hal-01915777
Submitted on 8 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Path-Length Analysis for Object-Oriented Programs

Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy

Patricia M. Hill

School of Computing, University of Leeds, United Kingdom

Étienne Payet

IREMIA, Université de La Réunion, France

Abstract
This paper describes a new static analysis for finding approximations to the path-length of variables in imperative, object-oriented programs. The path-length of a variable \( v \) is the cardinality of the longest chain of pointers that can be followed from \( v \). It is shown how such information may be used for automatic termination inference of programs dealing with dynamically created data-structures.

Key words: Abstract interpretation, abstract compilation, termination analysis, object-oriented programming.

1 Introduction
In this paper, by applying the standard framework of abstract interpretation \cite{5}, we present a new static analysis, called path-length analysis, that provides information useful for verifying the termination of imperative, object-oriented program code. Note that, since termination of computer programs is an undecidable property, a termination analysis can only be approximate: if the analyzer proves termination, then the program is guaranteed to terminate, but, in general, the converse cannot hold.

\textsuperscript{1} Email: fausto.spoto@univr.it
\textsuperscript{2} Email: hill@comp.leeds.ac.uk
\textsuperscript{3} Email: Etienne.Payet@univ-reunion.fr

This paper is electronically published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
For logic and functional programs, termination analysis has been widely researched [1,7,8,11,12,13]; often the proof of termination of such programs will aim to show that recursive calls strictly decrease w.r.t. a given norm. Norms can both measure the size of the terms, and information more specific to the kind of algorithm implemented by the program. For instance, for processing lists which, for logic programs, are a standard, ubiquitous data-structure, the length of the list is a useful norm. While this technique works fine for terms, others consider termination in the presence of numerical variables [18].

For imperative or object-oriented programs, the problem seems more difficult than for (pure) logic programs because of the sharing of data-structure between variables, cyclicity of data-structures and destructive updates. Moreover, apart from using the values of the numerical variables [14], norms relating to the structure of the data such as list-length for logic programs have not been considered. As a consequence, termination analyses for imperative programs have been restricted, up to now, to programs dealing with numerical variables only [3,10,15]. Thus, in spite of the fact that automatic certification of termination is particularly relevant for real-time systems, where a program has strict time constraints [4], research where non-termination may be due to factors other than the actual values of the numerical variables is lacking.

In this paper, we are concerned with object-oriented programs, where objects are the main entities; so that their size and size of the data-structures contained therein, are natural and useful norms for establishing termination. The static path-length analysis we present here aims to provide a description of the path-length of the program variables where the path-length of a variable refers to the cardinality of the longest chain of pointers that can be followed from that variable. The analysis is presented and proved correct in the abstract interpretation framework [5] for static analysis. The main aim of the analysis is to support (automatic) termination inference for programs working over dynamically created data-structures (objects).

This paper shows how to transform a concrete program dealing with dynamically created data-structures into an abstract program dealing with numerical constraints on the path-lengths of the data-structures. Observe that the transformation may be viewed as an abstract compilation [9], since it is defined as a fixpoint computation over an abstraction of the program; this allows optimisation to take place before the abstract fixpoint is computed. Proofs of correctness can be found in [19].

To obtain an intuition of our analysis, consider the classes in Figure 1, which are written in the syntax of Section 4. That syntax is a normalised version of the syntax of a Java-like language; that normalisation (Section 4) is a useful simplification for presenting the analysis (Section 5); here we just need to know that the keyword with introduces the methods’ local variables, and the variable out holds the methods’ return value. The classes implement a list of subscriptions to a service, such as cable television. Some subscriptions come from abroad, and have a higher monthly cost than others. The method
foreigners over a list of subscriptions builds a new list, containing only the foreign ones. Let variable s have type Subs and consider the following loop:

$$\text{while } (s \neq \text{null}) \{ s := s.\text{next}; \}$$

Let us write $\hat{s}$ for the path-length of $s$ before the command $s := s.\text{next}$ is executed and $\hat{s}$ for its path-length afterwards. We call $\hat{s}$ an input variable and $\hat{s}$ an output variable. A condition which lets us prove termination of the loop is $\hat{s} > \hat{s}$. It expresses the fact that the path-length of $s$ strictly decreases at each iteration of the loop. On the assumption that $s$ does not contain any cycle, our analysis compiles the command $s := s.\text{next}$ into that constraint automatically. This shows the importance of cyclicity information (which corresponds to occur-check in logic programs) and the use of a pair of variables $\check{v}, \hat{v}$ for the same program variable $v$, to model destructive updates.

Consider next a more complicated example:

$$\text{while } (s \neq \text{null}) \{ s := s.\text{foreigners}(); s := s.\text{next}; \}$$

The loop body now consists of two commands. The first, $s := s.\text{foreigners}()$, satisfies the constraint $\hat{s} \geq \check{s}$ (removing local subscriptions can only shrink the list of subscriptions). Our path-length analysis will derive that constraint, as shown in more detail in Section 2. The second command $s := s.\text{next}$ is compiled into the constraint $\hat{s} > \check{s}$, as before, so that the compilation of the body of the loop is now the composition $(\hat{s} \geq \check{s}) \circ (\hat{s} > \check{s})$. This is computed by matching the output variable $\hat{s}$ on the left of $\circ$ with the input variable $\check{s}$ on the right, taking the conjunction and then projecting the matched variables away; this results in the constraint $\hat{s} > \check{s}$ proving that this loop also terminates.

