UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Precise Type Analysis by Abstract Interpretation[1]

Jens Knoop[2]
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau

Wolfgang Golubski
Fachbereich Elektrotechnik und Informatik
Universität-GH Siegen
Hölderlinstraße 3
D–57068 Siegen

MIP-9619
November 1996

[1] A preliminary version of this article appeared in joint Proceedings of the 13^{th} Workshop of the GI-FG 2.1.4 "Alternative Konzepte für Sprachen und Rechner" and the 15^{th} Workshop of the GI-ITG-FG 3.1.2 "Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware" (cf. [KG2]).

[2] Author's current address: Fachbereich 10 — Informatik, Carl von Ossietzky Universität Oldenburg, Uhlhornsweg 49 - 55, D-26111 Oldenburg, Germany. E-mail: jens.knoop@informatik.uni-oldenburg.de. Most of the work was done while the author was at the University of Passau. The author's work is partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

Abstract

Type analysis is the key for improving the efficiency of untyped object-oriented programs with dynamic method binding by replacing costly "late" by the more efficient "early" binding. The precision of type analysis and hence the runtime gain achievable depends above all on the treatment of *method calls*, *program branches*, and *assignments*. A *monomorphic*, *monovariant*, and *non-deterministic* treatment of method calls and program branches, and a *non-destructive* treatment of assignments are major sources introducing imprecision. In this article we present a new type analysis based on abstract interpretation, which systematically addresses and overcomes these problems. The new type analysis is unique for treating method calls *polymorphically* and *polyvariantly*, program branches and method sends almost *deterministically*, and assignments *destructively*. Above all, it is the almost deterministic treatment of program branches and method sends which makes our approach exceptional, and lets it improve on all previous related approaches to type analysis.

Keywords

(Untyped) object-oriented languages, type analysis, abstract interpretation, data flow analysis, *MOP*-Solution, *MFP*-Solution, (Object-Oriented) Coincidence Theorem, Smalltalk, (classical) program optimization.

Contents

1 Motivation **1**
Related Work . . . 2
Structure of the Paper . . . 3

2 Preliminaries **3**

3 OO-Analysis Models and OO-Analysis Graphs **6**
3.1 OO-Analysis Models . . . 6
3.2 OO-Analysis Graphs . . . 7

4 Abstract Interpretation **12**
4.1 Local Abstract Semantics . . . 12
4.2 First Globalization: The Meet Over All Paths Semantics . . . 12
4.3 Second Globalization: The Maximal Fixed Point Semantics . . . 13
4.4 Safety and Coincidence . . . 14

5 The Type Analysis **15**
5.1 Specification . . . 15

5.2 Main Result: Preciseness 18
5.3 An Illustrating Example 19

6 Conclusions **19**

References **21**

A The Generic Algorithm **24**

1 Motivation

Dynamic ("late") method binding is a popular feature of object-oriented programming languages in order to enhance flexibility. On the other hand, it usually leads to less efficient language implementations than *static ("early") method binding*. Replacing late by early binding where possible is thus a major mean for improving the efficiency of object-oriented programs with dynamic method binding (cf. [Go]). In general, this requires that the class of a message receiver is known at compile time, which necessitates precise type information. For untyped object-oriented languages like Smalltalk this information must explicitly be computed by a static program analysis, i.e., a *type analysis*.

The precision of type analysis depends above all on the treatment of *method calls*, *program branches*, and *assignments*. Major sources introducing imprecision into type analysis evolve from treating method calls *monomorphically* and *monovariantly*, program branches and method sends *non-deterministically*, and assignments *non-destructively*. The latter means treating assignments in a "cumulative" or "collecting" fashion as it is done by many type analyses instead of overwriting the value stored in a variable whenever it is assigned afresh as it is conform to the standard semantics of assignments. In this article we systematically address these problems, and present a new type analysis based on abstract interpretation which is unique to treat

- method calls *polymorphically* and *polyvariantly*,
- program branches and method sends almost *deterministically*, and
- assignments *destructively*.

Besides to systematically combine and enjoy these features (in comparison to related approaches addressing specific features only), it is particularly the almost deterministic treatment of program branches and method sends which makes our approach exceptional. Technically, this is achieched by means of *filter* functions. Intuitively, a filter blocks the propagation of type information, if it does not qualify for the program continuation guarded by it. As a side-effect this enables our analysis to detect *dead code* in an argument program, i.e., code which will *never* be executed, an information from which it directly profits as dead program parts cannot invalidate type information. The power introduced by filters is demonstrated when applying our analysis to the Smalltalk-like language of [OPS]: Here, it reveals that programs of this language are *statically deterministic*, i.e., a program encodes just a single program execution.

Fundamental to achieve this result is to adapt the framework of *abstract interpretation* (cf. [CC, Ma]), which has proved to be a powerful and theoretically well-founded common basis for static program analyses, to the object-oriented setting. An important feature of this extension, which in spirit is quite similar to the interprocedural extensions of [SP] and [KS], is to separate (1) the *specification* of an analysis problem and its intuitively desired solution from (2) its *computation*:

(1) the specification of an analysis problem directly induces a *reference solution* reflecting the program's operational semantics. It is conceptually easy to understand, and defines the intuitively desired solution of the problem under consideration. (2), the framework provides a *generic* computation procedure, which can directly be fed with the specification under consideration. A *safety* and a *coincidence theorem* then give sufficient conditions for the *correctness* and even the *coincidence* of the fixed point solution computed and the reference solution desired. Thus, in contrast to previous approaches *preciseness* of type analysis has a *formal meaning* in our approach. Moreover, the designer of an analysis need not know any details of the generic algorithm or of the theorems when applying the framework. In fact, knowing the specification is sufficient at this level, which eases application.

The power of our type analysis is demonstrated in the example of Figure 9, where it is unique to obtain precise type information. This example will be discussed in more detail in Section 5.3.

Related Work

Type analysis for dynamically-typed object-oriented languages is an active field of research within the last fifteen years, and there is a broad variety of approaches that have been proposed for it. In spirit most closely related to our approach, though not really comparable, are the type inference based approaches as e.g. in [Ag1, Ag2, BI, GJ, Jo, OPS, PS1, PS2, Su], and the data flow analysis (DFA) based approaches as e.g. in [PC1, PC2, VHU]. They characterize the state-of-the-art of the type inference and the data flow analysis based approaches, respectively.

The type inference approaches of [Ag2, OPS, PS2] improve on previous approaches of this group particularly in taking the context of message sends into account in order to improve the accuracy of the type information derived. The algorithm of [Ag2] considers even the cartesian product of the receiver type and all argument types of a method send. However, as it is typical for type inference, these algorithms do not perform a flow-sensitive analysis. This is addressed by the data flow based approaches. Here, the algorithms of [PC1, PC2] are exceptional for heuristically focussing the type analysis on program parts with "high" impact on the analysis results. The price for the improved precision is an exponential worst case time complexity. Our approach, which also has an exponential worst case time complexity, is not heuristic. A different setting has recently been considered by Pande and Ryder proposing a type analysis for C++ [PR].

Conceptually completely different and incomparable to our approach are the approaches of [AH, Ch, HCU, HU], which propose an interleaving of program execution with type analysis, e.g. during the pauses of the programmer (user), in order to exploit the feedback yielded by monitoring ("profiling") previous program executions.

Moreover, the approaches cited above, including also the conceptually different approach of hierarchy analysis of [DGC] and the definition-use chain based approach of [La], focus on algorithmic aspects, and do not systematically address correctness and preciseness issues, which is a central concern of our approach.

Structure of the Paper

After the preliminary Section 2, in which we define our Smalltalk-like programming language *SM*, we introduce in Section 3 OO-analysis models and OO-analysis graphs as appropriate representations of *SM*-programs for the formal development of our framework. In Section 4 we generalize the framework of abstract interpretation to the object-oriented setting. Afterwards we present our type analysis in Section 5 and demonstrate its power by means of an illustrating example in Section 5.3. Section 6, subsequently, contains our conclusions, and the Appendix presents the detailed generic algorithm.

2 Preliminaries

We consider a Smalltalk-like programming language *SM*, which is complex enough to demonstrate the essential features of our approach.

Syntax: *SM* is a variant of the language investigated in [OPS], enriched in order to avoid that the analysis problem becomes trivial for our analysis as for the original language of [OPS]. As in [OPS], a *program* is a set of classes followed by an *expression* called *main program*. *Classes* can be defined by means of inheritance. As in [OPS], metaclasses and blocks are omitted, which allows us to concentrate on the central features of our approach. However, in addition to the language of [OPS], *SM* contains a predefined class *integer* and two primitive methods for addition " $+$ " and multiplication " $*$ " of integer objects. Additionally, there is an equality test " $=$ ", which is assumed to never yield a runtime error, and which checks whether its arguments denote the same object. In this case, the test returns the value `nil`; otherwise, it returns the special value `nNil`. Moreover, there is an `if-then-else`-expression and an `instanceOf:`-expression in *SM*. The `if-then-else`-expression tests if the condition is `nil`, the `instanceOf:`-expression checks if the receiver is a member of the corresponding class. In this case, the expression evaluates to this class; otherwise, it evaluates to `nil`. Finally, there are two loop expressions with the standard semantics of "while" and "repeat" statements. Figure 1 summarizes the syntax of *SM* in terms of a context-free grammar.

