Abstract Interpretation: A Uniform Framework for Type Analysis and Classical Optimization of Object-Oriented Programs

Jens Knoop * Wolfgang Golubski †

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. *Classical optimization* is concerned with improving the efficiency of "conventional" procedural programs, e.g. by avoiding unnecessary executions and computations of statements and values at run-time. In this article we enhance the framework of *abstract interpretation* to the object-oriented setting, and demonstrate that it uniformly supports type analysis and classical optimization of object-oriented programs. This is demonstrated by means of a conceptually new type analysis based on abstract interpretation, which is exceptional for its power due to its almost deterministic treatment of program branches and method sends, and by constructing an algorithm, which is unique to place primitive method calls *computationally optimally* in a program, and which directly evolves from its interprocedural *code motion* counterpart. Moreover, it is demonstrated that type analysis is not a prerequisite of classical optimization in our approach, but in general it enhances its power.

Keywords: (Untyped) object-oriented languages, type analysis, abstract interpretation, data flow analysis, (classical) program optimization, *MOP*-Solution, *MFP*-Solution, (Object-Oriented) Coincidence Theorem, code motion, partial redundancy elimination, Smalltalk.

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*. Thus, replacing late by early binding where possible is 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 *type analysis*.

Classical optimization is concerned with improving the efficiency of programs of "conventional" procedural languages. The efficiency gain of typical optimizations like *code motion* (cf. [MR]), *dead code elimination* (cf. [KRS3]), *assignment motion* (cf. [KRS4]), or *constant propagation* (cf. [KU]) stems from avoiding unnecessary executions and computations of statements and values at run-time. *Data flow analysis* (*DFA*) computes here the side-conditions under which the transformations are correct and profitable. Of course, this requires information which is outside the scope of type analysis.

*Fachbereich 10 — Informatik, Carl von Ossietzky Universität Oldenburg, Uhlhornsweg 84, D-26129 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, Germany. The work of the author is partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

†Universität-GH Siegen, Fachbereich 12 — Elektrotechnik & Informatik, Hölderlinstraße 3, D-57068 Siegen, Germany. E-mail: golubski@informatik.uni-siegen.de.

Abstract interpretation has proved to be a general and theoretically well-founded framework for static program analysis (cf. [CC, Ma]). In this article we adapt the framework of abstract interpretation to the object-oriented setting, which in spirit is quite similar to the interprocedural extensions of [SP] and [KS], and demonstrate that it uniformly supports type analysis and classical optimization of object-oriented programs.

This supports a two-step approach for optimization: First, type analysis in order to replace late by early binding, where possible. And second, classical optimization. In fact, by means of the new framework all the classical optimization techniques, which are based on abstract interpretation, can now be made available for object-oriented programs. Moreover, it is worth noting that type analysis is *not* a prerequisite of classical optimization in our approach, but in general precomputed type information enhances its power.

In this article we demonstrate this by developing a conceptually new type analysis based on abstract interpretation, which is exceptional for its power due to its almost deterministic treatment of program branches and method sends, and by constructing an algorithm for the *computationally optimal* placement of primitive method calls in a program. This algorithm, which is unique to achieve this result, evolves directly from its intra- and interprocedural code motion counterparts of [KRS2] and [KRS5].

The accuracy of type analysis depends strongly on the treatment of program branches and method sends. A nondeterministic treatment is a major source for introducing imprecisions into type analysis. The analysis we present achieves an almost deterministic treatment by means of *filter* functions. 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 type analysis to detect *dead code* in the argument program, i.e., code which will never be executed, an information from which it directly profits as dead program parts cannot invalidate computed type information.

Moreover, it is an important point of our approach that it provides a generic algorithm, which can directly be fed with the concise specifications of the type analysis and the data flow analyses required by classical optimizations. Thus, the designer of an analysis need not to construct and implement for each analysis a new algorithm himself. This supports rapid prototyping and unburdens him from proving his implementation correct. In fact, he need not even to know any details of the generic algorithm when applying the framework.

Related Work: There is a broad variety of approaches for computing type information in object-oriented languages. Particularly important are type inference based approaches as e.g. in [Ag, BI, GJ, OPS, PS1, PS2, Su], and data flow analysis based approaches as e.g. in [PC, VHU].

The algorithms of [Ag, OPS, PS1, PS2] characterize the state-of-the-art of the type inference based approaches. They improve on previous type inference algorithms in that they take the context of message sends into account in order to improve the accuracy of the derived type information. However, as it is typical for type inference, they do not perform a flow-sensitive analysis, which is addressed by the data flow based approaches. Here, the algorithm of [PC] is exceptional in that it heuristically focusses the type analysis to program parts with "high" impact on the analysis results. The price for the improved precision is an exponential worst case time complexity.

Conceptually completely different and incomparable to our approach are the approaches of [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.

All these algorithms are specifically designed for type analysis, and do not address classical optimization. This also holds for the conceptually different approach of hierarchy analysis of [DGC]. On the other hand, there are few approaches dealing with classical optimization of object-oriented programs as e.g. the one of [La], which presents a type analysis based on definition-use chains, and sketches an application to constant propagation. However, like the

other approaches also the approach of [La] focusses on algorithmic aspects, and does not systematically address correctness issues. In contrast, this is a central concern of our approach.

