Path-Length Analysis for Object-Oriented Programs

Fausto Spoto [1]

Dipartimento di Informatica, Università di Verona, Italy

Patricia M. Hill [2]

School of Computing, University of Leeds, United Kingdom

Étienne Payet [3]

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 [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.

[1] Email: `fausto.spoto@univr.it`

[2] Email: `hill@comp.leeds.ac.uk`

[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:

$$\texttt{while}\ (s\ !=\texttt{null})\ \{s := s.\texttt{next};\}$$

Let us write $\check{s}$ for the path-length of s *before* the command s:=s.next is executed and $\hat{s}$ for its path-length *afterwards*. We call $\check{s}$ an *input* variable and $\hat{s}$ an *output* variable. A condition which lets us prove termination of the loop is $\check{s} > \hat{s}$. 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.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:

$$\texttt{while}\ (s\ !=\texttt{null})\ \{s := s.\texttt{foreigners}();\ s := s.\texttt{next};\}$$

The loop body now consists of two commands. The first, $s := s$.foreigners(), satisfies the constraint $\check{s} \geq \hat{s}$ (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 $s := s$.next is compiled into the constraint $\check{s} > \hat{s}$, as before, so that the compilation of the body of the loop is now the *composition* $(\check{s} \geq \hat{s}) \circ (\check{s} > \hat{s})$. 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 $\check{s} > \hat{s}$ 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 $V = \{temp, this, out\}$. In this example, we want to prove that both foreigners methods in Figure 1 satisfy the invariant $\widehat{out} \leq \widecheck{this}$, 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.

$$\big(U(V) \wedge \hat{res} < \check{this}\big) = \begin{pmatrix} \hat{out} = \check{out} \wedge \hat{temp} = \check{temp} \wedge \\ \hat{this} = \check{this} \wedge \hat{res} < \check{this} \end{pmatrix}. \tag{1}$$

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

$$\begin{aligned} &\begin{pmatrix} \hat{out} = \check{out} \wedge \hat{temp} = \check{temp} \wedge \\ \hat{this} = \check{this} \wedge \hat{res} < \check{this} \end{pmatrix} \circ \big(U(V \setminus temp) \wedge \hat{temp} = \check{res}\big) \\ &= \begin{pmatrix} \hat{out} = \check{out} \wedge \hat{temp} = \check{temp} \wedge \\ \hat{this} = \check{this} \wedge \hat{res} < \check{this} \end{pmatrix} \circ \begin{pmatrix} \hat{out} = \check{out} \wedge \hat{this} = \check{this} \wedge \\ \hat{temp} = \check{res} \end{pmatrix}. \end{aligned} \tag{2}$$

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

$$\begin{aligned} &\exists_{\{\overline{out},\overline{res},\overline{temp},\overline{this}\}} \begin{pmatrix} \overline{out} = \check{out} \wedge \overline{temp} = \check{temp} \wedge \overline{this} = \check{this} \wedge \overline{res} < \check{this} \wedge \\ \hat{out} = \overline{out} \wedge \hat{this} = \overline{this} \wedge \hat{temp} = \overline{res} \end{pmatrix} \\ &\qquad = (\hat{out} = \check{out} \wedge \hat{this} = \check{this} \wedge \hat{temp} < \check{this}). \end{aligned} \tag{3}$$

Equation (3) says that, by assigning *this*.`next` to *temp*, the final path-length

of *temp* is strictly smaller than the initial path-length of *this*.

Consider now the abstraction of *temp*.`foreigners`(). 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

$$(\hat{out} = \check{out}) \wedge \underbrace{\begin{pmatrix} J(\texttt{Subs.foreigners}) \sqcup \\ J(\texttt{ForeignSubs.foreigners}) \end{pmatrix}}_{\iota} \begin{bmatrix} \check{this} \mapsto \check{temp}, \\ \hat{out} \mapsto \hat{res} \end{bmatrix}. \quad (4)$$