Inheritance and name clashes: As in [OPS], we assume that inheritance is removed from an argument program. This can be achieved by a simple pre-compilation step (cf. [PS2]), which in essence expands all class definitions occurring in the argument program by adding renamed versions of all instance variables and methods inherited except for those being overwritten. Thus, in the resulting program every class definition contains all information including the inherited one explicitly and can be regarded as a separate unit. In particular, each `super`–message is replaced by a `self`–message sent to a renamed version of the inherited method. In general

(*Program*)	P	::=	$C_1 \ldots C_k\ E$
(*Class*)	C	::=	`class` $ClassId$ [`inherits` $ClassId$]
			`var` $Id_1 \ldots Id_k\ M_1 \ldots M_n$
			`end` $ClassId$
(*Method*)	M	::=	`method` $m_1 : Id_1 \ldots m_n : Id_n\ E$
(*Simple Expression*)	SE	::=	$ClassId$ `new` | `self` | `super` | Id |
			`nil` | *integer*
(*Simple Method Call*)	SMC	::=	$SE\ m_1 : SE_1 \ldots m_n : SE_n$
(*Primitive Method Call*)	PMC	::=	$Id_1\ +\ Id_2 \mid Id_1\ *\ Id_2$
(*Simple Expression or Method Call*)	$SEMC$	::=	$SE \mid SMC \mid PMC$
(*Simple Condition*)	SC	::=	SE | SE `instanceOf` : $ClassId$ | $Id_1\ =\ Id_2$
(*Expression*)	E	::=	$SEMC \mid Id := SEMC \mid E_1; E_2$ |
			`if` SC `then` E_1 `else` E_2 |
			`while` SC `do` E | `repeat` E `until` SC

Figure 1: The Syntax of the Programming Language *SM*

this preprocess leads to better results of our type analysis, but it is worth noting that it is not a prerequisite of our approach. Moreover, without loss of generality we assume that all instance variables of a program are denoted by different identifiers.

OO-program models: Following [KG1] we represent *SM*-programs by means of transition system like graphs throughout this article, called *OO-program models*.[1] An OO-program model is a system of *method models* representing the methods of the classes of the argument program. A method model is a graph $M = (\mathbf{N}, \mathbf{E}, \mathbf{s}, \mathbf{e})$ with node set $\mathbf{N}$, edge set $\mathbf{E}$, and a unique *start node* $\mathbf{s}$ and *end node* $\mathbf{e}$, which are assumed to have no predecessors and successors, respectively. The edges of M represent both the statements (i.e., simple expressions, simple or primitive method calls, simple conditions, or assignments whose right hand side expressions are simple expressions, or simple or primitive method calls), and the control flow of the underlying method, while the nodes just represent program points. This gives a method model the flavour of a transition system, and thus, we will use the notions "nodes" and "states", and "edges" and "transitions" of a method model synonymously. Without loss of generality we assume that incoming transitions of end nodes of method models are not labelled.[2]

Figure 2 shows the OO-program model, which will be used as the running example throughout this article. The program of this example, containing the class declarations *Main*, *A*, *B*, and *C*, which except for class *Main* implement a local method m, is complex enough to illustrate the central features and the power of our approach.

[1] OO-program models are an OO-variant of the program models originally introduced in [St].

[2] If necessary, appropriate transitions can always be added.

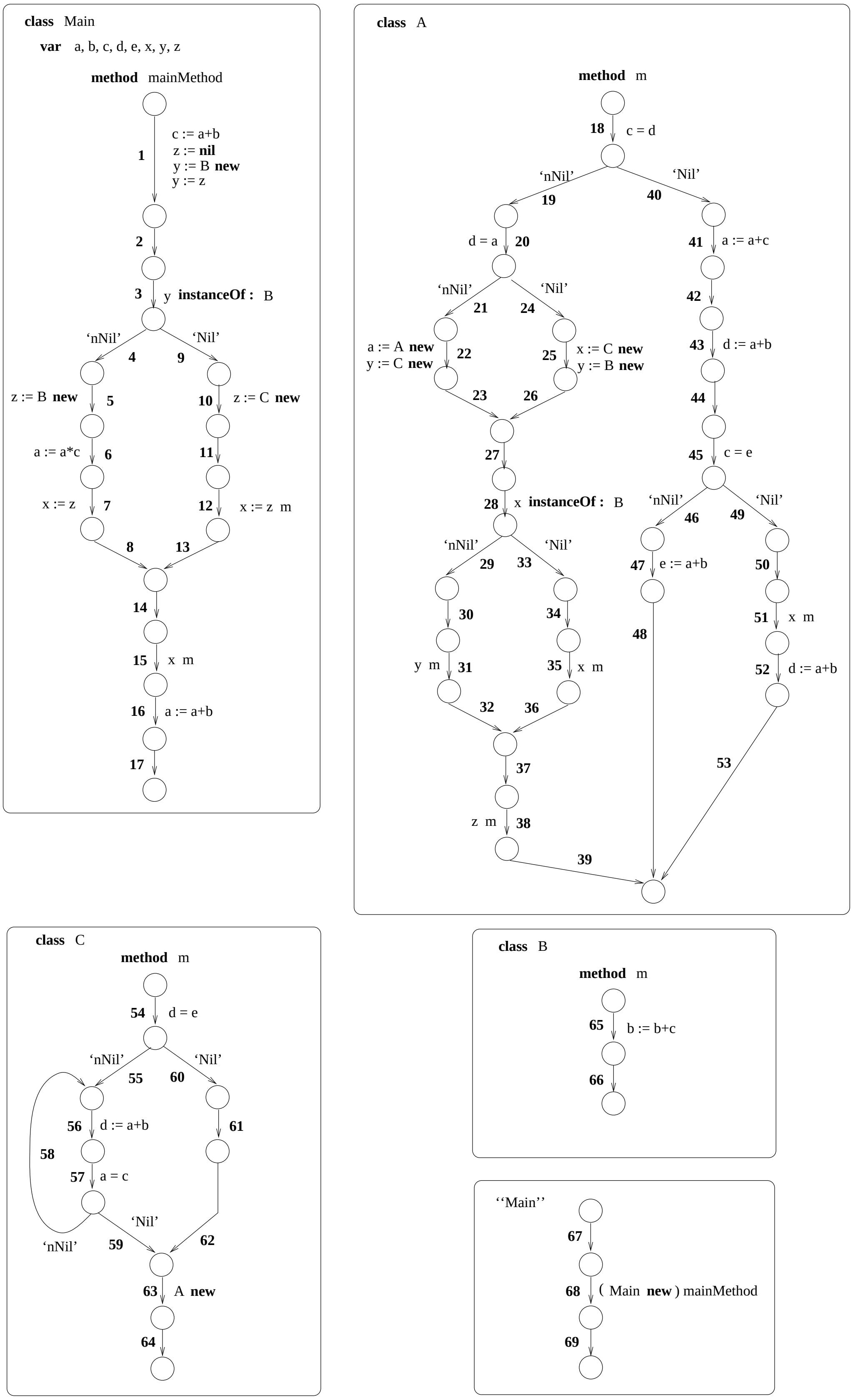

Figure 2: The OO-Program Model of the Running Example

3 OO-Analysis Models and OO-Analysis Graphs

OO-program models are quite a natural representation of *SM*-programs. However, in order to exploit the full power of our approach, our analyses work on a slightly modified variant of OO-program models, which we call *OO-analysis models*. In contrast to an OO-program model, an OO-analysis model makes the functional character of a method of an *SM*-program explicit, and allows us by means of filter transitions to properly deal with branch statements.

3.1 OO-Analysis Models

Given an OO-program model T, its corresponding *OO-analysis model* T' results from the following three-step procedure: First, splitting every transition involving a method call into a sequence of two transitions, where the first one stores the receiver of the method call in a fresh variable associated with the call site, and the second one performs the method call with the original receiver replaced by the fresh variable. Second, introducing for every method of T a fresh variable for explicitly storing its result. Third, introducing *filter* transitions in order to enable our analysis to properly deal with branching conditions. In the following we illustrate these steps in more detail.

Step 1 — Splitting Method Calls: Transitions $e \equiv (s,t)$, which are labelled by an expression or an assignment of the form

$$[Id\ :=\ ]\ SE\ m_1 : SE_1 \ldots m_n : SE_n$$

are split by a new state r. The resulting transitions (s,r) and (r,t) are then labelled by $e\text{-}sendsTo\ := SE$ and $[Id\ :=\ ]\ e\text{-}sendsTo\ m_1 : SE_1 \ldots m_n : SE_n$, respectively, as illustrated in Figure 3. This step simplifies our type analysis as the expression fixing the receiver of a method call is always given by a variable identifier.

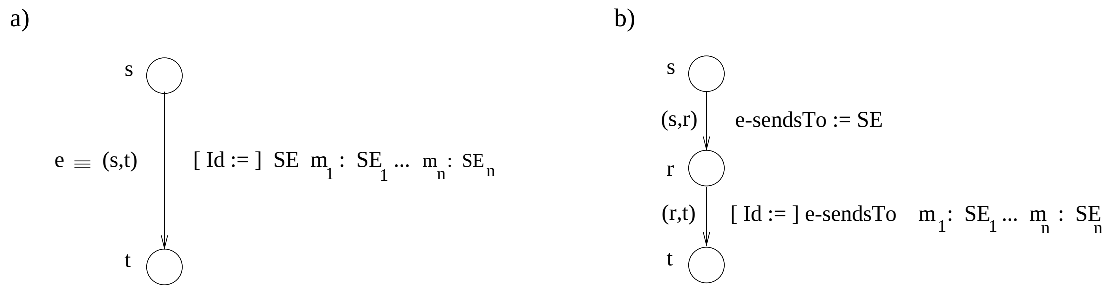

Figure 3: Splitting Method Calls

Step 2 — Making the Functional Character of Methods Explicit: For each class C of T, and each method $m \equiv m_1 : \ldots m_n :$ of C, a fresh variable $Res\text{-}C.m_1 : \ldots m_n :$ is introduced. Intuitively, $Res\text{-}C.m_1 : \ldots m_n :$ will be used for explicitly storing the result of the last expression evaluated on calling m in C. To this end consider the set of all incoming transitions of the predecessors of the end state of m. These transitions are either labelled by a simple expression, a simple or a primitive method call, or by an assignment whose right hand side expression is a simple expression, or a simple or a primitive method call. In the first three cases replace the label of the transition under consideration by $Res\text{-}C.m_1: \ldots m_n: := SEMC$. In the latter three cases annotate the transition leaving the destination state of the edge under consideration by the assignment $Res\text{-}C.m_1: \ldots m_n: := Id$ as illustrated in Figure 4.