Structure of the Paper: After the preliminary Section 2, in which we present our Smalltalk-like programming language *SM*, we introduce in Section 3 OO-analysis models and OO-analysis graphs as representations of *SM*-programs which are appropriate for the formal development of our framework. In Section 4 we generalize the framework of abstract interpretation to the object-oriented setting. Subsequently, we present our type analysis in Section 5, and show how to adapt classical optimizations to the object-oriented setting in Section 6 considering the computationally optimal placement of primitive method calls in a program for illustration. Section 7, finally, 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. Its syntax is defined in terms of a context-free grammar in Figure 1. In essence, *SM* is a slightly enriched variant of the Smalltalk-like programming language investigated in [PS1]. As in [PS1], a *program* is a set of classes followed by an *expression* called *main program*. *Classes* can be defined by means of inheritance. As in [PS1], metaclasses and blocks are omitted, which allows us to concentrate on the central features of our approach. However, in addition to the language of [PS1], *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 run-time error, and which checks whether its arguments denote the same object. In this case, the test returns the value `nil`; otherwise, 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.

(*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	::=	$SE \mid SE$ `instanceOf` $: ClassId \mid Id_1\ =\ Id_2$
(*Expression*)	E	::=	$SEMC \mid Id := SEMC \mid E_1; E_2 \mid$
			`if` SC `then` E_1 `else` $E_2 \mid$
			`while` SC `do` $E \mid$ `repeat` E `until` SC

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

As in [PS1], we assume that inheritance is removed from an argument program, which can be achieved by a simple pre-compilation step (cf. [PS2]). In essence, this step 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. In general this leads to better results of our type analysis, but it is worth noting that it is not a prerequisite of our approach.

SM-programs are represented in the following by means of transition system like graphs, which we call *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 the end node of a method model has exactly one incoming transition, which is not labelled at all.[2]

Figure 2 shows an OO-program model for illustration, which will be used as the running example throughout this article. It consists of the class declarations *Main*, A, B, and C, where each of the classes A, B, and C implements a local method m, and is complex enough to illustrate the central features of our approach.

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.

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.

Figure 3 shows the OO-analysis model corresponding to the OO-program model of Figure 2. While this representation (even in the presence of recursion) is perfectly well-suited for the denotational characterization of our analyses, their operational characterization specifying their intuitively desired reference solutions, relies 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 an *OO-analysis graph* $T^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$. In essence, T^* results from 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.

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 run-time 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,

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

[2] If necessary, such a transition can always be added.