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

$$\begin{pmatrix} 0 = \check{temp} \wedge \hat{temp} = \check{temp} \wedge \\ \hat{this} = \check{this} \wedge \hat{out} = \check{out} \end{pmatrix} \sqcup (0 < \check{temp} \wedge \iota[\check{this} \mapsto \check{temp}])$$

$$= \begin{pmatrix} 0 = \check{temp} = \hat{temp} \wedge \\ \hat{this} = \check{this} \wedge \hat{out} = \check{out} \end{pmatrix} \sqcup (0 < \check{temp} \wedge \iota[\check{this} \mapsto \check{temp}]). \quad (5)$$

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

$$(\check{out} = \check{temp} = 0) \wedge \begin{bmatrix} ((3) \circ (0 = \check{temp} = \hat{temp} \wedge \hat{this} = \check{this} \wedge \hat{out} = \check{out})) \sqcup \\ ((3) \circ (0 < \check{temp} \wedge \iota[\check{this} \mapsto \check{temp}])) \end{bmatrix} \circ (\check{out} = \hat{out})$$

which we simplify into

$$(\check{out} = \check{temp} = 0) \wedge \begin{bmatrix} (0 = \hat{temp} \wedge 0 < \check{this} \wedge \hat{this} = \check{this} \wedge \hat{out} = \check{out}) \sqcup \\ \exists_{\overline{temp}} (0 < \overline{temp} < \check{this} \wedge \iota[\check{this} \mapsto \overline{temp}]) \end{bmatrix} \circ (\check{out} = \hat{out}). \quad (6)$$

Consider `ForeignSubs.foreigners` now. We abstract its first two lines in

$$\hat{out} = 1 \wedge \hat{this} = \check{this} \wedge \hat{temp} < \check{this}. \quad (7)$$

The abstraction of the call $\mathit{temp}.\texttt{foreigners}()$ is as in Equation (4). To abstract $\mathit{out}.\texttt{next} := \mathit{temp}.\texttt{foreigners}()$, we use the rule for $v.\texttt{f} := \mathit{exp}$ in Figure 3, with $S = \{\mathit{out}\}$. Hence the abstraction of this field update is

$$(4) \circ (\hat{\mathit{out}} \leq \check{\mathit{out}} + \check{\mathit{res}} \wedge \hat{\mathit{this}} = \check{\mathit{this}} \wedge \hat{\mathit{temp}} = \check{\mathit{temp}}) \\ = \exists_{\overline{\mathit{res}}} \left[\iota[\check{\mathit{this}} \mapsto \check{\mathit{temp}}, \hat{\mathit{out}} \mapsto \overline{\mathit{res}}] \wedge \hat{\mathit{out}} \leq \check{\mathit{out}} + \overline{\mathit{res}} \right]. \quad (8)$$

The abstraction of the conditional is similar to that seen before. We do not need to abstract $\mathit{out}.\texttt{channels} := \mathit{this}.\texttt{channels}$ 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:

$$(\check{\mathit{out}} = \check{\mathit{temp}} = 0) \wedge \\ \left\{ (7) \circ \left[(0 = \check{\mathit{temp}} = \hat{\mathit{temp}} \wedge \hat{\mathit{this}} = \check{\mathit{this}} \wedge \hat{\mathit{out}} = \check{\mathit{out}}) \sqcup (0 < \check{\mathit{temp}} \wedge (8)) \right] \right\} \\ \circ (\check{\mathit{out}} = \hat{\mathit{out}}). \quad (9)$$

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

$$J(\texttt{Subs.foreigners}) = J(\texttt{ForeignSubs.foreigners}) = \iota = (\hat{\mathit{out}} \leq \check{\mathit{this}})$$

into Equations (6) and (9). From the first equation we get

$$(\check{\mathit{out}} = \check{\mathit{temp}} = 0) \wedge \\ \left[\begin{array}{c} (0 = \hat{\mathit{temp}} \wedge 0 < \check{\mathit{this}} \wedge \hat{\mathit{this}} = \check{\mathit{this}} \wedge \hat{\mathit{out}} = \check{\mathit{out}}) \sqcup \\ \exists_{\overline{\mathit{temp}}} (0 < \overline{\mathit{temp}} < \check{\mathit{this}} \wedge \hat{\mathit{out}} \leq \overline{\mathit{temp}}) \end{array} \right] \circ (\check{\mathit{out}} = \hat{\mathit{out}})$$