The paper is organized as follows. Section 2 presents a detailed example of path-length analysis. Section 3 contains the mathematical preliminaries of the work. Section 4 presents a simplified object-oriented language. Section 5 defines the path-length constraints and the abstract semantics which derives them for the program under analysis. Section 6 concludes.

## 2 A Detailed Example of Analysis

Let us show the analysis for the two methods foreigners in Figure 1. Both methods work over a set of variables $V = \{\text{temp, this, out}\}$. In this example, we want to prove that both foreigners methods in Figure 1 satisfy the invariant $\text{out} \leq \text{this}$, which states that the path-length of the list of subscriptions returned by those methods is no longer than the path-length of the list provided to them as input, through the implicit this parameter.

The analysis starts by abstractly compiling the two foreigners methods. The rules for this abstraction are given in Figure 3. Let us compute the abstraction of the method Subs.foreigners, assuming it is called on a non-cyclical object. By the rule for v.f in Figure 3, the abstraction of this.next is
class Object {}
class Subs extends Object {
  int channels; Subs next;
  int monthlyCost() { out := channels / 2 } // in euros
  ForeignSubs foreigners() with temp:Subs {
    temp := this.next;
    if (temp = null) then {} else out := temp.foreigners()
  }
}
class ForeignSubs extends Subs {
  int monthlyCost() { out := channels * 2 } // more expensive
  ForeignSubs foreigners() with temp:Subs {
    out := new ForeignSubs; // program point *
    temp := this.next;
    if (temp = null) then {} else out.next := temp.foreigners();
    out.channels := this.channels
  }
}

Fig. 1. Our running example: a list of subscriptions to a service.

\[(U(V) \land \text{res} < \text{this}) = \left(\begin{array}{l}
\hat{\text{out}} = \text{out} \land \hat{\text{temp}} = \text{temp}\land \\
\text{this} = \text{this} \land \text{res} < \text{this}
\end{array}\right) \circ (U(V \setminus \text{temp}) \land \hat{\text{temp}} = \text{res})\]

Equation (1) states that the path-length of the three variables in scope does not change by evaluating \(\text{this}.\text{next}\), while the result of this evaluation, stored in the distinguished variable \(\text{res}\), has a path-length which is strictly smaller than that of \(\text{this}\). To abstract the assignment \(\text{temp} := \text{this}.\text{next}\), we use the rule for \(v := \text{exp}\) in Figure 3. The abstraction is then

\[
\left(\begin{array}{l}
\hat{\text{out}} = \text{out} \land \hat{\text{temp}} = \text{temp}\land \\
\text{this} = \text{this} \land \text{res} < \text{this}
\end{array}\right) \circ \left(\begin{array}{l}
\hat{\text{out}} = \text{out} \land \hat{\text{this}} = \text{this}\land \\
\text{temp} = \text{res}
\end{array}\right).\]  

(2)

We compute the sequential composition \(\circ\) of two path-length constraints by identifying the output variables of the left-hand side of \(\circ\) with the input variables of its right-hand size. These variables are then projected away. For (2) we get

\[
\exists_{\{\text{out, res, temp, this}\}} \left(\begin{array}{l}
\text{out} = \hat{\text{out}} \land \text{temp} = \hat{\text{temp}} \land \text{this} = \hat{\text{this}} \land \text{res} < \hat{\text{this}}\land \\
\hat{\text{out}} = \text{out} \land \hat{\text{this}} = \text{this} \land \text{temp} = \text{res}
\end{array}\right) = (\hat{\text{out}} = \text{out} \land \hat{\text{this}} = \text{this} \land \hat{\text{temp}} < \text{this}).\]  

(3)

Equation (3) says that, by assigning \(\text{this}.\text{next}\) to \(\text{temp}\), the final path-length
of \( temp \) is strictly smaller than the initial path-length of \( this \).

Consider now the abstraction of \( temp.f\)oreigners(). A preliminary sharing analysis determines that \( out \) is the only variable which does not share, here, with \( temp \). Hence \( NS = \{ out \} \) (Figure 3) and the abstraction of this method call is

\[
(o\tilde{u}t = o\tilde{u}t) \land \left( J(Subs.f\)oreigners) \sqcup \begin{cases} \tilde{this} \mapsto \tilde{temp}, \\ \tilde{out} \mapsto res \end{cases} \right)_{\tilde{\iota}}. \tag{4}
\]

The interpretations \( J \) for the two \( f\)oreigners methods are currently unknown and we cannot simplify (4). The abstraction of \( out := temp.f\)oreigners() is hence \( (4) \circ (\tilde{this} = \tilde{this} \land \tilde{temp} = \tilde{temp} \land out = res) = \iota[\tilde{this} \mapsto \tilde{temp}] \), where \( \iota \) is defined in (4). We can now compute the abstraction of the conditional, which by Figure 3 is

\[
\left( \begin{array}{l}
0 = \tilde{temp} \land \tilde{temp} = \tilde{temp} \\
\tilde{this} = \tilde{this} \land \tilde{out} = \tilde{o\tilde{u}t}
\end{array} \right) \sqcup (0 < \tilde{temp} \land \iota[\tilde{this} \mapsto \tilde{temp}]) = \left( \begin{array}{l}
0 = \tilde{temp} = \tilde{temp} \\
\tilde{this} = \tilde{this} \land \tilde{out} = \tilde{out}
\end{array} \right) \sqcup (0 < \tilde{temp} \land \iota[\tilde{this} \mapsto \tilde{temp}]). \tag{5}
\]