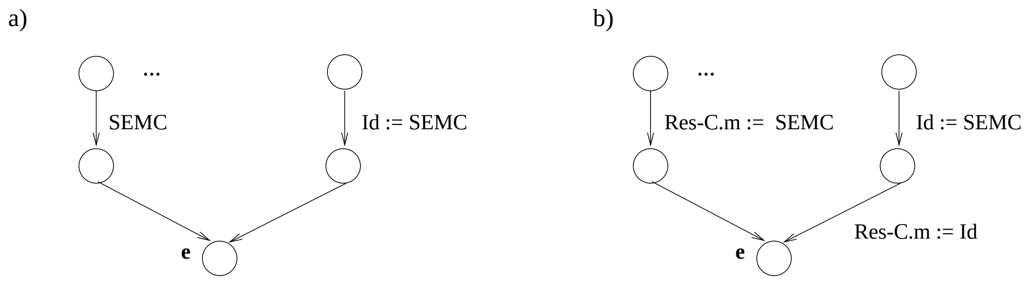

Figure 4: Making the Functional Character of Method Calls Explicit

Step 3 — Introducing Filters: The third step is applied to all transitions e labelled by a simple condition. These transitions represent the branching condition of the if-statements of the underlying *SM*-program. Similar to step 1, we first introduce a fresh variable *e-cond* for e, which allows us a uniform treatment of branching conditions independently of their particular syntactic shape during our type analysis. If the condition of e is of the form *SE* or $Id_1 = Id_2$, we replace the label of e by *e-cond* := *SE* and *e-cond* := $Id_1 = Id_2$, respectively. Otherwise, i.e., if the label is of the form *SE* `instanceOf` : $ClassId$, we split e by a new state and label the resulting two transitions by *e-cond* := *SE*, and *e-cond* := *e-cond* `instanceOf` : $ClassId$. Finally, the two outgoing transitions e_1 and e_2 of the destination state of e, which represent the "Nil" ("true") and the "nNil" ("false") successors of e, are labelled by the expressions *e-cond* `eqNil` and *e-cond* `neqNil`, respectively. The complete transformation of this step is illustrated in Figure 5.

Figure 6 shows the OO-analysis model of the OO-program model of Figure 2.

3.2 OO-Analysis Graphs

Whereas OO-analysis models (even in the presence of recursion) are perfectly well-suited for the denotational characterization of our analyses, their operational characterization specifying the intuitively desired reference solutions, rely on the explicit representation of the control flow caused by method calls. In anology to interprocedural DFA (cf. [SP, KS]), this is achieved by combining the method models of T' to

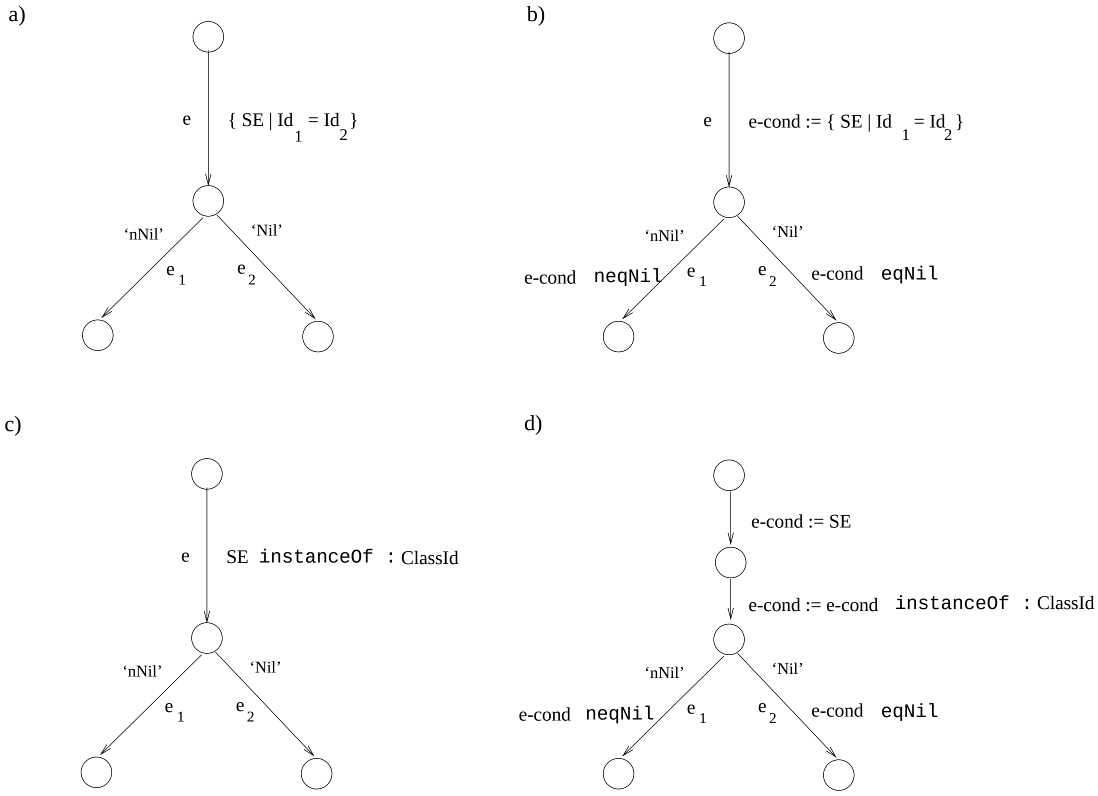

Figure 5: Introducing Filters

an *OO-analysis graph* $T^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$. In essence, T^* results from its underlying OO-analysis model T' by replacing every message send transition invoking a method m by a set of transitions connecting its start node with the start nodes of all method models named m, and their end nodes with the end node of the message send transition under consideration.

This transformation is illustrated in Figure 7 and Figure 8. For clarity, the "inter-object" control flow transitions are represented by dashed lines. Moreover, *start* and *end* denote two functions mapping a method model to its start node and its end node, respectively.

In the example of Figure 7 the transition e is assumed to represent the simple method call

$$\textit{e-sendsTo}\ m_1 : SE_1 \ldots m_n : SE_n$$

with $\mathit{Impl}(\mathit{Method}(e)) = \{C_1, \ldots, C_k\}$, i.e., $C_1, \ldots, C_k$ are the classes containing a method declaration with identifier $m \equiv m_1 : \ldots m_n :$. For each class $C \in \mathit{Impl}(\mathit{Method}(e))$ a new transition e^C_{fil} starting in $r \equiv \mathit{source}(e)$ is introduced. It is labelled by the expression *e-sendsTo* $C.m$. Intuitively, this transition will serve as a filter for checking whether the receiver of the method called is an instance of class C. The destination state of the new transition is connected with the start state of the method model of m in C. This transition is labelled with an assignment representing the parameter transfer, i.e., $f^C_1, \ldots, f^C_n$ are here assumed to denote the formal parameters of m in C. Additionally, the end state

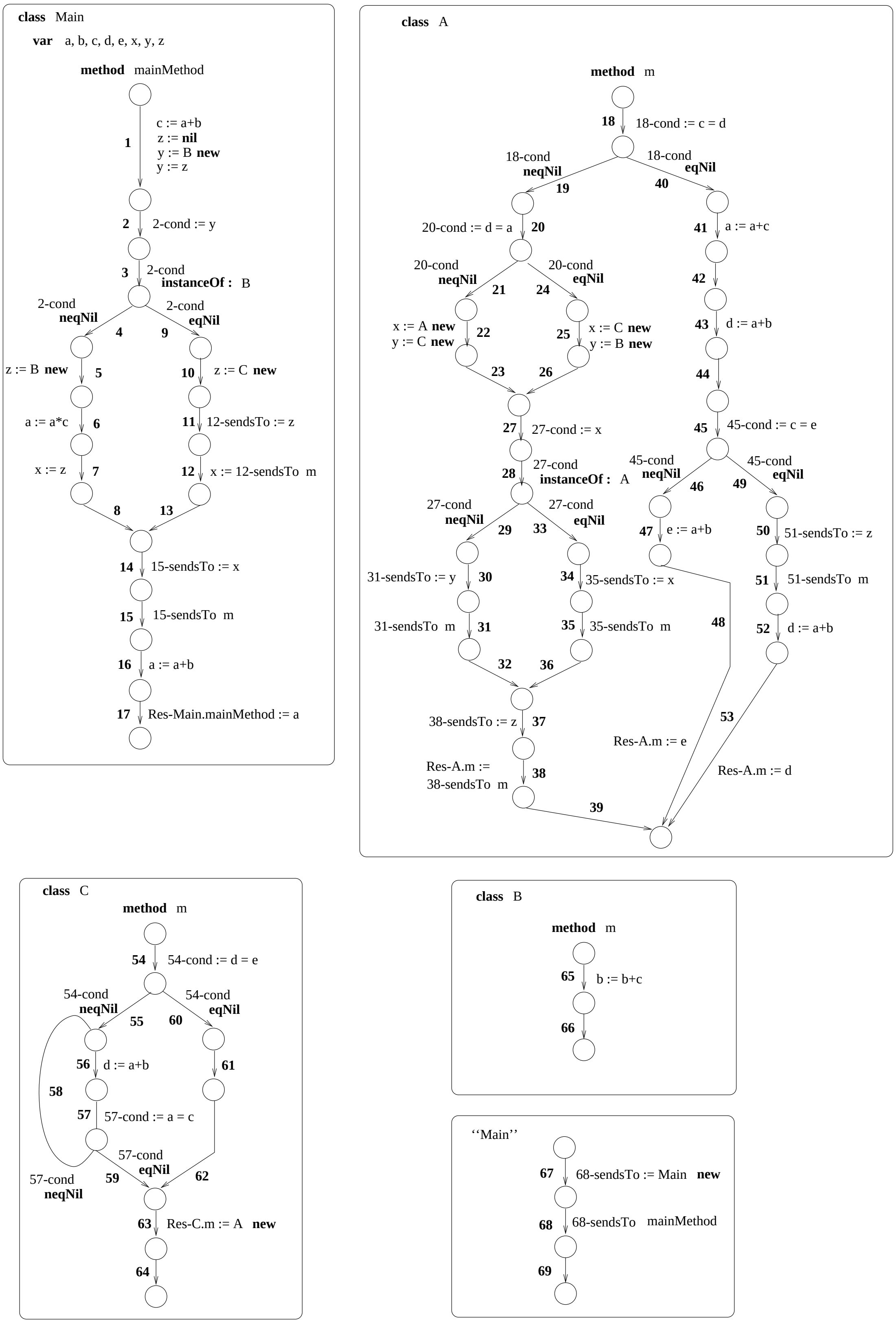

Figure 6: The OO-Analysis Model of the Running Example

of this method model is connected with the destination state $t \equiv dest(e)$. Moreover, another transition is inserted starting in $r \equiv source(e)$, which is labelled by the expression *e-sendsTo* `SthOutOf` $\{C_1, \ldots, C_k\}$.[3] Intuitively, this filter gives us a handle for treating method calls being not understood. The destination state of this transition is directly connected with the destination state $t \equiv dest(e)$. Finally, the original transition e is removed. The result of the complete transformation is illustrated in Figure 7.