which entails

$$\left\{ (\check{\mathit{out}} = 0) \wedge \left[\left(\begin{array}{c} 0 = \hat{\mathit{temp}} \wedge 0 < \check{\mathit{this}} \wedge \\ \hat{\mathit{this}} = \check{\mathit{this}} \wedge \hat{\mathit{out}} = \check{\mathit{out}} \end{array} \right) \sqcup (\hat{\mathit{out}} < \check{\mathit{this}}) \right] \right\} \circ (\check{\mathit{out}} = \hat{\mathit{out}})$$

which itself entails

$$\left\{ (0 = \hat{\mathit{temp}} \wedge 0 < \check{\mathit{this}} \wedge \hat{\mathit{this}} = \check{\mathit{this}} \wedge \hat{\mathit{out}} = 0) \sqcup (\hat{\mathit{out}} < \check{\mathit{this}}) \right\} \circ (\check{\mathit{out}} = \hat{\mathit{out}})$$

and, finally, $\left\{ (\hat{\mathit{out}} < \check{\mathit{this}}) \sqcup (\hat{\mathit{out}} < \check{\mathit{this}}) \right\} \circ (\check{\mathit{out}} = \hat{\mathit{out}}) = (\hat{\mathit{out}} < \check{\mathit{this}})$. Note that $\hat{\mathit{out}} < \check{\mathit{this}}$ entails the invariant $\hat{\mathit{out}} \leq \check{\mathit{this}}$.

For the second one, we first reduce (8) to $\exists_{\overline{\mathit{res}}} (\overline{\mathit{res}} \leq \check{\mathit{temp}} \wedge (\hat{\mathit{out}} \leq \check{\mathit{out}} + \overline{\mathit{res}})) = (\hat{\mathit{out}} \leq \check{\mathit{out}} + \check{\mathit{temp}})$. Hence (9) is

$$(\check{\mathit{out}} = \check{\mathit{temp}} = 0) \wedge \left\{ (7) \circ \left[(0 = \check{\mathit{temp}} = \hat{\mathit{temp}} \wedge \hat{\mathit{this}} = \check{\mathit{this}} \wedge \hat{\mathit{out}} = \check{\mathit{out}}) \sqcup \\ (0 < \check{\mathit{temp}} \wedge \hat{\mathit{out}} \leq \check{\mathit{out}} + \check{\mathit{temp}}) \right] \right\} \circ (\check{\mathit{out}} = \hat{\mathit{out}})$$

which entails

$$
\begin{aligned}
&(\check{out} = \check{temp} = 0) \wedge \left\{(7) \circ \left[\hat{out} \le \check{out} + \check{temp}\right]\right\} \circ (\check{out} = \hat{out}) \\
&= (\check{out} = \check{temp} = 0) \wedge \exists_{\overline{temp}}\left\{\overline{temp} < \check{this} \wedge \hat{out} \le 1 + \overline{temp}\right\} \circ (\check{out} = \hat{out}) \\
&\qquad = (\check{out} = \check{temp} = 0 \wedge \hat{out} \le \check{this}) \circ (\check{out} = \hat{out}) = (\hat{out} \le \check{this}).
\end{aligned}
$$

3 Preliminaries

A total (partial) function f is denoted by $\mapsto$ ($\rightarrow$). The *domain* (*codomain*) of f is $dom(f)$ ($rng(f)$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $dom(f) = \{v_1, \ldots, v_n\}$ 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 $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq dom(f)$ (to $dom(f) \setminus s$). 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 gf. The components of a *pair* are separated by $\star$. A definition of S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$.