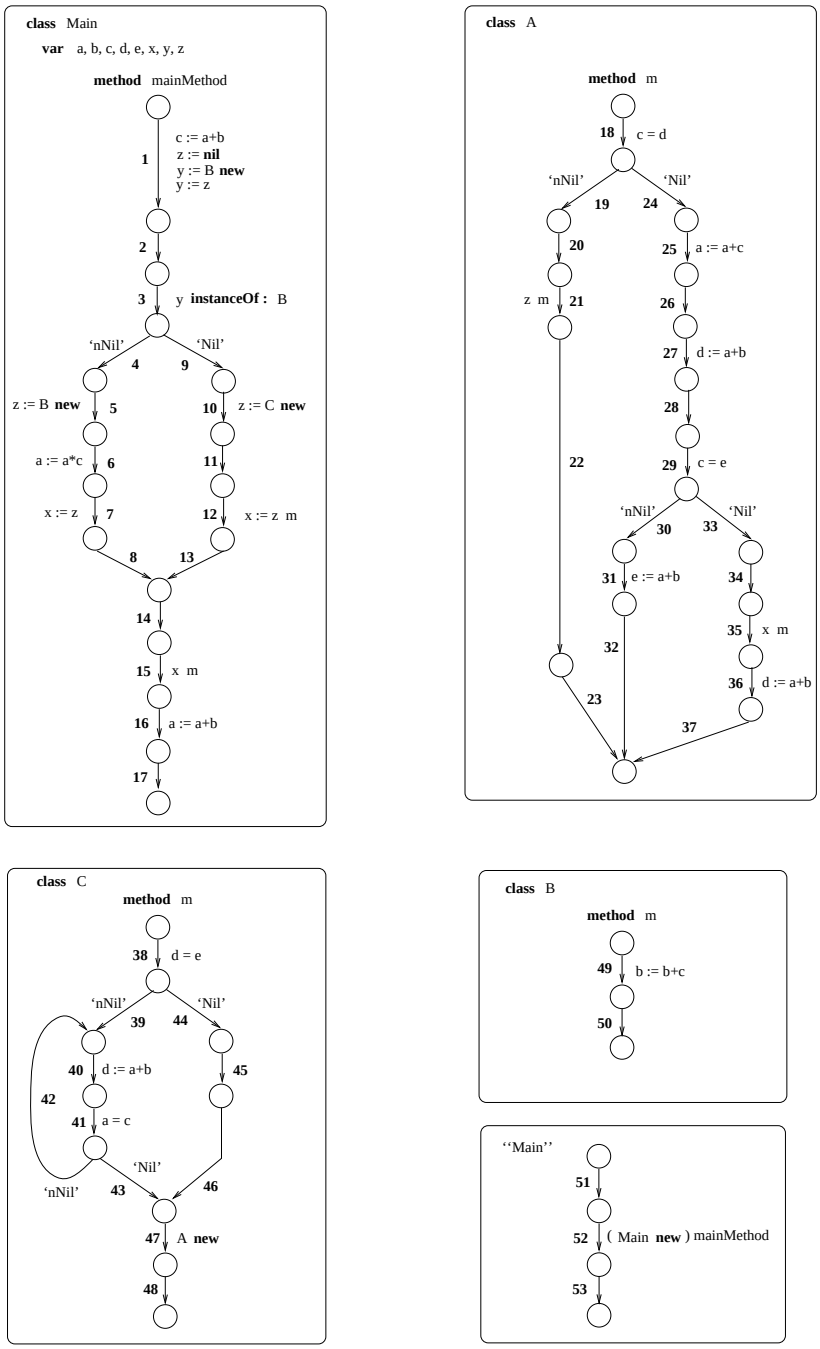

Figure 2: The OO-Program Model of the Running Example

identifying matching call and return transitions with "(" and ")", respectively, the set of valid paths corresponds to the prefix-closed language of balanced parentheses. 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.

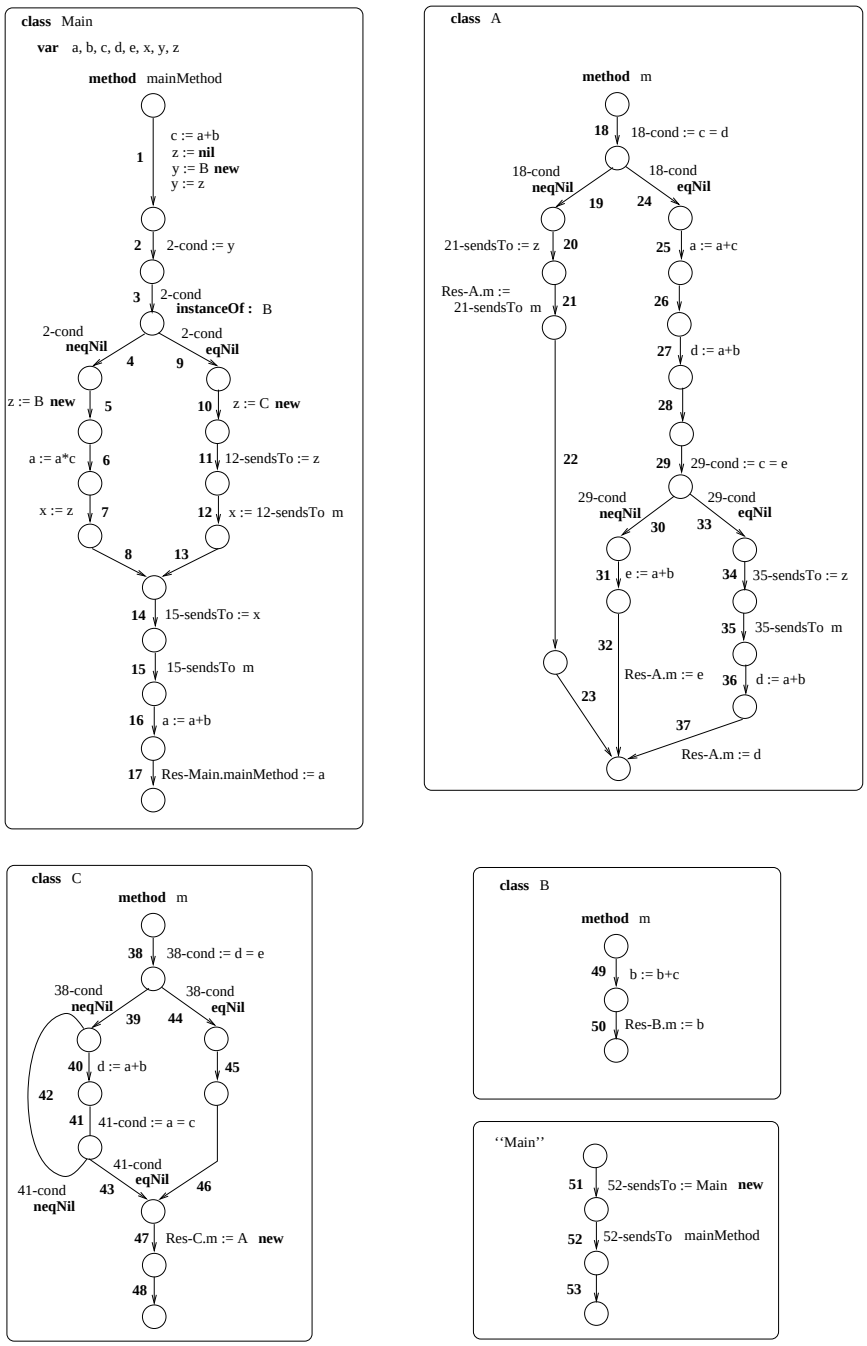

Figure 3: The OO-Analysis Model of the Running Example

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 $\top$, whose 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.