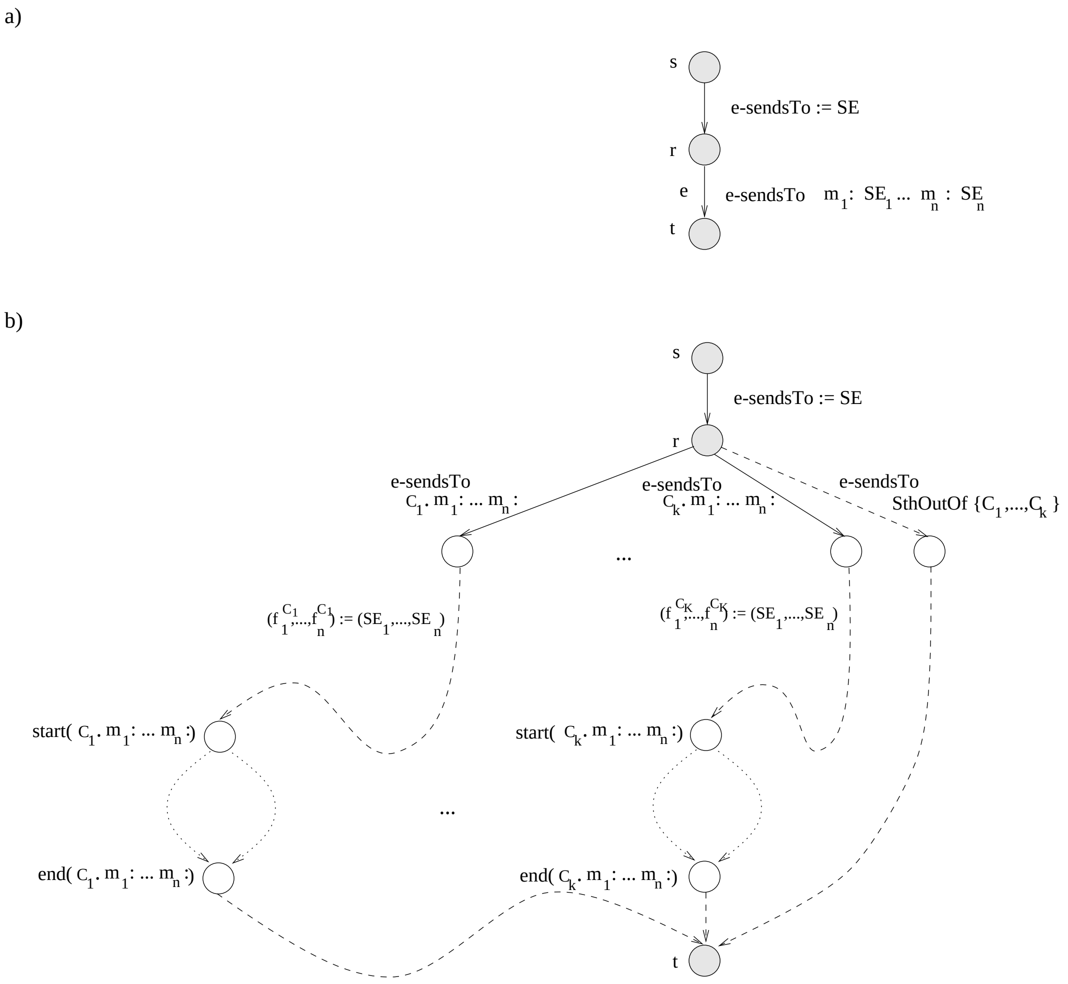

Figure 7: Making the Inter-Object Control Flow Explicit: "Pure" Method Calls

Figure 8 illustrates the remaining case, where e represents an assignment with a simple method call as its right hand side expression. The transformation performed in this case is the same as above except that additionally the transitions linking the end states of called methods with the destination state $t \equiv dest(e)$ are labelled by the assignment $Id := Res\text{-}C.m_1 : \ldots m_n :$ in order to model the assignment to the left hand side variable Id of the underlying assignment of T.

[3] `SthOutOf` is an abbreviation of "Something out of".

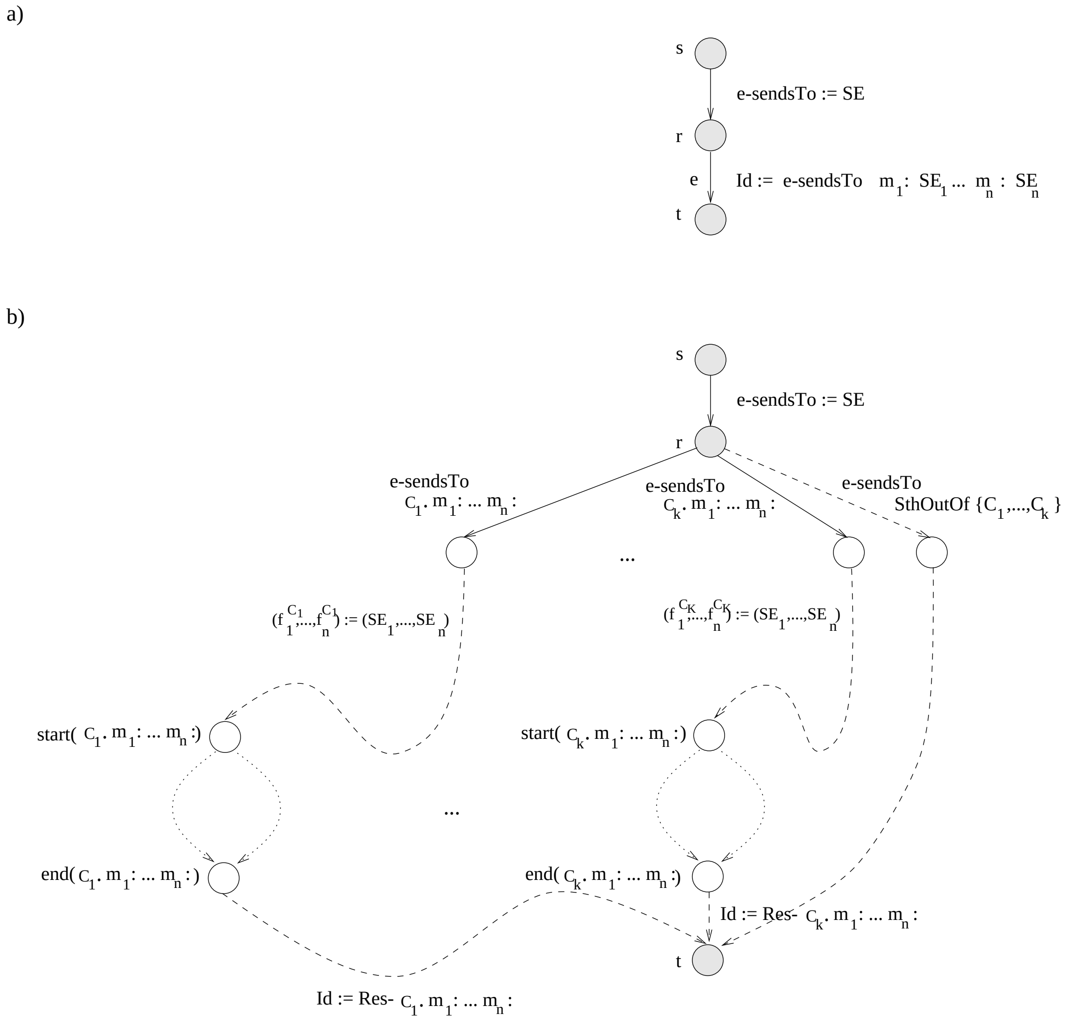

Figure 8: Making the Inter-Object Control Flow Explicit: “Assignment” Calls

Valid Paths: OO-Paths

Like for an interprocedural flow graph (cf. [SP, KS]), it is worth noting that there are paths in T^*, which do not correspond to a runtime execution of the underlying program, and thus give rise to the notion of *valid paths*. Validity imposes a consistency constraint on (finite) paths, which guarantees that they respect the call/return behaviour of method calls. Intuitively, identifying matching call and return transitions with “(” and “)”, respectively, the set of valid paths corresponds to the prefix-closed language of balanced parentheses. A formal definition, wich evolves straightforward from the definition of interprocedurally valid paths of Sharir and Pnueli [SP], and where $\mathbf{P}_{T^*}[m,n]$ denotes the set of all (finite) paths from m to n in T^*, is given below.

Definition 3.1 (Valid Paths (OO-Paths))
Let $p \in \mathbf{P}_{T^*}[m,n]$*. Then* p *is a* valid path (*or for short: an OO-path*) *iff the tuple* $(e_1, \ldots, e_r)$*, which results from* p *by deleting all transitions in* $E^* \backslash (E^*_{call} \cup E^*_{ret})$*, is*

well-formed in the following sense:

- *if there is no return transition in* $(e_1, \ldots, e_r)$, *then* $(e_1, \ldots, e_r)$ *is well-formed,*
- *otherwise, let* j *be the smallest index in* $\{1, \ldots, r\}$ *such that* e_j *is a return transition. Then* $(e_1, \ldots, e_r)$ *is well-formed iff* $j > 1$ *and* e_{j-1} *is a call transition of* E^*_{call} *matching* e_j, *and the remaining sequence after deleting* e_{j-1} *and* e_j *is well-formed too.*

In the following we denote the set of all valid paths (for short: *OO-paths*) from m to n by $\mathbf{OP}[m, n]$.

4 Abstract Interpretation

Abstract interpretation has proved to be a powerful and theoretically well-founded framework for static program analysis (cf. [CC, Ma]). The point of this approach is to replace the "full" semantics of a program by a simpler, more abstract version, which is tailored to deal with a specific problem. In this section we adapt this approach to the OO-setting of this article following the lines of [KG1].