A *poset* $S \star \le$ is a set S with a reflexive, transitive and antisymmetric relation $\le$. If $C \star \le$ and $A \star \preceq$ 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.,* $\alpha(c_1) \preceq \alpha(c_2)$ if $c_1 \le c_2$, similarly for γ) such that $\gamma\alpha$ is extensive (*i.e.,* $c \le \gamma\alpha(c)$ for any $c \in C$) and $\alpha\gamma$ is reductive (*i.e.,* $\alpha\gamma(a) \le a$ 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*) $\mathcal{V}$ (including *res*, *out*, *this*) and a finite poset of *classes* (or *types*) $\mathcal{K}\star\le$ ordered by a *subclass relation* $\le$. We write $F(\kappa)$ for the set of fields of class $\kappa \in \mathcal{K}$.

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

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 $exp ::= \texttt{null} \mid \texttt{new}\ \kappa \mid v \mid v.\texttt{f} \mid v.\texttt{m}(v_1,\ldots,v_n)$ and $com ::= v := exp \mid v.\texttt{f} := exp \mid \{com;\cdots;com\} \mid \texttt{if}\ v = w\ \texttt{then}\ com\ \texttt{else}\ com \mid \texttt{if}\ v = \texttt{null}\ \texttt{then}\ com\ \texttt{else}\ com$, where $\kappa \in \mathcal{K}$ and $v, w, v_1, \ldots, v_n \in \mathcal{V}$ are distinct. Each method $\kappa.\texttt{m}$ is *defined* in class κ with a statement $\kappa_0\ \texttt{m}(w_1 : \kappa_1, \ldots, w_n : \kappa_n)\ \texttt{with}\ w_{n+1} : \kappa_{n+1}, \ldots, w_{n+m} : \kappa_{n+m}\ \texttt{is}\ com$, where $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ are distinct, not in $\{out, res, this\}$ 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 *out* of type κ_0 to store its *return value*. Let $body(\kappa.\texttt{m}) = com$, $input(\kappa.\texttt{m}) = \{this, w_1, \ldots, w_n\}$, $output(\kappa.\texttt{m}) = \{out\}$, $locals(\kappa.\texttt{m}) = \{w_{n+1}, \ldots, w_{n+m}\}$ and finally $scope(\kappa.\texttt{m}) = input(\kappa.\texttt{m}) \cup output(\kappa.\texttt{m}) \cup locals(\kappa.\texttt{m})$.

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

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 *null*. A memory binds such locations to objects, which contain a class tag and the frame for their fields.

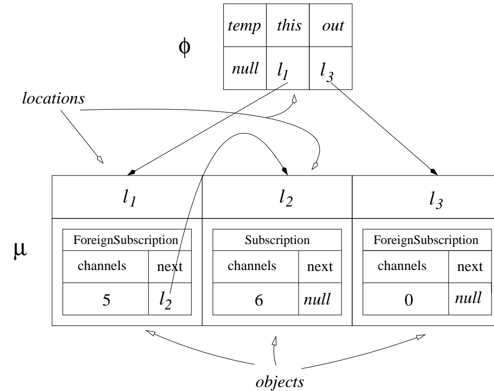

Fig. 2 State (frame $\phi \star$ memory μ).

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

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

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

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

Example 4.8 The state $\phi \star \mu \in \Sigma_{\{temp,this,out\}}$ 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 Σ_V to $\Sigma_{V'}$. The set of denotations from V to V' is $\Delta_{V,V'}$.

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

Let V be a set of variables with $res \notin V$. Let I be an interpretation. In [19], this is defined as the *denotation for expressions* $\mathcal{E}^I[\![_]\!] : exp \mapsto (\Sigma_V \to \Sigma_{V\cup\{res\}})$ and the *denotation for commands* $\mathcal{C}^I[\![_]\!] : com \mapsto (\Sigma_V \to \Sigma_V)$. 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 $res \notin V$ holding the expression's value: $\mathcal{E}^I[\![_]\!] : exp \mapsto (\Sigma_V \to \Sigma_{V\cup\{res\}})$, where I is an interpretation. Namely, given an input state $\phi\star\mu$, the denotation of `null` binds res to *null* in ϕ. That of `new` κ binds res to a new location bound to a new object of class κ. That of v copies v into res. That of $v.\mathtt{f}$ accesses the object $o = \mu(\phi(v))$ bound to v (provided $\phi(v) \neq null$) and copies the field `f` of o (*i.e.,* $o.\phi(\mathtt{f})$) into 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 $\sigma^\dagger$ for the method, whose formal parameters (including *this*) are bound to the actual parameters.