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:

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

The *OMOP*-solution directly reflects our desires. In general, however, it does not specify an effective computation procedure.[3]

Second Globalization: The Maximal Fixed Point Semantics

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. This is quite similar to interprocedural DFA (cf. [KS, SP]). It 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 $[\![\, end(m) \,]\!]$, where $end(m)$ 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:*

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

and

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

where $Id_{\mathcal{C}}$ 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 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^{pure}_{mc} (E^{ass}_{mc}) 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 appropiate name of the different classes of T ($\bigsqcap\{\ldots\}$), and of the effect

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

of a probably not understood message ($[\![e_{SthOutOf}]\!]^*$). Note that the steps, in which the effect of calling a method is computed, directly reflect the phases of executing a method call: *Filtering* the information ($[\![e^C_{fil}]\!]^*$), *transferring* the parameters ($[\![e^C_{pt}]\!]^*$), *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 ($[\![e^C_{ass}]\!]^*$).

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)

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

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:

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

As in intra- and interprocedural DFA the *OMFP*-approach is practically relevant because it induces an iterative computation procedure for computing the *OMFP*-solution. This is effective if the function lattice satisfies the descending chain condition, and if the local semantic functions are monotonic. The question of correctness or even optimality (coincidence) of the *OMFP*-solution with respect to the *OMOP*-solution is answered in the following section, where we present the object-oriented generalizations of the (intraprocedural) Safety and Coincidence Theorems of Kam and Ullman [KU].

Correctness and Coincidence

Analogously to their conventional counterparts, the OO-Safety Theorem and the OO-Coincidence Theorem give sufficient conditions for the correctness and the coincidence of the *OMFP*-solution with respect to the *OMOP*-solution. In fact, 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.* $\forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ OMFP_{([\![\ ]\!]^*, c_0)}(n) \sqsubseteq OMOP_{([\![\ ]\!]^*, c_0)}(n)$, *if the local semantic functions* $[\![e]\!]^*$, $e \in E^*$, *are all monotonic.*

Theorem 4.4 (OO-Coincidence Theorem) *The OMFP-solution and the OMOP-solution coincide, i.e.* $\forall\, c_0 \in \mathcal{C}\ \forall\, n \in N.\ OMFP_{([\![\ ]\!]^*, c_0)}(n) = OMOP_{([\![\ ]\!]^*, c_0)}(n)$, *if the local semantic functions* $[\![e]\!]^*$, $e \in E^*$, *are all distributive.*

5 The Type Analysis

In this section we present our type analysis for an arbitrary, but fixed OO-analysis model T and its corresponding OO-analysis graph T^* within the framework of Section 4. Let $\mathcal{C}' =_{df} \mathcal{C}_{Id} \cup \{Int, Nil, nNil\}$ 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^*. Moreover, let $\mathcal{PS}$ denote the powerset lattice $\mathcal{P}(\mathcal{C}')$, which is enriched by an artificial least element $\bot_{ps}$ and a greatest element $\top_{ps}$. Central is now the definition of the local abstract semantics, which can directly be fed into the generic algorithm of Section A computing the *OMFP*-solution.

Local Abstract Semantics

Data Domain: The data domain of our type analysis is given by the function lattice $\mathcal{FL} =_{df} [\mathcal{V}' \to \mathcal{PS}]$, where the meet operation is given by pointwise set union:

$$(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{FL}, \cup, \cap, \supseteq, \top_{fl}, \bot_{fl})$$

Here, $\top_{fl}$ and $\bot_{fl}$ are defined by: $\forall\, v \in \mathcal{V}'.\ \top_{fl}(v) =_{df} \top_{ps}$ and $\bot_{fl}(v) =_{df} \bot_{ps}$

Intuitively, program points, which are annotated with $\bot_{fl}$ after the termination of the type analysis are not reachable at run-time, and thus are dead code. Conversely, the existence of program points which are annotated with $\top_{fl}$ after the termination indicates that a message call has not been understood. In the following we use the convention that if $M \in \mathcal{P}(\mathcal{C}')$, $v \in \mathcal{V}'$, and $f \in \mathcal{FL} \backslash \{\top_{fl}, \bot_{fl}\}$, then $f[M \backslash v]$ is the unique function $f[M \backslash v]$ of $\mathcal{FL}$, which is defined by $f[M \backslash v](x) = M$, if $x = v$, and by $f[M \backslash v](x) = f(x)$, otherwise.