4.1 Local Abstract Semantics

As usual in abstract interpretation, the (global) abstract semantics is induced by a local abstract semantics $[\![\ ]\!]^* : E^* \to (\mathcal{C} \to \mathcal{C})$, which gives abstract meaning to every transition of an OO-analysis graph in terms of a transformation on a complete lattice $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top)$ with least element $\bot$ and greatest element $\top$. Its elements are assumed to represent the data flow information of interest.

The global abstract semantics then results from one of the following two globalization approaches: the "operational" *meet over all paths* (*MOP*) approach leading to the *MOP*-solution, and the "denotational" *maximal fixed point* (*MFP*) approach leading to the *MFP*-solution in the sense of Kam and Ullman [KU], respectively.

4.2 First Globalization: The Meet Over All Paths Semantics

The operational *MOP*-approach globalizes a local abstract semantics by directly mimicing possible program executions: It "meets" (intersects) all informations, which belong to a program path reaching the program point under consideration. The object-oriented variant of this approach is as follows, where $[\![\ ]\!]^*$ is the straightforward extension of a local abstract semantics to OO-paths, i.e., for each path $p \in \mathbf{OP}[m, n]$ it is defined by:[4]

$$[\![\, p \,]\!]^* =_{df} \begin{cases} Id_{\mathcal{C}} & \text{if } p = \varepsilon \\ [\![\, (e_2, \ldots, e_q) \,]\!]^* \circ [\![\, e_1 \,]\!]^* & \text{otherwise} \end{cases}$$

[4] "$=_{df}$" means "*equal by definition*", and $Id_{\mathcal{C}}$ denotes the identity on $\mathcal{C}$.

The *OMOP*-Solution:

$$\forall\, c_0 \in \mathcal{C}\ \forall\, n \in N^*.\ \mathit{OMOP}_{([\![\ ]\!]^*, c_0)}(n) =_{df} \bigsqcap \{\, [\![\, p\,]\!]^*(c_0) \mid p \in \mathbf{OP}[\mathbf{s}^*, n] \,\}$$

The *OMOP*-solution directly reflects our desires. Unfortunately, however, in general it does not specify an effective computation procedure.[5]

4.3 Second Globalization: The Maximal Fixed Point Semantics

As in interprocedural DFA (cf. [KS, SP]) the key to the object-oriented extension of the denotational approach is a preprocess for computing the meanings of method send transitions in terms of the meaning of the methods called. In the OO-setting this requires the introduction of an auxiliary semantic functional $[\![\![\ ]\!]\!]$, which gives meaning to whole method models. Intuitively, $[\![\![\, n\,]\!]\!]$ transforms data flow information that is assumed to be valid at the entry of the method containing n into the corresponding data flow information being valid at n. Hence, the meaning function of the method m is given by $[\![\![\, \mathit{end}(m)\,]\!]\!]$, where $\mathit{end}(m)$ denotes the end node of the method model m. In comparison to the settings of [KS, SP], the preprocess is generalized here for computing also the meaning of assignments whose right hand side terms are method calls.

Definition 4.1 (Global Semantics of Methods)
$[\![\![\ ]\!]\!] : N \to (\mathcal{C} \to \mathcal{C})$ and $[\![\![\ ]\!]\!] : E \to (\mathcal{C} \to \mathcal{C})$ are defined as the greatest solution of the equation system given by:

$$[\![\![\, n\,]\!]\!] = \begin{cases} \mathit{Id}_{\mathcal{C}} & \text{if } n \in \mathit{start}(\mathcal{M}_{Mdl}) \\ \bigsqcap\{[\![\![\, (m,n)\,]\!]\!] \circ [\![\![\, m\,]\!]\!] \mid m \in \mathit{pred}_{mm(n)}(n)\} & \text{otherwise} \end{cases}$$

and

$$[\![\![\, e\,]\!]\!] = \begin{cases} [\![\, e_{SthOutOf}\,]\!]^* \sqcap & \\ \bigsqcap\{\, [\![\, e^C_{ass}\,]\!]^* \circ [\![\![\, \mathit{end}(\mathit{Model}(\mathit{Method}(e), C))\,]\!]\!] \circ [\![\, e^C_{pt}\,]\!]^* \circ [\![\, e^C_{fil}\,]\!]^* & \\ \qquad\qquad \mid C \in \mathit{Impl}(\mathit{Method}(e))\,\} & \text{if } e \in E^{ass}_{mc} \\ [\![\, e_{SthOutOf}\,]\!]^* \sqcap & \\ \bigsqcap\{\, [\![\![\, \mathit{end}(\mathit{Model}(\mathit{Method}(e), C))\,]\!]\!] \circ [\![\, e^C_{pt}\,]\!]^* \circ [\![\, e^C_{fil}\,]\!]^* & \\ \qquad\qquad \mid C \in \mathit{Impl}(\mathit{Method}(e))\,\} & \text{if } e \in E^{pure}_{mc} \\ [\![\, e\,]\!]^* & \text{otherwise} \end{cases}$$

where $\mathit{Id}_{\mathcal{C}}$ denotes the identity on $\mathcal{C}$, $\bigsqcap$ the "componentwise" meet operation on functions on $\mathcal{C}$, $\mathcal{M}_{Mdl}$ the set of all method models, start and end two functions yielding the start node and the end node of a method model, Method the method

[5] Think for example of loops or of recursive method calls in a program.

identifier invoked by its argument transition, Impl the set of classes implementing its argument method, Model the method specified by its arguments, mm the method model containing its argument node, and E_{mc}^{pure} (E_{mc}^{ass}) the set of all transitions representing a method call (an assignment with right hand side method call).

Intuitively, the semantics of a transition e involving a method call is the meet over the effects of all methods of the appropriate name of the different classes of T ($\sqcap\{\dots\}$), and of the effect of a probably not understood message ($[\![\, e_{SthOutOf} \,]\!]^*$). Moreover, note that in any reasonable analysis context (see e.g. Section 5) the steps, in which the effect of calling a method is computed, directly reflect the phases of executing a method call: *Filtering* the information ($[\![\, e_{fil}^C \,]\!]^*$), *transferring* the parameters ($[\![\, e_{pt}^C \,]\!]^*$), *evaluating* the method body ($[\![\, end(Model(\,Method(e), C)) \,]\!]$), and, in case of assignments with a right hand side method call, *assigning* the result of the method call to the left hand side variable ($[\![\, e_{ass}^C \,]\!]^*$).

The following equation system, where *source*, *Class*, and *Sender* denote functions yielding the start node of a transition, the class of a node, and the set of transitions invoking an argument method, respectively, then characterizes the *OMFP*-approach:

Equation System 4.2 (The *OMFP*-Equation System)

$$\mathbf{dfi}(n) \quad = \quad \begin{cases} c_0 & \text{if } n = start(mainMethod) \\ \sqcap\{\, [\![\, e_{pt}^{Class(n)} \,]\!]^* \circ [\![\, e_{fil}^{Class(n)} \,]\!]^* (\mathbf{dfi}(source(e))) \mid e \in Sender(mm(n)) \,\} & \\ & \text{if } n \in start(\mathcal{M}_{Mdl}) \backslash \{start(mainMethod)\} \\ \sqcap\{\, [\![\, (m,n) \,]\!](\mathbf{dfi}(m)) \mid m \in pred_{mm(n)}(n) \,\} & \text{otherwise} \end{cases}$$

Denoting the greatest solution of Equation System 4.2 with respect to a start information $c_0 \in \mathcal{C}$ by dfi_{c_0}, the *OMFP*-solution is defined by:

$$\textbf{The } \mathit{OMFP}\textbf{-Solution: } \forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ \mathit{OMFP}_{([\![\ ]\!]^*, c_0)}(n) =_{df} \mathsf{dfi}_{c_0}(n)$$

As in intra- and interprocedural DFA the *OMFP*-approach is practically relevant because it induces an iterative computation procedure for computing the *OMFP*-solution. The detailed generic algorithm is recalled in Appendix A. It is effective if the function lattice satisfies the descending chain condition, and if the local semantic functions are monotonic. The question of correctness (safety) or even optimality (coincidence) of the *OMFP*-solution with respect to the *OMOP*-solution is answered by the object-oriented generalizations of the (intraprocedural) Safety and Coincidence Theorems of Kam and Ullman [KU], which we are going to recall next.

4.4 Safety and Coincidence

Analogously to their conventional counterparts, the OO-Safety Theorem 4.3 and the OO-Coincidence Theorem 4.4 give sufficient conditions for the correctness and the coincidence of the *OMFP*-solution with respect to the *OMOP*-solution (cf. [KG1]). Along the lines of [KS, SP] we can prove:

Theorem 4.3 (OO-Safety Theorem)
The OMFP-solution is a correct approximation of the OMOP-solution, i.e.,

$$\forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ \mathit{OMFP}_{([\![\ ]\!]^*, c_0)}(n) \sqsubseteq \mathit{OMOP}_{([\![\ ]\!]^*, c_0)}(n)$$

if the local semantic functions $[\![\, e \,]\!]^*$, $e \in E^*$, *are all monotonic.*[6]

Theorem 4.4 (OO-Coincidence Theorem)
The OMFP-solution and the OMOP-solution coincide, i.e.,

$$\forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ \mathit{OMFP}_{([\![\ ]\!]^*, c_0)}(n) = \mathit{OMOP}_{([\![\ ]\!]^*, c_0)}(n)$$

if the local semantic functions $[\![\, e \,]\!]^*$, $e \in E^*$, *are all distributive.*[7]

5 The Type Analysis

In this section we develop our type analysis applying the framework of Section 4. In order to allow an unparameterized notation we consider an arbitrary, but fixed OO-analysis model T' and its corresponding OO-analysis graph T^*. Let $\mathcal{C}' =_{df} \mathcal{C}_{Id} \cup \{\mathit{Int}, \mathit{Nil}, \mathit{nNil}\}$ be the set of class identifiers of T' enriched by the new elements *Int*, *Nil*, and *nNil*, which represent the class of integers and the special values `nil` and `nNil`, respectively, and let $\mathcal{V}' =_{df} \mathcal{V}_{Id}(T^*)$ denote the set of all variable identifiers of T^*.