The abstraction of the whole \( Subs.f\)oreigners should hence be \( (3) \circ (5) \). However, at the beginning of its execution, both \( out \) and \( temp \) are bound to \( null \). Hence we can assume \( o\tilde{u}t = \tilde{temp} = 0 \). And at its end, we only want to observe the value of \( out \). Hence we remove all output variables but \( o\tilde{u}t \), by composing with \( o\tilde{u}t = \tilde{o\tilde{u}t} \). In conclusion, the abstract compilation of that method is \( (o\tilde{u}t = 0) \land [(3) \circ (5) \circ (o\tilde{u}t = o\tilde{u}t)] \), that is

\[
(o\tilde{u}t = 0) \land \left[ ((3) \circ (0 = \tilde{temp} = \tilde{temp} \land \tilde{this} = \tilde{this} \land o\tilde{u}t = o\tilde{u}t)) \sqcup ((3) \circ (0 < \tilde{temp} \land \iota[\tilde{this} \mapsto \tilde{temp}])) \right] \circ (o\tilde{u}t = o\tilde{u}t)
\]

which we simplify into

\[
(o\tilde{u}t = 0) \land \left[ (0 = \tilde{temp} \land 0 < \tilde{this} \land \tilde{this} = \tilde{this} \land o\tilde{u}t = o\tilde{u}t) \sqcup \exists_{\tilde{\text{temp}}}(0 < \tilde{temp} < \tilde{this} \land \iota[\tilde{this} \mapsto \tilde{temp}]) \right] \circ (o\tilde{u}t = o\tilde{u}t). \tag{6}
\]

Consider \( ForeignSubs.f\)oreigners now. We abstract its first two lines in

\[
o\tilde{u}t = 1 \land \tilde{this} = \tilde{this} \land \tilde{temp} < \tilde{this}. \tag{7}
\]
The abstraction of the call \textit{temp.foreigners()} is as in Equation (4). To abstract \texttt{out.next := temp.foreigners()}, we use the rule for \texttt{v.f := exp} in Figure 3, with \( S = \{ \text{out} \} \). Hence the abstraction of this field update is

\[
\text{(4) } \circ (\text{out} \leq \text{out} + \text{res} \land \text{this} = \text{this} \land \text{temp} = \text{temp})
\]

\[
= \exists_{\text{temp}} \left[ l[\text{this} \mapsto \text{temp}, \text{out} \mapsto \text{res}] \land \text{out} \leq \text{out} + \text{res} \right]. \quad (8)
\]

The abstraction of the conditional is similar to that seen before. We do not need to abstract \texttt{out.channels := this.channels} since it deals with integer values only, so it is irrelevant to our analysis. The abstraction of the method \texttt{ForeignSubs.foreigners()} is, in conclusion:

\[
(\text{out} = \text{temp} = 0) \land \\
\{ (0 = \text{temp} = \text{temp} \land \text{this} = \text{this} \land \text{out} = \text{out}) \sqcup (0 < \text{temp} \land (8)) \} \circ (\text{out} = \text{out}). \quad (9)
\]

Now that we have computed the abstraction of both \textit{foreigners} methods, we can prove that \textit{out} \leq \textit{this} is an invariant for them. To this purpose, we plug the interpretation 

\( J(\text{Subs.foreigners}) = J(\text{ForeignSubs.foreigners}) = \iota = (\text{out} \leq \text{this}) \) into Equations (6) and (9). From the first equation we get

\[
(\text{out} = \text{temp} = 0) \land \\
\left[ (0 = \text{temp} \land 0 < \text{this} \land \text{this} = \text{this} \land \text{out} = \text{out}) \sqcup \exists_{\text{temp}} (0 < \text{temp} < \text{this} \land \text{out} \leq \text{temp}) \right] \circ (\text{out} = \text{out})
\]

which entails

\[
\left\{ (\text{out} = 0) \land \left[ (0 = \text{temp} \land 0 < \text{this} \land \text{this} = \text{this} \land \text{out} = \text{out}) \sqcup (\text{out} < \text{this}) \right] \right\} \circ (\text{out} = \text{out})
\]

which itself entails

\[
\left\{ (0 = \text{temp} \land 0 < \text{this} \land \text{this} = \text{this} \land \text{out} = \text{out}) \sqcup (\text{out} < \text{this}) \right\} \circ (\text{out} = \text{out})
\]

and, finally, \( \{ (\text{out} < \text{this}) \sqcup (\text{out} < \text{this}) \} \circ (\text{out} = \text{out}) = (\text{out} < \text{this}). \) Note that \( \text{out} < \text{this} \) entails the invariant \( \text{out} \leq \text{this} \).