Local Semantic Functional: The local semantic functional of our type analysis is given by the functional

$$[\![\ ]\!]^* : E^* \to (\mathcal{FL} \to \mathcal{FL})$$

which is defined by:

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

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

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

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \\ f[\{Int\} \backslash Id] & \text{if } f(Id_1) = f(Id_2) = \{Int\} \\ \top_{fl} & \text{otherwise} \end{cases}$$

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

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \text{ or } \{C\} = f(e\text{-}cond) \\ f[\{Nil\} \backslash e\text{-}cond] & \text{if } C \notin f(e\text{-}cond) \\ f[\{C, Nil\} \backslash e\text{-}cond] & \text{otherwise} \end{cases}$$

  - $e\text{-}cond := Id_1\ =\ Id_2$:

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \\ f[\{Nil, nNil\} \backslash e\text{-}cond] & \text{otherwise} \end{cases}$$

- *Expressions:* if e represents an expression of the form SE: $[\![e]\!]^*(f) =_{df} f$
- *Filters:* if e represents a filter of the form

– *e-sendsTo* $C.m_1 : \ldots m_n$:

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad C \in f(\textit{e-sendsTo}) \\ \bot_{fl} & \text{otherwise} \end{cases}$$

– *e-sendsTo* `SthOutOf` $\mathit{Impl}(\mathit{Method}(e))$

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \\ \top_{fl} & \text{if } f(\textit{e-sendsTo}) \setminus \mathit{Impl}(\mathit{Method}(e)) \neq \emptyset \\ \bot_{fl} & \text{otherwise} \end{cases}$$

– *e-cond* `eqNil`

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad \mathit{Nil} \in f(\textit{e-cond}) \\ \bot_{fl} & \text{otherwise} \end{cases}$$

– *e-cond* `neqNil`

$$[\![e]\!]^*(f) =_{df} \begin{cases} f & \text{if } f \in \{\bot_{fl}, \top_{fl}\} \text{ or} \\ & \quad f(\textit{e-cond}) \setminus \{\mathit{Nil}\} \neq \emptyset \\ \bot_{fl} & \text{otherwise} \end{cases}$$

Start Information: The start information of our type analysis is given by the function $f_0 \in \mathcal{FL}$, which is defined by $f_0(v) = \{\mathit{Nil}\}$ for all $v \in \mathcal{V}'$.

Filter transitions and the treatment of assignments are two major reasons for the power of our type analysis.

First, filter transitions allow us an almost deterministic treatment of branch and method-send transitions. Actually, we do not know of any other approach achieving this. Intuitively, a filter "blocks" the propagation of type information, if it is 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 only, if C is a receiver of the current message. Otherwise, the type information is blocked, which is technically achieved by propagating the trivial information $\bot_{fl}$ 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` $\mathit{Impl}(\mathit{Method}(e))$, which have to deal with method calls being not understood. This situation arises, if the receiver of a method call is an object of a class outside of $\mathit{Impl}(\mathit{Method}(e))$. This occurs, if $f(\textit{e-sendsTo}) \setminus \mathit{Impl}(\mathit{Method}(e)) \neq \emptyset$. In this case the filter propagates the element $\top_{fl}$ of our data domain, which indicates this error situation. Similarly, assignments with a primitive method call on their right hand side propagate $\top_{fl}$, if the method call is invoked with non-integer operands.

Second, the abstract semantics of assignments reflect that the effect of an assignment is "destructive", i.e., the previously stored value is overwritten by the value of the right hand side expression. This is in contrast to many approaches for type analysis where assignments are treated in a "cumulative" or "collecting" fashion. This, however, is not only unintuitive regarding the ordinary semantics of assignments, it is also a major source for introducing imprecision into the results of type analysis; a problem, which is usually heuristically dealt with by means of "global" and "local" types.

Correctness of the Type Analysis

In order to prove the correctness of our type analysis, we must show that it is a safe approximation of its *OMOP*-solution. The following simple lemma is here the key:

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

In fact, combining Lemma 5.1 with the Coincidence Theorem 4.4, we obtain as desired:

Theorem 5.2 (Correctness) *The OMFP-solution of our type analysis is a safe approximation of its OMOP-solution.*

Moreover, the generic algorithm of Section A terminates with the *OMFP*-solution. Essentially, this is a consequence of the monotonicity of the local semantic functions and the fact that $\mathcal{FL}$ satisfies the descending chain condition, which is trivial because it is finite. Note that in general the *OMFP*-solution of our type analysis does not coincide with its *OMOP*-solution because the semantic functions of filter transitions are not distributive.

The Running Example

Figure 4 high-lightens some of the most important results of our type analysis. It detects that the left branch following transition **3** is never taken because the states of the left branch are annotated by the initial information $\bot_{fl}$ after its termination. Hence, the transitions $\mathbf{4}, \ldots, \mathbf{9}$ are identified as *dead code* and can be eliminated. As a consequence, the transitions **2**, **3**, and **9** become superfluous, and can be replaced by a new transition $(dest(\mathbf{1}), source(\mathbf{10}))$ as illustrated in Figure 5. Moreover, the type analysis 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 with the information that the left branch of method *mainMethod* is dead code, our analysis detects that at the transitions **15** and **35** method m of an object of class A is invoked. Additionally, it reveals that method m of class B is never invoked. Thus, for the running example our type analysis does not only guarantee the absence of run-time errors due to messages that are not understood, it even gives precise type information for every message-send site.