5.1 Specification

Following Section 4, specifying our type analysis requires to fix the data domain, the local abstract semantics and the start information. Subsequently, these components can directly be fed into the generic computation procedure of Section A for computing the type information in the sense of the *OMFP*-solution.

Data Domain: The data domain of our type analysis is given by the powerset lattice:

$$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{P}(\mathcal{F}), \cup, \cap, \supseteq, \mathcal{F}, \emptyset)$$

where $\mathcal{P}$ denotes the powerset operator, and $\mathcal{F} =_{df} [\mathcal{V}' \to \mathcal{C}'] \cup \{\top_f : \mathcal{V}' \to \{\Lambda\}\}$ the set of all functions from $\mathcal{V}'$ to $\mathcal{C}'$ enriched by the constant function $\top_f$, which maps every variable to an "artificial" element $\Lambda \notin \mathcal{C}'$. Intuitively, a function $f \in \mathcal{F}$ attached to a program point represents the type information which is valid for some program execution reaching it. The function $\top_f$ will be used for modelling that a message call has not been understood. Moreover, program points which

[6] A function $f : \mathcal{C} \to \mathcal{C}$, $\mathcal{C}$ a complete lattice, is called *monotonic* if and only if $\forall\, c, c' \in \mathcal{C}.\ c \sqsubseteq c' \Rightarrow f(c) \sqsubseteq f(c')$.

[7] A function $f : \mathcal{C} \to \mathcal{C}$, $\mathcal{C}$ a complete lattice, is called *distributive* if and only if $\forall C \subseteq \mathcal{C}.\ f(\bigsqcap C) = \bigsqcap \{f(c) \mid c \in C\}$.

are annotated with $\emptyset$ after the termination of the type analysis are not reachable at runtime, and thus are dead code. Conversely, the existence of program points which are annotated with $\mathcal{F}$ after the termination indicates that a message call has not been understood. In the following we use the convention that if $f \in \mathcal{F}\backslash\{\top_f\}$, and $c \in \mathcal{C}'$, then $f[c\backslash v]$ is the unique function $f[c\backslash v]$ of $\mathcal{F}$, which is defined by $f[c\backslash v](x)=c$, if $x=v$, and by $f[c\backslash v](x)=f(x)$, otherwise.

Local Abstract Semantics: The local abstract semantics of our type analysis is given by the functional $[\![\ ]\!]^* : E^* \to (\mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F}))$. It is defined by:

$$\forall\, e \in E^*\ \forall\, F \in \mathcal{P}(\mathcal{F}).\ [\![\, e\,]\!]^*(F) =_{df} \begin{cases} \mathcal{F} & \text{if } \top_f \in F \\ \bigcup\{\, [\![\, e\,]\!](f) \mid f \in F\,\} & \text{otherwise} \end{cases}$$

where the auxiliary functional $[\![\ ]\!] : E^* \to (\mathcal{F}\backslash\{\top_f\} \to \mathcal{P}(\mathcal{F}))$ is given by:[8]

- *Assignments:* if e represents an assignment of the form
  - $Id := SE$

$$[\![\, e\,]\!](f) =_{df} \begin{cases} f[A\backslash Id] & \text{if } SE \equiv A\ \texttt{new} \\ f[Class(e)\backslash Id] & \text{if } SE \equiv \texttt{self} \\ f[f(x)\backslash Id] & \text{if } SE \equiv x,\ x \in \mathcal{V}' \\ f[Nil\backslash Id] & \text{if } SE \equiv \texttt{nil} \\ f[Int\backslash Id] & \text{if } SE \equiv integer \end{cases}$$

  - $Id := Id_1\ \{+|*\}\ Id_2$

$$[\![\, e\,]\!](f) =_{df} \begin{cases} f[Int\backslash Id] & \text{if } f(Id_1) = f(Id_2) = Int \\ \mathcal{F} & \text{otherwise} \end{cases}$$

  - $e\text{-}cond := e\text{-}cond$ `instanceOf` $: C :$

$$[\![\, e\,]\!](f) =_{df} \begin{cases} f & \text{if } f(e\text{-}cond) = C \\ f[Nil\backslash e\text{-}cond] & \text{otherwise} \end{cases}$$

  - $e\text{-}cond := Id_1\ =\ Id_2\ :\quad [\![\, e\,]\!](f) =_{df} \{f[Nil\backslash e\text{-}cond], f[nNil\backslash e\text{-}cond]\}$
- *Expressions:* if e represents an expression of the form $SE\ :\quad [\![\, e\,]\!](f) =_{df} f$
- *Filters:* if e represents a filter of the form
  - $e\text{-}sendsTo\ C.m_1 : \ldots m_n :$

$$[\![\, e\,]\!](f) =_{df} \begin{cases} f & \text{if } f(e\text{-}sendsTo) = C \\ \emptyset & \text{otherwise} \end{cases}$$

[8]For simplicity we identify singleton sets with their unique element.

– e-$sendsTo$ `SthOutOf` $Impl(Method(e))$

$$[\![\, e \,]\!](f) =_{df} \begin{cases} \mathcal{F} & \text{if } f(e\text{-}sendsTo) \notin Impl(Method(e)) \\ \emptyset & \text{otherwise} \end{cases}$$

– e-$cond$ `eqNil`

$$[\![\, e \,]\!](f) =_{df} \begin{cases} f & \text{if } f(e\text{-}cond) = Nil \\ \emptyset & \text{otherwise} \end{cases}$$

– e-$cond$ `neqNil`

$$[\![\, e \,]\!](f) =_{df} \begin{cases} f & \text{if } f(e\text{-}cond) \neq Nil \\ \emptyset & \text{otherwise} \end{cases}$$

Start information: The start information is given by the singleton set $\{ f_0 \}$, where $f_0 \in \mathcal{F}$ is defined by $f_0(v) = Nil$ for all $v \in \mathcal{V}'$. Intuitively, this choice of the start information expresses that we do not assume anything about the context in which the argument program is called.

As mentioned before the complete specification can now directly be fed into the generic algorithm of our framework computing the *OMFP*-solution (cf. Appendix A). Central for the power of our analysis, which is the so-called *disjunctive completion* of the monotonic type analysis of [KG1], are the following features:

Polymorphic and polyvariant treatment of method calls: This is a consequence of the preprocess computing the full semantics of method calls with respect to the data domain and the local abstract semantics under consideration (cf. Definition 4.1). In fact, this is automatically taken care of. When applying the framework, one does not have to bother about it.

Almost deterministic treatment of program branches and method sends: This is a consequence of introducing filter transitions, which allow us a "deterministic-like" interpretation of branch and method send transitions. In fact, we do not know of any other approach achieving this. Intuitively, a filter "blocks" the propagation of type information, if it does not qualify for a program continuation guarded by the filter. For example, the filter e-$sendsTo$ $C.m_1 : \ldots m_n :$ propagates type information, i.e., a function $f \in \mathcal{F}$, only, if C is the receiver of the current message. Otherwise, the type information is blocked. Technically this is achieved by propagating the trivial information $\emptyset$ only. Similarly, this holds for the filter transitions e-$cond$ `eqNil` and e-$cond$ `neqNil` guarding conditional branches. Important and slightly different is the effect of filters of the form e-$sendsTo$ `SthOutOf` $Impl(Method(e))$, which are dealing with method calls being not understood. This situation arises, if the receiver of a method call is an object of a class that does not implement the required method. This occurs, if $f(e\text{-}sendsTo) \notin Impl(Method(e))$. In this case the filter propagates the bottom element $\mathcal{F}$ of our data domain in order to indicate the type error. Similarly, an assignment with a primitive method call on its right hand side propagates $\mathcal{F}$, if it is invoked with non-integer operands. The power introduced by filters is central for enabling our type analysis to reveal that programs of the language investigated in [OPS] are statically deterministic.

Destructive treatment of assignments: The local semantics defined for assignments guarantees that the effect of an assignment is "destructive", i.e., the previously stored value is replaced by the value of the right hand side expression. This is conform to the standard semantics of assignments and can in fact easily be achieved by abstract interpretation based approaches like ours. Thus, it is worth noting that such a destructive treatment of assignments is still in contrast to many approaches for type analysis treating them in a "cumulative" or "collecting" fashion. This, however, is not only unintuitive regarding their ordinary semantics, it also introduces dramatic imprecisions into the results of type analysis.

5.2 Main Result: Preciseness

Formally, preciseness of our type analysis means that for every variable v the type information

$$\mathit{TypeInfo}_{\mathit{OMFP}}(n)(v) =_{df} \{\, f(v) \mid f \in \mathit{OMFP}_{([\![\ ]\!]^*,\{f_0\})}(n) \,\}$$

which is induced by the *OMFP*-solution and computed by our algorithm, coincides with the desired reference solution induced by the *OMOP*-solution given by

$$\mathit{TypeInfo}_{\mathit{OMOP}}(n)(v) =_{df} \{\, f(v) \mid f \in \mathit{OMOP}_{([\![\ ]\!]^*,\{f_0\})}(n) \,\}.$$

The key for establishing this coincidence is the Distributivity Lemma 5.1 given below. It is an immediate consequence of the fact that the local semantic functions process the elements of an argument set of functions separately and independently.

Lemma 5.1 (Distributivity)
The local semantic functions $[\![\, e \,]\!]^*$, $e \in E^*$, are all distributive.

Combining Lemma 5.1 and Coincidence Theorem 4.4, and exploiting the finiteness of $\mathcal{F}$, we obtain as desired:

Theorem 5.2 (Preciseness)
The OMFP-solution of our type analysis is effectively computable and coincides with its OMOP-solution, i.e.,

$$\forall\, n \in N.\ \mathit{OMFP}_{([\![\ ]\!]^*,\{f_0\})}(n) = \mathit{OMOP}_{([\![\ ]\!]^*,\{f_0\})}(n).$$

5.3 An Illustrating Example

The OO-analysis model of our running example of Figure 6 containing the class declarations *Main*, A, B, and C and implementing except for *Main* all a local method m, illustrates the power of our type analysis by high-lightening some of its most important results displayed in Figure 9. First, our type analysis detects that the left branch following transition **3** is never taken because the states of the left branch are annotated by the initial information $\emptyset$ after the termination of the analysis. Moreover, it detects that at transition **12** method m of an object of class C is invoked, and that this call yields an object of class A. Combining this observation with the information that the left branch of method *mainMethod* is dead code, our analysis reveals that the transitions **15** and **51** both invoke method m of an object of class A.