For the second one, we first reduce (8) to \( \exists_{\text{res}} (\text{res} \leq \text{temp} \land (\text{out} \leq \text{out} + \text{res})) = (\text{out} \leq \text{out} + \text{temp}) \). Hence (9) is

\[
(\text{out} = \text{temp} = 0) \land \left\{ (7) \circ [(0 = \text{temp} = \text{temp} \land \text{this} = \text{this} \land \text{out} = \text{out}) \sqcup (0 < \text{temp} \land \text{out} \leq \text{out} + \text{temp})] \right\} \circ (\text{out} = \text{out})
\]
which entails

\[(\tilde{\text{out}} = \text{temp} = 0) \land \{ (7) \circ [\text{out} \leq \text{out} + \text{temp}] \} \circ (\tilde{\text{out}} = \text{out}) \]

\[= (\text{out} = \text{temp} = 0) \land \exists_{\text{temp}} \{ \text{temp} < \text{this} \land \text{out} \leq 1 + \text{temp} \} \circ (\tilde{\text{out}} = \text{out}) \]

\[= (\text{out} = \text{temp} = 0 \land \text{out} \leq \text{this}) \circ (\text{out} = \text{out}) = (\text{out} \leq \text{this}).\]

3 Preliminaries

A total (partial) function \( f \) is denoted by \( \mapsto \) (\( \to \)). The domain (codomain) of \( f \) is \( \text{dom}(f) \) (\( \text{rng}(f) \)). We denote by \([v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]\) the function \( f \) where \( \text{dom}(f) = \{v_1, \ldots, v_n\} \) and \( f(v_i) = t_i \) for \( i = 1, \ldots, n \). Its update is \( f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m] \), where the domain may be enlarged. By \( f\vert_s (f\vert_{\bar{s}}) \) we denote the restriction of \( f \) to \( s \subseteq \text{dom}(f) \) (to \( \text{dom}(f) \setminus s \)). If \( f(x) = x \) then \( x \) is a fixpoint of \( f \). The composition \( f \circ g \) of functions \( f \) and \( g \) is such that \( (f \circ g)(x) = g(f(x)) \) so that we also denote it as \( g \circ f \). The components of a pair are separated by \( \ast \). A definition of \( S \) such as \( S = a \ast b \), with \( a \) and \( b \) meta-variables, silently defines the pair selectors \( s.a \) and \( s.b \) for \( s \in S \).

A poset \( S \times \leq \) is a set \( S \) with a reflexive, transitive and antisymmetric relation \( \leq \). If \( C \times \leq \) and \( A \times \leq \) are posets (the concrete and the abstract domain), a Galois connection [6] is a pair of monotonic maps \( \alpha : C \mapsto A \) and \( \gamma : A \mapsto C \) (\( i.e. \), \( \alpha(c_1) \leq \alpha(c_2) \) if \( c_1 \leq c_2 \), similarly for \( \gamma \)) such that \( \gamma \alpha \) is extensive (\( i.e. \), \( c \leq \gamma \alpha(c) \) for any \( c \in C \)) and \( \alpha \gamma \) is reductive (\( i.e. \), \( \alpha \gamma(a) \leq a \) for any \( a \in A \)). It has been shown that in order to define a Galois connection, and hence an abstract interpretation [6], it is enough to prove that \( A \) is (isomorphic to) a Moore family of \( C \) \( i.e. \), that it is closed w.r.t. greatest lower bounds of \( C \) and that it contains the top element of \( C \).

4 Our Simple Object-Oriented Language

Syntax. Variables are typed and bound to values. We do not consider primitive types that are not heap-allocated but rather held inside the activation frame or the local frame of an object.

Definition 4.1 A program has a set of variables (or identifiers) \( V \) (including \( \text{res}, \text{out}, \text{this} \)) and a finite poset of classes (or types) \( K \times \leq \) ordered by a subclass relation \( \leq \). We write \( F(\kappa) \) for the set of fields of class \( \kappa \in K \).

Example 4.2 In Figure 1, we have \( K = \{\text{Object, Subs, ForeignSubs}\} \), where \( \text{Object} \) is the top of the hierarchy. Moreover, \( \text{ForeignSubs} \leq \text{Subs} \). We are not interested in primitive types. Hence we have \( F(\text{Object}) = \emptyset \) and \( F(\text{Subs}) = F(\text{ForeignSubs}) = \{\text{next}\} \).

Our expressions and commands are normalised versions of Java’s. Only syntactically distinct variables can be actual parameters of a method call (this is just a form of normalisation and does not prevent them being bound
to shared data-structures at run-time); in assignments, leftvalues are either a variable or the field of a variable; conditionals only check equality or nullness of variables; loops, such as the while commands in Section 1, are implemented through recursion. Note that these simplifying assumptions may be relaxed without affecting subsequent results.

**Definition 4.3** Expressions and commands are \( exp ::= \text{null} \mid \text{new} \ k \mid v \mid v.f \mid v.m(v_1, \ldots, v_n) \) and \( \text{com} ::= v ::= \text{exp} \mid v.f ::= \text{exp} \mid \{ \text{com}; \cdots; \text{com} \} \mid \text{if } v = w \text{ then } \text{com} \mid \text{else } \text{com} \mid \text{if } v = \text{null} \text{ then } \text{com} \mid \text{else } \text{com} \), where \( k \in K \) and \( v, w, v_1, \ldots, v_n \in V \) are distinct. Each method \( \kappa.m \) is defined in class \( \kappa \) with a statement \( \kappa_0 \ m(w_1 : \kappa_1, \ldots, w_n : \kappa_n) \) with \( w_{n+1} : \kappa_{n+1}, \ldots, w_{n+m} : \kappa_{n+m} \) is \( \text{com} \), where \( w_1, w_n, w_{n+1}, \ldots, w_{n+m} \in V \) are distinct, not in \( \{ \text{out}, \text{res}, \text{this} \} \) and have type \( \kappa_1, \ldots, \kappa_n, \kappa_{n+1}, \ldots, \kappa_{n+m} \in \mathcal{K} \), respectively. Variables \( w_1, \ldots, w_n \) are the formal parameters of the method, \( w_{n+1}, \ldots, w_{n+m} \) are its local variables. The method also uses a variable \( \text{out} \) of type \( \kappa_0 \) to store its return value. Let \( \text{body}(\kappa.m) = \text{com} \), \( \text{input}(\kappa.m) = \{ \text{this}, w_1, \ldots, w_n \} \), \( \text{output}(\kappa.m) = \{ \text{out} \} \), \( \text{locals}(\kappa.m) = \{ w_{n+1}, \ldots, w_{n+m} \} \) and finally \( \text{scope}(\kappa.m) = \text{input}(\kappa.m) \cup \text{output}(\kappa.m) \cup \text{locals}(\kappa.m) \).