6 Classical Optimization

In this section we are concerned with showing how to extend by means of the framework of Section 4 optimization techniques originally developed for conventional procedural languages to the OO-setting. It is worth noting that type information is not a prerequisite for this, but in general it enhances the power of the optimizing transformations because method sends can be handled more appropriately. Moreover, if type information is given, dead and superfluous code identified by the type analysis can be eliminated. Remaining branches are then nondeterministically treated, and method sends as nondeterministic *higher order branch statements* according to the type information given. Technically, this is achieved by replacing in the equation system of Definition 4.1 the set $Impl(Method(e))$ by the (smaller) set of classes given by the type information, and by analogously adapting the set *Sender* in Equation System 4.2. Subsequently, the process proceeds as in interprocedural DFA and optimization.

In the remainder of this section we demonstrate this by constructing an OO-version of an algorithm for *code motion*, which is a well-known powerful classical optimization.[4] Intuitively, code motion improves the run-time efficiency by avoiding unnecessary recomputations of values at run-time. This is achieved by replacing the original computations of a program by temporaries that are initialized at certain program points. For conventional intra- and interprocedural programs it is well-known that placing the computations *as early as possible* in a program, while maintaining its semantics, leads to computationally optimal results (cf. [KRS1, KRS2, KRS5]). This carries over to the OO-setting. The point here is that a primitive method call for multiplication or addition has the same effect than a term computation in a conventional program.

[4] Code motion is also called *partial redundancy elimination* (cf. [MR]).

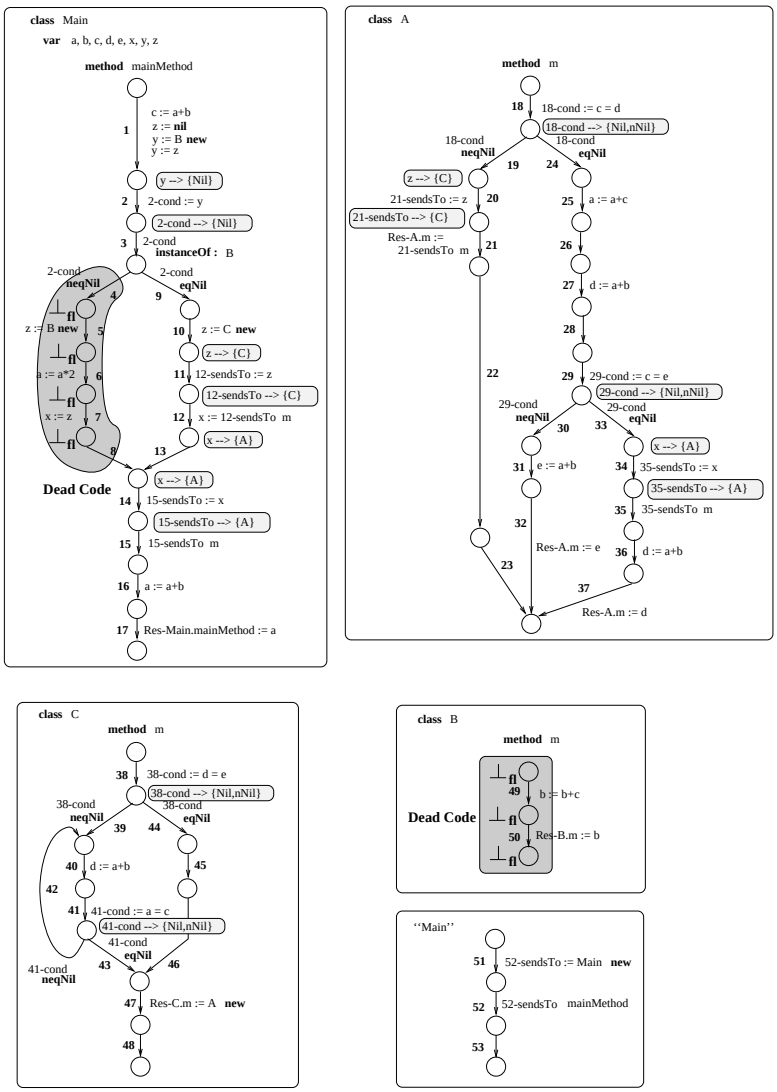

Figure 4: The Central Results of the Type Analysis

The power of the new algorithm is demonstrated by the example of Figure 2, where it is, based on the results of our type analysis, unique to achieve the optimization result of Figure 5. It eliminates all partially redundant occurrences of $a + b$ at the transitions **16**, **31**, and **40** by a reference to a new temporay **h** that is initialized at the transitions **1** and **26**.