The denotation of a command is a partial map from an initial to a final state: $\mathcal{C}^I[\![_]\!] : com \mapsto (\Sigma_V \to \Sigma_V)$. The denotation of $v{:=}exp$ uses that of exp to get a state where res holds exp's value. Then it copies res into v and removes res. Similarly for $v.\mathtt{f} := exp$, but 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 `then` or that of `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 *null*. It restricts the final state to *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 $I'(\kappa.\mathtt{m})$ is

$$\left[\lambda(\phi\star\mu)\in\Sigma_{input(\kappa.\mathtt{m})}.(\phi[out\mapsto null, w_{n+1}\mapsto null,\ldots,w_{n+m}\mapsto null]\star\mu)\right]\circ \\ \circ\,\mathcal{C}^I[\![body(\kappa.\mathtt{m})]\!]\circ\left[\lambda(\phi\star\mu)\in\Sigma_{scope(\kappa.\mathtt{m})}.(\phi|_{out}\star\mu)\right].$$

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

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

Example 4.13 Consider the state $\phi\star\mu$ in Figure 2 and let $V=\{temp, this, out\}$. 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 $L_V(\phi\star\mu)(temp)=\varnothing$, $L_V(\phi\star\mu)(this)=\{l_1,l_2\}$ and $L_V(\phi\star\mu)(out)=\{l_3\}$.

Definition 4.14 Two variables $v_1,v_2\in V$ *share* in $\phi\star\mu\in\Sigma_V$ if there is a location which is reachable from both *i.e.,* if $L_V(\phi\star\mu)(v_1)\cap L_V(\phi\star\mu)(v_2)\neq\varnothing$ [17]. A variable $v\in V$ is *cyclical* in $\phi\star\mu$ if $\phi(v)\neq null$ 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 $\phi\star\mu$ in Figure 2, variable *this* shares with *this* itself and *out* with *out* itself. No variable shares with *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 $I(\kappa.\mathtt{m})(\phi\star\mu)=\phi'\star\mu'$ and $L=dom(\mu)\setminus\cup\{L_{input(\kappa.\mathtt{m})}(\phi\star\mu)(v)\mid v\in input(\kappa.\mathtt{m})\}$, then $\mu|_L=\mu'|_L$ (unreachable locations are not modified).

5 Path-Length Analysis

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

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

$$len^0(\phi \star \mu)(v) = 0$$

$$len^{i+1}(\phi \star \mu)(v) = \begin{cases} 0 & \text{if } \phi(v) = null \\ 1 + \max\left\{ len^i(o.\phi \star \mu)(f) \,\middle|\, \begin{array}{l} o = \mu\phi(v) \text{ and} \\ f \in dom(o.\phi) \end{array} \right\} & \text{otherwise.} \end{cases}$$

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

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

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

Example 5.4 Consider $V = \{temp, this, out\}$. A path-length relational constraint in $\mathsf{PL}_{V,V}$ is $\widehat{temp} < \widecheck{temp} \wedge \widehat{this} \leq \widecheck{this} \wedge \widecheck{this} \leq \widehat{this} \wedge \widehat{out} \leq \widecheck{this} + \widecheck{out} + 1$.

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 $\widecheck{len}(\sigma) = [\check{v} \mapsto len(\sigma)(v) \mid v \in V]$ and $\widehat{len}(\sigma) = [\hat{v} \mapsto len(\sigma)(v) \mid v \in V]$.

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 $pl \in \mathsf{PL}_{V,V'}$ is

$$\gamma(pl) = \left\{ \delta \in \Delta_{V,V'} \,\middle|\, \begin{array}{l} \text{for all } \sigma \in \Sigma_V \text{ if } \delta(\sigma) \text{ is defined} \\ \text{then } \widecheck{len}(\sigma) \cup \widehat{len}(\delta(\sigma)) \models pl \end{array} \right\}.$$

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

$$\delta(\phi \star \mu) = \begin{cases} \phi[temp \mapsto null] \star \mu & \text{if } \phi(temp) \neq null \\ undefined & \text{if } \phi(temp) = null. \end{cases}$$

We have $\delta \in \gamma(pl)$ where pl is the constraint in Example 5.4.