**Example 4.4** For \textit{ForeignSubs.foreigners} in Figure 1 (just \textit{foreigners} below) we have \( \text{input}(\text{foreigners}) = \{ \text{this} \} \), \( \text{output}(\text{foreigners}) = \{ \text{out} \} \) and \( \text{locals}(\text{foreigners}) = \{ \text{temp} \} \).

**Semantics.**

We use a denotational semantics, hence compositional, in the style of [21]. However, we use a more complex notion of state, which assumes an infinite set of locations. As we assume a denotational semantics, a state has a single frame, rather than an activation stack of frames.

A frame binds variables (identifiers) to locations or \text{null}. A memory binds such locations to objects, which contain a class tag and the frame for their fields.

**Definition 4.5** Let \( \text{Loc} \) be an infinite set of locations. We define frames, objects and memories as\( \text{Frame}_V = \{ \phi \mid \phi \in V \rightarrow \text{Loc} \cup \{ \text{null} \} \} \), \( \text{Obj} = \{ \kappa*\phi \mid k \in \mathcal{K}, \phi \in \text{Frame}_{F(k)} \} \) and \( \text{Memory} = \{ \mu \in \text{Loc} \rightarrow \text{Obj} \mid \text{dom}(\mu) \text{ is finite} \} \). A new object of class \( k \) is \( \text{new}(k) = \kappa*\phi \), with \( \phi(f) = \text{null} \) for each \( f \in F(k) \).

**Example 4.6** Figure 2 shows a frame \( \phi \) and a memory \( \mu \). Different occurrences of the same location are linked. For instance, variable \textit{this} is bound to location \( l_1 \) and \( \mu(l_1) \) is a \textit{ForeignSubs} object. Objects are shown as boxes in memory with a class tag and a local frame mapping fields to locations or \text{null}.  

![Fig. 2 State (frame \( \phi \star \text{memory } \mu \)).](image-url)
A computation state is a frame and a memory with no dangling pointers.

**Definition 4.7** Let $V$ be the set of variables in scope at a given program point $p$. The set of possible states at $p$ is

$$
\Sigma_V = \left\{ \phi \times \mu \middle| \phi \in \text{Frame}_V, \mu \in \text{Memory}, \text{rng}(\phi) \subseteq \text{dom}(\mu) \right. \\
\text{for all } l \in \text{dom}(\mu) \text{ we have } \text{rng}(\mu(l), \phi) \subseteq \text{dom}(\mu) \left. \right\}.
$$

**Example 4.8** The state $\phi \times \mu \in \Sigma_{\{\text{temp, this, out}\}}$ in Figure 2 might be that of an interpreter at program point $*$ in Figure 1.

*Denotations* are the input/output semantics of a piece of code. *Interpretations* provide a denotation to each method.

**Definition 4.9** A denotation from $V$ to $V'$ is a partial map from $V$ to $V'$. The set of denotations from $V$ to $V'$ is $V;V'$. 

**Definition 4.10** An interpretation $I$ maps methods to denotations, such that $I(\kappa.m) : \Sigma_{\text{input}(\kappa.m)} \rightarrow \Sigma_{\text{output}(\kappa.m)}$ for each method $\kappa.m$.

Let $V$ be a set of variables with $\text{res} \notin V$. Let $I$ be an interpretation. In [19], this is defined as the denotation for expressions $E^I[.] : \text{exp} \mapsto (\Sigma_V \rightarrow \Sigma_{V \cup \{\text{res}\}})$ and the denotation for commands $C^I[.] : \text{com} \mapsto (\Sigma_V \rightarrow \Sigma_V)$. We only discuss them informally here. Expressions in our language have side-effects and return a value. Hence their denotations are partial maps from an initial to a final state containing a distinguished variable $\text{res} \notin V$ holding the expression’s value: $E^I[.] : \text{exp} \mapsto (\Sigma_V \rightarrow \Sigma_{V \cup \{\text{res}\}})$, where $I$ is an interpretation. Namely, given an input state $\phi \times \mu$, the denotation of $\text{null}$ binds $\text{res}$ to $null$ in $\phi$. That of $\text{new } \kappa$ binds $\text{res}$ to a new location bound to a new object of class $\kappa$. That of $v$ copies $v$ into $\text{res}$. That of $v.f$ accesses the object $o = \mu(\phi(v))$ bound to $v$ (provided $\phi(v) \neq \text{null}$) and copies the field $f$ of $o$ (i.e., $o.\phi(f)$) into $\text{res}$. That of method call uses the dynamic class of the receiver to fetch the denotation of the method from $I$. It plugs it in the calling context, by building a starting state $\sigma^+$ for the method, whose formal parameters (including $\text{this}$) are bound to the actual parameters.