Most strikingly the power of our analysis is demonstrated in the subgraph encapsulated by the transitions **27** and **38**. At the start node of transition **27** it detects that x and y represent either a *pair* of objects of classes A and C (function f_1 contributed by the left incoming branch) or of classes C and B (function f_2 contributed by the right incoming branch). The filters associated with the transitions **29** and **33** let only pass f_1 and f_2, respectively. As a consequence, our analysis detects that the receivers of both method sends at transitions **31** and **35** are objects of class C. In fact, we do not know of any other type analysis to detect this.

Moreover, by means of the results sketched above, our analysis also detects that method m of class B is never invoked. Thus, for the example of Figure 2 our type analysis does not only guarantee the absence of runtime errors due to messages being not understood, it even gives precise type information for every message send site. To the best of our knowledge, no previous related type analysis would succeed on this example.

6 Conclusions

Treating method calls monomorphically and monovariantly, program branches and method sends non-deterministically, and assignments non-destructively are well-known major reasons leading to imprecise results of type analysis. In this article we focussed on these key issues and presented a new type analysis based on abstract interpretation, which systematically overcomes the related problems. In fact, the new type analysis is unique for treating method calls polymorphically and polyvariantly, program branches and method sends almost deterministically, and assignments destructively. The key to this result was to extend the framework of abstract interpretation to the OO-setting. The impact of this approach, however, goes far beyond of our type analysis. As illustrated in [KG1], it links classical program analysis and optimization to the OO-setting: Along the lines of [KRS1] all the powerful optimizations developed for "conventional" procedural languages like *code motion* (cf. [KRS2]), *partial dead code elimination* (cf. [KRS3]), *assignment mo-*

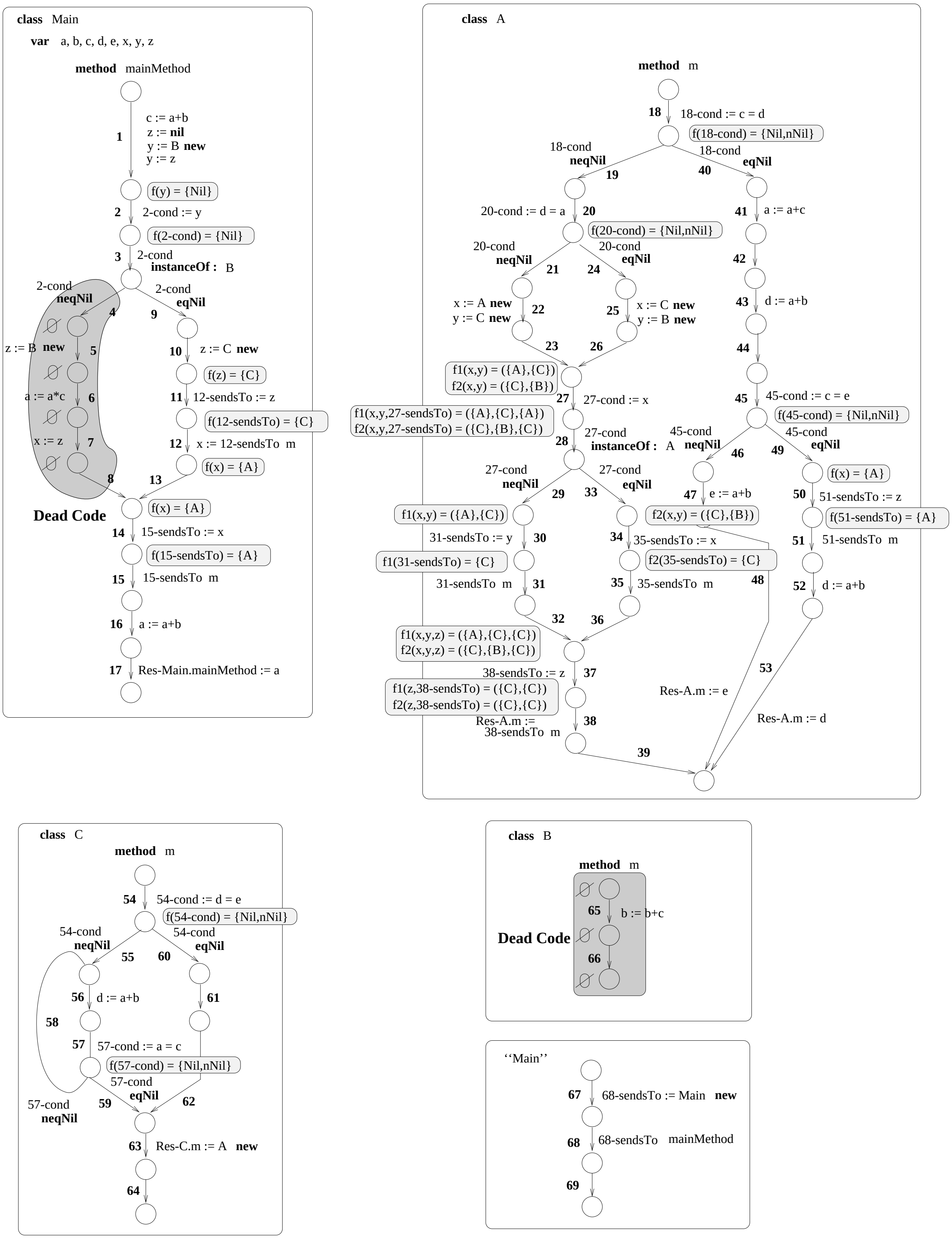

Figure 9: Illustrating the Power of the new Type Analysis

tion (cf. [KRS4]), or *strength reduction* (cf. [KRS5]) can now be made available for OO-programs. Moreover, due to the genericity of the algorithm of our framework, transferring these techniques does not require to construct and implement for each required analysis a new algorithm. This supports rapid prototyping and unburdens the compiler writer from proving the newly implemented algorithms correct.

References

[Ag1] Agesen, O. Constraint-based type inference and parametric polymorphism, In *Proceedings of the 1st International Static Analysis Symposium (SAS'94) (Namur, Belgium)*, Springer-Verlag, Heidelberg, LNCS 864 (1994), 70 - 100.

[Ag2] Agesen, O. The cartesian product algorithm. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95) (Aarhus, Denmark)*, Springer-Verlag, Heidelberg, LNCS 952 (1995), 2 - 26.

[AH] Agesen, O., and Hölzle, U. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the 10th ACM SIGPLAN Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), ACM SIGPLAN Notices 30*, 10, (1995), 91 - 107.

[BI] Borning, A. H., and Ingalls, D. H. H. A type declaration and inference system for Smalltalk. In *Conference Record of the 9th International Symposium on Principles of Programming Languages (POPL'82) (Albuquerque, New Mexico)*, 1982, 133 - 141.

[CC] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL'77) (Los Angeles, California)*, 1977, 238 - 252.

[Ch] Chambers, C. The design and implementation of the SELF Compiler, an optimizing compiler for object-oriented programming languages. Dissertation, Stanford University, 1992.

[DGC] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95) (Aarhus, Denmark)*, Springer-Verlag, Heidelberg, LNCS 952 (1995), 77 - 101.

[Go] Golubski, W. Type analysis in object–oriented languages - The crucial points. In *Proceedings of the 10th International Symposium on Computer* *and Information Sciences* (*ISCIS X*) (*Izmir, Turkey*), Vol. 2, (1995), 659-665.

[GJ] Graver, J. O., and Johnson, R. E. A type system for Smalltalk. In *Conference Record of the 17th International Symposium on Principles of Programming Languages* (*POPL'90*) (*San Francisco, California*), 1990, 136 - 150.

[HCU] Hölzle, U., Chambers, C., and Ungar, D. Optimizing dynamically–typed object-oriented languages with inline caches. In *Proceedings of the European Conference on Object-Oriented Programming* (*ECOOP'91*) (*Geneva, Switzerland*), Springer-Verlag, Heidelberg, LNCS 512 (1991), 21 - 38.

[HU] Hölzle, U., and Ungar, D. Optimizing dynamically–dispatched calls with run–time feedback. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (*PLDI'94*) (*Orlando, Florida*), *ACM SIGPLAN Notices 29*, 6 (1994), 326 - 336.

[Jo] Johnson, R. E. Type-checking Smalltalk. In *Proceedings of the 1st ACM SIGPLAN Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'86*) (*Portland, Oregon*), *ACM SIGPLAN Notices 21*, 11, (1986), 315 - 321.

[KG1] Knoop, J., and Golubski, W. Abstract interpretation: A uniform approach for type analysis and classical optimization of object-oriented programs. In *Proceedings of the 1st International Symposium on Object-Oriented Technology "The White OO Nights"* (*WOON'96*) (*St. Petersburg, Russia*), June 20 - 21, 1996. Available via anonymous ftp: ftp.informatik.uni-stuttgart.de/pub/eiffel/WOON_96

[KG2] Knoop, J., and Golubski, W. Precise type analysis by means of abstract interpretation. In *joint Proceedings of the 13th Workshop of the GI-FG 2.1.4 "Alternative Konzepte für Sprachen und Rechner" and the 15th Workshop of the GI-ITG-FG 3.1.2 "Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware* (*PARS*)*" on "Programmiersprachen und hardware-nahe Programmierung"* (*Physikzentrum Bad Honnef, Germany*), Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen Nr. 15, (1996), 59 - 68.

[KS] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction* (*CC'92*) (*Paderborn, Germany*), Springer-Verlag, Heidelberg, LNCS 641 (1992), 125 - 140.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Towards a tool kit for the automatic generation of interprocedural data flow analyses. To appear in *Journal of Programming Languages*.

[KRS2] Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. *Transactions on Programming Languages and Systems 16*, 4 (1994), 1117 - 1155.

[KRS3] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94) (Orlando, Florida), ACM SIGPLAN Notices 29*, 6 (1994), 147 - 158.

[KRS4] Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95) (La Jolla, California), ACM SIGPLAN Notices 30*, 6 (1995), 233 - 245.

[KRS5] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

[KU] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, (1977), 309 - 317.

[La] Larchevêque, J. M. Interprocedural type propagation for object-oriented languages. In *Science of Computer Programming 22*, (1994), 257 - 282.