The map γ of Definition 5.6 maps the elements of $\mathsf{PL}_{V,V'}$ into elements of the concrete domain $\wp(\Delta_{V,V'})$. The abstract domain $\mathsf{PL}_{V,V'}$ is closed *w.r.t.* $\wedge$ and represents the top of $\wp(\Delta_{V,V'})$ as the empty, tautological constraint *true*. Hence it is an abstract domain having γ 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 $pl_1 \in \mathsf{PL}_{V',V}$ and $pl_2 \in \mathsf{PL}_{V,V''}$. We define their *composition* $pl_1 \circ pl_2 \in \mathsf{PL}_{V',V''}$ as $\exists_{\{\overline{v} \mid v \in V\}}(pl_1[\hat{v} \mapsto \overline{v} \mid v \in V] \wedge pl_2[\check{v} \mapsto \overline{v} \mid v \in V])$.

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

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 $res \notin V$. 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 `null`, `new` κ, v and $v.\mathtt{f}$ 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 `null`, 1 for `new` κ (since the fields of the newly-created objects are all bound to *null*) and the path-length of variable v for the expression v. The path-length of the value of $v.\mathtt{f}$ 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 $J(\kappa.\mathtt{m})$ for all the methods $\kappa.\mathtt{m}$ 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 $v = \mathtt{null}$ 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 *null*, and that only *out* is observable at the end of a method (Definition 4.11).

Definition 5.11 The abstract transformer on path-length interpretations trans-

$$\mathcal{PLE}^J[\![\texttt{null}]\!] = U(V) \wedge \hat{res} = 0$$
$$\mathcal{PLE}^J[\![\texttt{new } \kappa]\!] = U(V) \wedge \hat{res} = 1$$
$$\mathcal{PLE}^J[\![v]\!] = U(V) \wedge \hat{res} = \check{v}$$
$$\mathcal{PLE}^J[\![v.\texttt{f}]\!] = \begin{cases} U(V) \wedge \hat{res} < \check{v} & \text{if } v \text{ is not cyclical} \\ U(V) \wedge \hat{res} \leq \check{v} & \text{otherwise} \end{cases}$$
$$\mathcal{PLE}^J[\![v.\texttt{m}(v_1, \ldots, v_n)]\!] = U(NS) \wedge \sqcup \left\{ J(\kappa.\texttt{m}) \begin{bmatrix} \check{this} \mapsto \check{v}, \hat{out} \mapsto \hat{res} \\ \check{w}_1 \mapsto \check{v}_1, \ldots, \check{w}_n \mapsto \check{v}_n \end{bmatrix} \middle| \begin{array}{l} \kappa.\texttt{m} \text{ can be} \\ \text{called here} \end{array} \right\}$$
$$\text{where } NS = \{x \in V \mid x \text{ does not share with any of } v, v_1, \ldots, v_n\}$$

$$\mathcal{PLC}^J[\![v{:=}exp]\!] = \mathcal{PLE}^J[\![exp]\!] \circ (U(V \setminus v) \wedge \hat{v} = \check{res})$$
$$\mathcal{PLC}^J[\![v.\texttt{f} := exp]\!] = \mathcal{PLE}^J[\![exp]\!] \circ setField^{v.\texttt{f}}$$
$$\text{where } setField^{v.\texttt{f}} = \begin{cases} \wedge\{\hat{x} \leq \check{x} + \check{res} \mid x \in S\} \wedge U(V \setminus S) \\ \quad \text{if } v \text{ and } res \text{ do not share} \\ \\ U(V \setminus S) \\ \quad \text{otherwise} \end{cases}$$
$$\text{and } S = \{x \in V \mid x \text{ shares with } v\}$$
$$\mathcal{PLC}^J\left[\!\!\left[\begin{array}{l} \texttt{if } v = w \texttt{ then } com_1 \\ \texttt{else } com_2 \end{array} \right]\!\!\right] = (\check{v} = \check{w} \wedge \mathcal{PLC}^J[\![com_1]\!]) \sqcup \mathcal{PLC}^J[\![com_2]\!]$$
$$\mathcal{PLC}^J\left[\!\!\left[\begin{array}{l} \texttt{if } v = \texttt{null then } com_1 \\ \texttt{else } com_2 \end{array} \right]\!\!\right] = (\check{v} = 0 \wedge \mathcal{PLC}^J[\![com_1]\!]) \sqcup (0 < \check{v} \wedge \mathcal{PLC}^J[\![com_2]\!])$$
$$\mathcal{PLC}^J[\![\{\}]\!] = U(V)$$
$$\mathcal{PLC}^J[\![\{com_1; \ldots; com_p\}]\!] = \mathcal{PLC}^J[\![com_1]\!] \circ \cdots \circ \mathcal{PLC}^J[\![com_p]\!].$$