The denotation of a command is a partial map from an initial to a final state: $C^I[.] : \text{com} \mapsto (\Sigma_V \rightarrow \Sigma_V)$. The denotation of $v := \text{exp}$ uses that of $\text{exp}$ to get a state where $\text{res}$ holds $\text{exp}$’s value. Then it copies $\text{res}$ into $v$ and removes $\text{res}$. Similarly for $v.f := \text{exp}$, but $\text{res}$ is copied into the field $f$ of the object bound to $v$, if any. The denotation of the conditionals checks their guard and then uses the denotation of $\text{then}$ or that of $\text{else}$. The denotation of a sequence of commands is the functional composition of their denotations.

The concrete denotational semantics of a program is the least fixpoint of the following transformer of interpretations, which corresponds to the immediate consequence operator of logic programming [2]. It evaluates the methods’ bodies in $I$, expanding the input state with local variables bound to $\text{null}$. It restricts the final state to $\text{out}$, so that Definition 4.10 is respected.
Definition 4.11 The following transformer on interpretations transforms an interpretation \( I \) into a new interpretation \( I' \) such that \( I'(\kappa . \mu) \) is

\[
[\lambda (\phi * \mu) \in \Sigma_{\text{input}(\kappa . \mu)} . (\phi[\text{out} \mapsto \text{null}, w_{n+1} \mapsto \text{null}, \ldots, w_{n+m} \mapsto \text{null}] * \mu)] \circ \\
\circ C' [\text{body}(\kappa . \mu)] \circ [\lambda (\phi * \mu) \in \Sigma_{\text{scope}(\kappa . \mu)} . (\phi[\text{out} * \mu])]
\]

The denotational semantics of a program is the least fixpoint of this transformer.

We introduce now a notion of reachability for locations. The reachable locations are those bound to the variables or to their fields or to the fields of the fields, and so on.

Definition 4.12 Let \( \mu \in \text{Memory} \) and \( l \in \text{dom}(\mu) \). We define the set of locations reachable from \( l \) in \( \mu \) as \( L(\mu)(l) = \bigcup \{ L^i(\mu)(l) \mid i \geq 0 \} \), where \( L^0(\mu)(l) = \text{rng}(\mu(l), \phi) \cap \text{Loc} \) and \( L^{i+1}(\mu)(l) = \bigcup \{ \text{rng}(\mu(l'), \phi) \cap \text{Loc} \mid l' \in L^i(\mu)(l) \} \). Let \( \phi * \mu \in \Sigma_V \) and \( v \in V \). We define

\[
L_V(\phi * \mu)(v) = \begin{cases} \\
\varnothing & \text{if } \phi(v) = \text{null} \\
\{ \phi(v) \} \cup L(\mu)(\phi(v)) & \text{otherwise.} 
\end{cases}
\]

Example 4.13 Consider the state \( \phi * \mu \) in Figure 2 and let \( V = \{ \text{temp}, \text{this}, \text{out} \} \). We have \( L^0(\mu)(l_1) = \{ l_2 \} \) and, for every \( i \geq 0 \), \( L^{i+1}(\mu)(l_1) = \varnothing \), \( L^i(\mu)(l_2) = \varnothing \) and \( L^i(\mu)(l_3) = \varnothing \). Hence \( L_V(\phi * \mu)(\text{temp}) = \varnothing \), \( L_V(\phi * \mu)(\text{this}) = \{ l_1, l_2 \} \) and \( L_V(\phi * \mu)(\text{out}) = \{ l_3 \} \).

Definition 4.14 Two variables \( v_1, v_2 \in V \) share in \( \phi * \mu \in \Sigma_V \) if there is a location which is reachable from both i.e., if \( L_V(\phi * \mu)(v_1) \cap L_V(\phi * \mu)(v_2) \neq \varnothing \) [17]. A variable \( v \in V \) is cyclical in \( \phi * \mu \) if \( \phi(v) \neq \text{null} \) and there exists \( l \in L(\mu)(\phi(v)) \) such that \( l \in L(\mu)(l) \) [16].

Example 4.15 From Example 4.13, we conclude that, in the state \( \phi * \mu \) in Figure 2, variable \( \text{this} \) shares with \( \text{this} \) itself and \( \text{out} \) with \( \text{out} \) itself. No variable shares with \( \text{temp} \). No variable is cyclical.

Sharing and cyclicity of program variables can be computed through shape analysis [20] or through some lighter, more specialised static analyses such as [17,16].

We use reachability (Definition 4.12) to refine Definition 4.10. We require that a method does not write into the locations \( L \) of the input state which are not reachable from the formal parameters. Programming languages such as Java and that of Section 4 satisfy this constraint. It is needed to prove the correctness of the abstract counterpart of method call (Figure 3).

Definition 4.16 We refine Definition 4.10 of interpretations \( I \) by requiring that, if \( I(\kappa . \mu)(\phi * \mu) = \phi' * \mu' \) and \( L = \text{dom}(\mu) \setminus \bigcup \{ L_{\text{input}}(\kappa . \mu)(\phi * \mu)(v) \mid v \in \text{input}(\kappa . \mu) \} \), then \( \mu|_L = \mu'|_L \) (unreachable locations are not modified).
5 Path-Length Analysis

The path-length of a variable $v$ in a state $\sigma$ is the length of the longest chain of pointers you can follow from $v$ in $\sigma$.