[Ma] Marriot, K. Frameworks for abstract interpretation. *Acta Informatica 30*, (1993), 103 - 129.

[MJ] Muchnick, S. S., and Jones, N. D. (Eds.). Program flow analysis: Theory and applications. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[OPS] Oxhøj, N., Palsberg, J., and Schwartzbach, M. I. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92) (Utrecht, The Netherlands)*, Springer-Verlag, Heidelberg, LNCS 615 (1992), 329 - 349.

[PC1] Plevyak, J., and Chien, A. A. Precise concrete type inference for object-oriented languages. In *Proceeings of the 9th ACM SIGPLAN Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94) (Portland, Oregon), ACM SIGPLAN Notices 29*, 10, (1994), 324 - 340.

[PC2] Plevyak, J., and Chien, A. A. Iterative flow analysis. Unpublished manuscript. Available from http://www-csag.cs.uiuc.edu/.

[PR] Pande, H.D., and Ryder, B.G. Static Type Determination and Aliasing for C++. *Technical Report LCSR-TR-236, Rutgers University*, (1994).

[PS1] Palsberg, J., and Schwartzbach, M. I. Object-oriented type inference. In *Proceedings of the 6th ACM SIGPLAN Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'91*) (*Phoenix, Arizona*), *ACM SIGPLAN Notices 26*, 11 (1991), 146 - 161.

[PS2] Palsberg, J., and Schwartzbach, M. I. Object-oriented type systems. John Wiley & Sons, 1994.

[St] Steffen, B. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming 21*, (1993), 115 - 139.

[Su] Suzuki, N. Inferring types in Smalltalk. In *Conference Record of the 8th International Symposium on Principles of Programming Languages* (*POPL'81*) (*Williamsburg, Virginia*), 1981, 187 - 199.

[SP] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. In [MJ], 1981, 189 - 233.

[VHU] Vitek, J., Horspool, R. N., and Uhl, J. S. Compile-time analysis of object–oriented programs. In *Proceedings of the 4th International Conference on Compiler Construction* (*CC'92*) (*Paderborn, Germany*), Springer-Verlag, Heidelberg, LNCS 641 (1992), 236 - 250.

A The Generic Algorithm

Algorithm A.1 (The Main Process: Computing the *OMFP*-**Solution)**

Input: *An OO-analysis model T, a complete lattice $\mathcal{C}$ satisfying the descending chain condition, a start information $c_0 \in \mathcal{C}$, and for every transition e of T the monotonic local semantic function $[\![\, e \,]\!]$ as computed by Algorithm A.2. Additionally, for each transition $e \in E_{mc}$ and every class $C \in Impl(Method(e))$ the local semantic functions $[\![\, e' \,]\!]^*$ for $e' \in \{e^C_{pt}, e^C_{fil}\}$, and in case of $e \in E^{ass}_{mc}$, additionally the local semantic function $[\![\, e^C_{ass} \,]\!]^*$.*

Output: *An annotation of T with the OMFP-solution (stored in dfi).*

Remark: *The variable workset controls the iterative process. Its elements are states of T. The auxiliary variable meet stores the result of the most recent meet operation.*

(*Initialization of the annotation array dfi and the variable workset*)
FORALL $n \in N \backslash \{start(mainMethod)\}$ **DO** $dfi[n] := \top$ **OD**;
$dfi[start(mainMethod)] := c_0$;
$workset := \{\, start(mainMethod) \,\}$;

(*Iterative fixed point computation*)
WHILE $workset \neq \emptyset$ **DO**
  LET $m \in workset$
    BEGIN
      $workset := workset \backslash \{ m \}$;
      (*Update the "environment" of node* m)
      FORALL $e \in E$ *with* $source(e) = m$ **DO**
        $meet := dfi[dest(e)] \sqcap [\![e]\!](dfi[source(e)])$;
        IF $dfi[dest(e)] \sqsupset meet$
          THEN $dfi[dest(e)] := meet$; $workset := workset \cup \{ dest(e) \}$ **FI**;
        IF $e \in E_{mc}$
          THEN
            FORALL $s \in start(Impl(Method(e)))$ **DO**
              $meet := dfi[s] \sqcap [\![e_{pt}^{Class(s)}]\!]^* \circ [\![e_{fil}^{Class(s)}]\!]^*(dfi[source(e)])$;
              IF $dfi[s] \sqsupset meet$
                THEN $dfi[s] := meet$; $workset := workset \cup \{ s \}$ **FI**
            OD
        FI
      OD
    END
OD.

Algorithm A.2 (The Preprocess: Computing the Effects of Methods)

Input: *An OO-analysis model* T, *a complete lattice* $\mathcal{C}$, *whose function lattice* $[\,\mathcal{C} \rightarrow \mathcal{C}\,]$ *satisfies the descending chain condition, and for every transition* $e \in E^*$ *of the OO-analyis graph of* T *a monotonic local semantic function* $[\![e]\!]^*$.

Output: *An annotation of* T *with functions* $[\![\![n]\!]\!] : \mathcal{C} \rightarrow \mathcal{C}$ (*stored in gtr, which stands for* global transformation), *and* $[\![e]\!] : \mathcal{C} \rightarrow \mathcal{C}$ (*stored in ltr, which stands for* local transformation) *representing the greatest solution of the equation system of Definition 4.1.*

Remark: *The variable workset controls the iterative process. Its elements are tuples, whose first components are states* $m \in N$ *of the OO-analysis model* T, *and whose second components are functions* $f : \mathcal{C} \rightarrow \mathcal{C}$, *which specify a new approximation for the function* $[\![\![m]\!]\!]$ *of the node of the first component. Note that due to the mutual interdependence of the definitions of* $[\![\![\ ]\!]\!]$ *and* $[\![\ ]\!]$ *the iterative approximation of* $[\![\![\ ]\!]\!]$ *is superposed by an "interprocedural" iteration step, which updates the current approximation of the effect* $[\![\ ]\!]$ *of call transitions. The auxiliary variable meet stores the result of the most recent meet operation.*

(*Initialization of the annotation arrays gtr and ltr and the variable workset*)
FORALL $n \in N$ **DO** $gtr[n] := \top_{[\mathcal{C} \rightarrow \mathcal{C}]}$ **OD**;
FORALL $e \in E$ **DO**
  IF $e \in E_{mc}^{ass}$

$$
\begin{array}{l}
\quad\quad \mathbf{THEN} \\
\quad\quad\quad ltr[e] := [\![\, e_{SthOutOf} \,]\!]^* \sqcap \\
\quad\quad\quad\quad \bigsqcap\{\, [\![\, e^C_{ass} \,]\!]^* \circ [\![\, end(Model(Method(e), C)) \,]\!] \circ [\![\, e^C_{pt} \,]\!]^* \circ [\![\, e^C_{fil} \,]\!]^* \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid C \in Impl(Method(e)) \,\} \\
\quad \mathbf{ELSEIF}\ \ e \in E^{pure}_{mc} \\
\quad\quad \mathbf{THEN} \\
\quad\quad\quad ltr[e] := [\![\, e_{SthOutOf} \,]\!]^* \sqcap \\
\quad\quad\quad\quad \bigsqcap\{\, [\![\, end(Model(Method(e), C)) \,]\!] \circ [\![\, e^C_{pt} \,]\!]^* \circ [\![\, e^C_{fil} \,]\!]^* \mid C \in Impl(Method(e)) \,\} \\
\quad\quad \mathbf{ELSE} \\
\quad\quad\quad ltr[e] := [\![\, e \,]\!]^*\ \ \mathbf{FI} \\
\mathbf{OD}; \\
workset\ := \{\, (s, Id_{\mathcal{C}}) \mid s \in start(\mathcal{M}_{Mdl}) \,\}; \\
\\
(\,\textit{Iterative fixed point computation}\,) \\
\mathbf{WHILE}\ workset \neq \emptyset\ \ \mathbf{DO} \\
\quad \mathbf{LET}\ \ (m, f) \in workset \\
\quad\quad \mathbf{BEGIN} \\
\quad\quad\quad workset := workset \setminus \{\, (m, f) \,\}; \\
\quad\quad\quad meet := gtr[m] \sqcap f; \\
\quad\quad\quad \mathbf{IF}\ \ gtr[m] \sqsupset meet \\
\quad\quad\quad\quad \mathbf{THEN} \\
\quad\quad\quad\quad\quad gtr[m] := meet; \\
\quad\quad\quad\quad\quad \mathbf{IF}\ \ m \in \{end(\mathcal{M}_{Mdl})\} \\
\quad\quad\quad\quad\quad\quad \mathbf{THEN} \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{FORALL}\ \ e \in Sender(mm(m))\ \ \mathbf{DO} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{IF}\ \ e \in E^{ass}_{mc} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{THEN} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ltr[e] := [\![\, e_{SthOutOf} \,]\!]^* \sqcap \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \bigsqcap\{\, [\![\, e^C_{ass} \,]\!]^* \circ [\![\, end(Model(Method(e), C)) \,]\!] \circ \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [\![\, e^C_{pt} \,]\!]^* \circ [\![\, e^C_{fil} \,]\!]^* \ \mid\ C \in Impl(Method(e)) \,\} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{ELSE} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ltr[e] := [\![\, e_{SthOutOf} \,]\!]^* \sqcap \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \bigsqcap\{\, [\![\, end(Model(Method(e), C)) \,]\!] \circ [\![\, e^C_{pt} \,]\!]^* \circ [\![\, e^C_{fil} \,]\!]^* \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid\ C \in Impl(Method(e)) \,\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{FI}; \\
\quad\quad\quad\quad\quad\quad\quad\quad workset := workset \cup \{\, (dest(e), ltr[e] \circ gtr[source(e)]) \,\} \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{OD} \\
\quad\quad\quad\quad\quad\quad \mathbf{ELSE} \\
\quad\quad\quad\quad\quad\quad\quad workset := workset \cup \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{\, (dest(e), ltr[e] \circ gtr[m]) \mid e \in E\ \ with\ \ source(e) = m \,\} \\
\quad\quad\quad\quad\quad \mathbf{FI} \\
\quad\quad\quad \mathbf{FI} \\
\quad\quad \mathbf{END} \\
\mathbf{OD}.
\end{array}
$$