As in the interprocedural setting the as-early-as-possible placement of computations is based on the computation of the set of program points where a computation is *up-safe* and *down-safe*, i.e., where it has been computed on every program path reaching the program point under consideration, and where it will be computed on every program continuation reaching the program's end node.[5] The DFA-problems for up-safety and down-safety are completely specified by the local semantic functionals $[\![\ ]\!]_{us}$ and $[\![\ ]\!]_{ds}$, where *Comp* and *Transp* are two local predicates, which are true for a transition e with respect to a computation t, if t occurs in the right hand side term of the statement of e, and if no operand of t is modified by it, respectively.

$$[\![e]\!]^*_{us} =_{df} \begin{cases} Const_{true} & \text{if } Transp(e) \wedge Comp(e) \\ Id_{\mathcal{B}} & \text{if } Transp(e) \wedge \neg Comp(e) \\ Const_{false} & \text{otherwise} \end{cases}$$

[5] Up-safety and down-safety are also known as *availability* and *anticipability (very busyness)*, respectively (cf. [MR]).

$$[\![e]\!]^*_{ds} =_{df} \begin{cases} Const_{true} & \text{if } Comp(e) \\ Id_{\mathcal{B}} & \text{if } \neg Comp(e) \wedge Transp(e) \\ Const_{false} & \text{otherwise} \end{cases}$$

These functionals can directly be fed into the generic algorithm of Appendix A. Moreover, as the semantic functions are all distributive, the computed *OMFP*-solutions for down-safety and up-safety coincide with their desired *OMOP*-solutions (cp. OO-Coincidence Theorem 4.4).

Then, down-safe start states are "earliest" (see $source(\mathbf{1})$), as well as other down-safe states which either have an "unsafe" predecessor or an incoming transition modifying an operand of the computation under consideration (see $source(\mathbf{26})$). After inserting an initialization statement at each earliest state, all original computations can be *replaced* by the corresponding temporary. For the running example, this results in the promised program of Figure 5, where it is easy to check that it is computationally optimal with respect to $a + b$.

7 Conclusions

Type analysis and classical optimization are two different means for improving the efficiency of untyped object-oriented programs. In this article we adapted the framework of abstract interpretation to the object-oriented setting and demonstrated that it uniformly supports type analysis and optimization. This supports a modular two-step procedure for optimization: First, type analysis, and second, optimization. Though precomputed type information makes the second step more powerful, it is worth noting that it is not a prerequisite for it in our approach. This is important because the worst case time complexity of our type analysis is like that of other powerful type analyses exponential. The modular structure of our approach, however, makes it easy to integrate also other, perhaps more efficient but less precise analyses. On the other hand, it is worth noting that there is no penalty on the complexity of the object-oriented variants of classical optimizations. They have the same complexity as their interprocedural counterparts. This is particularly important for the class of bitvector based optimizations, which are most relevant in practice because of their broad range of applications including e.g. code motion, assignment motion, (partial) dead code elimination, strength reduction and definition-use-chaining. These techniques are even as efficient as in the intraprocedural base case. An implementation of our approach is in progress.

References

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

[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.

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

[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.

[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, LNCS 952 (1995), 77 - 101.