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 $J'(\kappa.\texttt{m})$ is

$$\left[U(input(\kappa.\texttt{m})) \wedge \{\hat{w}_{n+i} = 0 \mid 1 \leq i \leq m\} \wedge \hat{out} = 0\right] \circ \mathcal{PLC}^J[\![body(\kappa.\texttt{m})]\!] \circ [\check{out} = \hat{out}].$$

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 $F^C \in \gamma(F^A)$.*

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

[1] K. R. Apt and D. Pedreschi. Reasoning about Termination of Pure Prolog Programs. *Information and Computation*, 106(1):109–157, September 1993.

[2] 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.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination Analysis of Integer Linear Loops. In M. Abadi and L. de Alfaro, editors, *Proc. of the 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502, San Francisco, CA, USA, August 2005. Springer-Verlag.

[4] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101, London, United Kingdom, September 2005. Springer-Verlag.

[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 Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[6] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[7] D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[8] S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *Theory and Practice of Logic Programming (TPLP)*, 5(1-2):75–91, January/March 2005.

[9] M. Hermenegildo, W. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(2 & 3):349–366, 1992.

[10] M. Kühnrich and N. D. Jones. Size Change Analysis of a Small C-like Language. Technical report, Copenhagen University, DIKU, 2004. Available at the address `http://www.cs.aau.dk/~mokyhn/publications/kuhnsct03.pdf`.

[11] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proc. of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, volume 36(3) of *ACM SIGPLAN Notices*, pages 81–92, London, UK, March 2001. ACM.

[12] N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In L. Naish, editor, *Proc. of the 14th International Conference on Logic Programming (ICLP)*, pages 63–77, Leuven, Belgium, July 1997. MIT Press.

[13] F. Mesnard and U. Neumerkel. Applying Static Analysis Techniques for Inferring Termination Conditions of Logic Programs. In P. Cousot, editor, *Proc. of the 8th Static Analysis Symposium (SAS)*, volume 2126 of *Lecture Notes in Computer Science*, pages 93–110, Paris, France, July 2001. Springer-Verlag.

[14] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251, Venice, Italy, 2004. Springer-Verlag.

[15] A. Podelski and A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. In M. Abadi and J. Palsberg, editors, *Proc. of the 32nd ACM Symposium on Principles of Programming Languages (POPL'05)*, pages 132–144, Long Beach, CA, USA, January 2005. ACM.

[16] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *Proc. of the 6th international conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06)*, Lecture Notes in Computer Science, Charleston, South Carolina, USA, January 2006. Springer-Verlag. To appear.

[17] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005. Springer-Verlag.

[18] A. Serebrenik and D. De Schreye. Inference of Termination Conditions for Numerical Loops. *Theory and Pratice of Logic Programming*, 4(5–6):719–751, September/November 2004.

[19] F. Spoto, M. P. Hill, and E. Payet. Path-Length Analysis for Object-Oriented Programs. Extended version with proofs. Available at `www.sci.univr.it/~spoto/papers.html`, 2006.

[20] R. Wilhelm, T. W. Reps, and S. Sagiv. Shape Analysis and Applications. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pages 175–218. CRC Press, 2002.

[21] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.