**Definition 5.1** Let $\phi \ast \mu \in \Sigma_V$ and $v \in V$. We define

\[
\begin{align*}
    \text{len}^0(\phi \ast \mu)(v) &= 0 \\
    \text{len}^{i+1}(\phi \ast \mu)(v) &= \begin{cases} 
        0 & \text{if } \phi(v) = \text{null} \\
        1 + \max \set{\text{len}^i(o.\phi \ast \mu)(f) \mid o = \mu\phi(v) \text{ and } f \in \text{dom}(o.\phi)} & \text{otherwise.}
    \end{cases}
\end{align*}
\]

The path-length of $v$ in $\phi \ast \mu$ is defined as $\text{len}(\phi \ast \mu)(v) = \lim_{i \to \infty} \text{len}^i(\phi \ast \mu)(v)$. Note that $\text{len}(\phi \ast \mu)(v) \in \{0, 1, 2, \ldots, \infty\}$.

**Example 5.2** In Figure 2 we have $\text{len}(\phi \ast \mu) = [\text{temp} \mapsto 0, \text{this} \mapsto 2, \text{out} \mapsto 1]$.

**Definition 5.3** A path-length relational constraint from $V$ to $V'$ is an integer linear constraint over the input variables $\{\hat{v} \mid v \in V\}$ and the output variables $\{\hat{v} \mid v \in V'\}$, which uses the predicates $\leq$ and $<$. The set of such constraints is $\text{PL}_{V;V'}$, with a least upper bound operation $\sqcup$ defined as the convex hull. The path length relational constraint $U(V) = \wedge \{\hat{v} = \hat{v} \mid v \in V\} \in \text{PL}_{V;V}$ is called the frame condition for $V$.

**Example 5.4** Consider $V = \{\text{temp}, \text{this}, \text{out}\}$. A path-length relational constraint in $\text{PL}_{V;V}$ is $\text{temp} < \text{temp} \land \text{this} \leq \text{this} \land \text{this} \leq \text{this} \land \text{out} \leq \text{this} + \text{out} + 1$.

In the following, we also use $v = w$ in the constraints, which is syntactical sugar for $v \leq w$ and $w \geq v$.

**Definition 5.5** Let $\sigma \in \Sigma_V$. We define $\text{len}(\sigma) = [\hat{v} \mapsto \text{len}(\sigma)(v) \mid v \in V]$ and $\hat{\text{len}}(\sigma) = [\hat{v} \mapsto \text{len}(\sigma)(v) \mid v \in V]$.

The concretisation of a path-length relational constraint is the set of denotations which satisfy the path-length relationship expressed by the constraint.

**Definition 5.6** The concretisation of a constraint $pl \in \text{PL}_{V;V'}$ is

\[
\gamma(pl) = \set{\delta \in \Delta_{V;V'} \mid \text{for all } \sigma \in \Sigma_V \text{ if } \delta(\sigma) \text{ is defined then } \hat{\text{len}}(\sigma) \cup \hat{\text{len}}(\delta(\sigma)) \models pl}.
\]

**Example 5.7** Consider $V = \{\text{temp}, \text{this}, \text{out}\}$ as in Figure 2 and the denotation $\delta \in \Delta_{V;V}$ such that

\[
\delta(\phi \ast \mu) = \begin{cases} 
    \phi[\text{temp} \mapsto \text{null}] \ast \mu & \text{if } \phi(\text{temp}) \neq \text{null} \\
    \text{undefined} & \text{if } \phi(\text{temp}) = \text{null}.
\end{cases}
\]
We have \( \delta \in \gamma(pl) \) where \( pl \) is the constraint in Example 5.4.

The map \( \gamma \) of Definition 5.6 maps the elements of \( \text{PL}_{V,V'} \) into elements of the concrete domain \( \varphi(\Delta_{V,V'}) \). The abstract domain \( \text{PL}_{V,V'} \) is closed w.r.t. \( \wedge \) and represents the top of \( \varphi(\Delta_{V,V'}) \) as the empty, tautological constraint \( \text{true} \). Hence it is an abstract domain having \( \gamma \) as its concretisation map (Section 3).

Two path-length relational constraints \( pl_1 \) and \( pl_2 \) are composed by matching the output variables of \( pl_1 \) with the input variables of \( pl_2 \) and then projecting away such variables.

**Definition 5.8** Let \( pl_1 \in \text{PL}_{V',V} \) and \( pl_2 \in \text{PL}_{V',V''} \). We define their composition \( pl_1 \circ pl_2 \in \text{PL}_{V',V''} \) as \( \exists \{ \bar{v} \in \overline{V} \} (pl_1[\bar{v} \mapsto \overline{v}] \land \overline{pl_2}[\bar{v} \mapsto \overline{v} | v \in V]) \).

**Definition 5.9** A path-length interpretation \( J \) maps each method \( \kappa.m \) to a path-length relational constraint \( J(\kappa.m) \in \text{PL}_{\text{input}(\kappa.m), \text{output}(\kappa.m)} \).

We can now define an abstract semantics as a compilation of the source program into a program over path-length relational constraints [9].

**Definition 5.10** Let \( V \) be a set of variables in scope with \( \text{res} \notin V \). Figure 3 provides an abstract semantics over path-length relational constraints, which corresponds to the concrete semantics of [19].