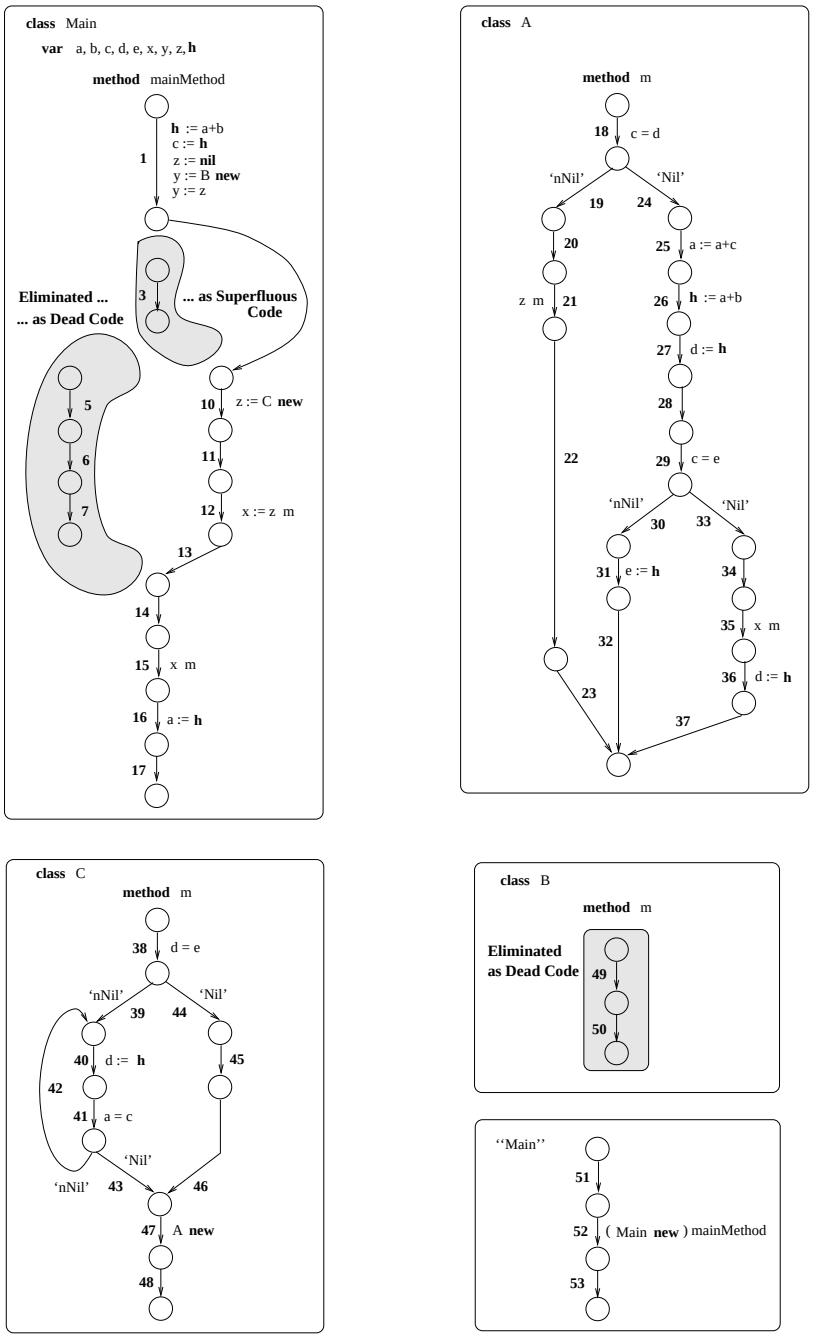

Figure 5: The Running Example after Computationally Optimal Code Motion

[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, 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.

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

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation* (*PLDI'92*) (San Francisco, California), *ACM SIGPLAN Notices 27*, 7 (1992), 224 - 234.

[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. A tool kit for constructing optimal interprocedural data flow analyses. Fakultät für Mathematik und Informatik, Universität Passau, Germany, MIP-Bericht Nr. 9413 (1994), 26 pages.

[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.

[MR] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2 (1979), 96 - 103.

[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, LNCS 615 (1992), 329 - 349.

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

[PS1] Palsberg, J., and Schwartzbach, M. I. Object-oriented type inference. In *Proceedings of the 6^{th} ACM SIGPLAN Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'91*), *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.

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

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

[VHU] Vitek, J., Horspool, R. N., and Uhl, J. S. Compile-time analysis of object–oriented programs. In *Proceedings of the 4^{th} International Conference on Compiler Construction* (*CC'92*) (Paderborn, Germany), Springer-Verlag, 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 ∈ N\{start(mainMethod)} DO dfi[n] := ⊤ OD;
dfi[start(mainMethod)] := c0;
workset := { start(mainMethod) };

( Iterative fixed point computation )
WHILE workset ≠ ∅ DO
   LET m ∈ workset
      BEGIN
         workset := workset\{ m };
         ( Update the "environment" of node m )
         FORALL e ∈ E with source(e) = m DO
            meet := dfi[dest(e)] ⊓ [[ e ]](dfi[source(e)]);
            IF dfi[dest(e)] ⊒ meet
               THEN
                  dfi[dest(e)] := meet;
                  workset := workset ∪ { dest(e) } FI;
            IF e ∈ E_mc
               THEN
                  FORALL s ∈ start(Impl(Method(e))) DO
                     meet := dfi[s] ⊓ [[ e_pt^Class(s) ]]* ∘ [[ e_fil^Class(s) ]]*(dfi[source(e)]);
                     IF dfi[s] ⊒ meet
                        THEN
                           dfi[s] := meet;
                           workset := workset ∪ { 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 $[\mathcal{C} \to \mathcal{C}]$ 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 $[\![n]\!] : \mathcal{C} \to \mathcal{C}$ (stored in gtr, which stands for* global transformation*), and $[\![e]\!] : \mathcal{C} \to \mathcal{C}$ (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 $f : \mathcal{C} \to \mathcal{C}$, 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** $gtr[n] := \top_{[\mathcal{C} \to \mathcal{C}]}$ **OD**;
FORALL $e \in E$ **DO**
  IF $e \in E^{ass}_{mc}$
    THEN
      $ltr[e] := [\![e_{SthOutOf}]\!]^* \sqcap$
        $\bigsqcap \{ [\![e^C_{ass}]\!]^* \circ [\![\, end(Model(Method(e), C)) \,]\!] \circ [\![e^C_{pt}]\!]^* \circ [\![e^C_{fil}]\!]^* \mid C \in Impl(Method(e)) \}$
  ELSEIF $e \in E^{pure}_{mc}$
    THEN
      $ltr[e] := [\![e_{SthOutOf}]\!]^* \sqcap$
        $\bigsqcap \{ [\![\, end(Model(Method(e), C)) \,]\!] \circ [\![e^C_{pt}]\!]^* \circ [\![e^C_{fil}]\!]^* \mid C \in Impl(Method(e)) \}$
    ELSE $ltr[e] := [\![e]\!]^*$ **FI**
OD;
$workset := \{ (s, id_{\mathcal{C}}) \mid s \in start(\mathcal{M}_{Mdl}) \}$;

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