Let us consider the constraints in Figure 3. The evaluation of the expressions \( \text{null}, \text{new} \kappa, v \) and \( v.f \) does not modify the path-length of the variables. This is expressed by the use of the frame condition \( U(V) \). The path-length of the value of those expressions is 0 for \( \text{null} \), 1 for \( \text{new} \kappa \) (since the fields of the newly-created objects are all bound to \( \text{null} \)) and the path-length of variable \( v \) for the expression \( v \). The path-length of the value of \( v.f \) is no longer than that of \( v \), but the strict inequality can be assumed only when \( v \) is non-cyclical.

The constraint for method call says that the variables which do not share with the parameters are not affected by the call. Moreover, the path-length of the result is computed by using the current approximation \( J(\kappa.m) \) for all the methods \( \kappa.m \) which might be called. The constraints for the assignments are the composition of the constraint for their right-hand side with a constraint which modifies a variable or field, respectively. The constraints for the conditionals are the convex hull of the constraints for the two branches of the conditionals. Precision is improved by taking into account the fact that if the guard \( v = w \) holds then variables \( v \) and \( w \) have the same path-length. Similarly, if the guard \( v = \text{null} \) holds then \( v \) has a path-length equal to 0; if it does not hold, \( v \) has a positive path-length. The sequential composition of denotations becomes composition of constraints in Figure 3.

The abstract immediate consequence operator reflects the fact that initially all local variables are bound to \( \text{null} \), and that only \( \text{out} \) is observable at the end of a method (Definition 4.11).
\[ \text{PLC}^J[\text{null}] = U(V) \land \text{res} = 0 \]
\[ \text{PLC}^J[\text{new } \kappa] = U(V) \land \text{res} = 1 \]
\[ \text{PLC}^J[v] = U(V) \land \text{res} = \bar{v} \]
\[ \text{PLC}^J[v.f] = \begin{cases} 
U(V) \land \text{res} < \bar{v} & \text{if } v \text{ is not cyclical} \\
(U(V) \land \text{res} \leq \bar{v}) & \text{otherwise}
\end{cases} \]

\[ \text{PLC}^J[v.m(v_1, \ldots, v_n)] = U(\text{NS}) \land \bigcup \left\{ J(\kappa.m) \begin{bmatrix} \text{this} \mapsto \bar{v}, \text{out} \mapsto \text{res} \\
\bar{w}_1 \mapsto \bar{v}_1, \ldots, \bar{w}_n \mapsto \bar{v}_n \end{bmatrix} : \kappa.m \text{ can be called here} \right\} \]

where \( \text{NS} = \{ x \in V \mid x \text{ does not share with any of } v, v_1, \ldots, v_n \} \)

\[ \text{PLC}^J[v := \text{exp}] = \text{PLC}^J[\text{exp}] \circ (U(V \setminus v) \land \bar{v} = \text{res}) \]
\[ \text{PLC}^J[v.f := \text{exp}] = \text{PLC}^J[\text{exp}] \circ \text{setField}^{v.f} \]
\[ \text{setField}^{v.f} = \begin{cases} \land \{ \bar{x} \leq \bar{x} + \text{res} \mid x \in S \} \land (U(V \setminus S)) & \text{if } v \text{ and res do not share} \\
U(V \setminus S) & \text{otherwise}
\end{cases} \]

and \( S = \{ x \in V \mid x \text{ shares with } v \} \)

\[ \text{PLC}^J[\text{if } v = w \text{ then } \text{com}_1 \text{ else } \text{com}_2] = (\bar{v} = \bar{w} \land \text{PLC}^J[\text{com}_1]) \cup \text{PLC}^J[\text{com}_2] \]
\[ \text{PLC}^J[\text{if } v = \text{null} \text{ then } \text{com}_1 \text{ else } \text{com}_2] = (\bar{v} = 0 \land \text{PLC}^J[\text{com}_1]) \cup (0 < \bar{v} \land \text{PLC}^J[\text{com}_2]) \]
\[ \text{PLC}^J[\{\} ] = U(V) \]
\[ \text{PLC}^J[\{ \text{com}_1; \ldots; \text{com}_p \}] = \text{PLC}^J[\text{com}_1] \circ \cdots \circ \text{PLC}^J[\text{com}_p]. \]

Fig. 3. The abstract path-length semantics of our language, assuming that \( V \) is the set of variables in scope in the method under analysis.

forms each \( J \) into a new path-length interpretation \( J' \) such that \( J'(\kappa.m) \) is

\[ [U(\text{input}(\kappa.m)) \land \{ \bar{w}_{n+i} = 0 \mid 1 \leq i \leq m \} \land \text{out} = 0] \circ \text{PLC}^J[\text{body}(\kappa.m)] \circ [\text{out} = \hat{\text{out}}]. \]

**Theorem 5.12** The abstract semantics of Figure 3 is correct w.r.t. the concrete semantics in [19]. Namely, if \( F^C \) is the least fixpoint of the transformer of Definition 4.11 and \( F^A \) is the least fixpoint of the transformer of Definition 5.11, we have \( F^C \in \gamma(F^A). \)

13
6 Conclusion

We have defined a static analysis for path-length analysis of imperative, object-oriented programs. We think that this is the first definition of a static analysis meant to support automatic termination analysis of imperative programs dealing with dynamically allocated data-structures.

An implementation is still missing. It is important in order to show how precise our analysis is, and how well it scales to real programs.

References


