Type Analysis and Type-test Elimination in Oberon-2

Jens Knoop and Falk Schreiber

Fakultät für Mathematik und Informatik – Universität Passau
Innstraße 33, D-94032 Passau, Germany
E-mail: {knoop|schreiber}@fmi.uni-passau.de

Abstract

This article shows how to enhance the well-established framework of abstract interpretation to the object-oriented setting considering an Oberon-2-like language for illustration. The focus of the presentation lies on the flexibility resulting from the genericity of the framework, which is demonstrated by different type analysis algorithms. Their results can be used for various optimizations like the elimination of unnecessary type tests and type assertions. Moreover, this article shows how a previously proposed approach for intraprocedural type analysis by Corney and Gough can be modeled in our approach, and even interprocedurally be enhanced.

1 Motivation

Type analysis of object-oriented programs is a necessary, but difficult task of program optimization in order to generate efficient code. In contrast to the imperative paradigm, where *abstract interpretation* (cf. [2, 8]) provides a uniform and theoretically well-founded platform for the construction of powerful analysis and optimization algorithms, for the object-oriented paradigm the situation is still characterized by approaches being quite specific or ad-hoc. Recently, Knoop and Schreiber have shown how to improve on this situation by transferring the abstract interpretation based approach for analysing and optimizing imperative programs to the object-oriented paradigm considering an Oberon-2-like language (cf. [6, 12]). Here, we put emphasis on the flexibility of this approach, which is demonstrated by two different *type analysis* algorithms focusing on *efficiency* and *precision*, respectively.

Related work. Type analysis for object-oriented programs has been an active field of research for years. The majority of approaches proposed (e.g. [10, 11]) focuses on untyped object-oriented languages, and does not (directly) apply to statically typed languages. In particular, the complex interplay of pointer variables and type-bound procedures in Oberon-2 lacks a direct counterpart in the settings considered there. Most closely related to our approach are the approaches of Corney and Gough [1], and Knoop and Golubski [4]. Corney and Gough proposed an intraprocedural type analysis algorithm for SimpleOberon; Knoop and Golubski demonstrated how to extend the abstract interpretation framework to untyped object-oriented languages.

```
Program     = { Decl;} { ProcDecl;} BEGIN  StatSeq  END.
Decl        = VAR  Ident:Type.
Type        = PreDefT | PT | RT.
PT          = POINTER TO  RT.
RT          = RECORD  [ ( Ident ) ]{ Ident : PreDefT;}.
StatSeq     = Stat {;Stat }.
Stat        = Ident := Expr | Ident ( { ActPar[;] } ) | WHILE  Expr  DO  StatSeq |
              IF  Expr  THEN  StatSeq  ELSE  StatSeq | ...
ProcDecl    = ProcHead ; ProcBody Ident ..
ProcHead    = PROCEDURE  [ Receiver ] Ident ( { FormalPar[;] } ).
ProcBody    = { Decl;} BEGIN  StatSeq  END.
FormalPar   = [VAR] Ident : Type.
```

Figure 1: Part of the SimpleOberon syntax.

2 The Language

We consider an Oberon-2-like programming language (cf. [9]), called *SimpleOberon*, which omits complex data structures like arrays or multilevel pointer structures, but is complex enough to demonstrate the essential features of the approach. The syntax of SimpleOberon is essentially given by the contextfree-like grammar of Figure 1. Central is the concept of Wirth's *type-extension* (cf. [14]), which allows a programmer to create a new record-type by adding fields to an existing record-type as illustrated in Figure 2. In the example, `rt0` is the *basis-type* of `rt1` and `rt2`, and {`rt0`, `rt1`, `rt2`, `rt3`} is the set of all *extension types* of `rt0`. Note that every type is an extension type of itself. Each record-type corresponds with a pointer-type. The *static type* of a pointer variable is the type of its declaration, its *dynamic type* is runtime dependent, and is an extension type of the static type. Classes in object-oriented languages do not only contain the data, but also the methods for manipulating them. In SimpleOberon, this is reflected in that procedures can be bound to record types. A procedure bound to a record-type is also accessible in all its extension types, unless it is overwritten. For example, `pv.P()` represents a call of the particular type-bound procedure `P`, which is bound to the dynamic type of the pointer variable `pv`.

Conventions. As usual we assume that inheritance is removed from the argument program, which can be achieved by expanding all record-type definitions by adding all inherited components except for type-bound procedures being overwritten. We introduce a special record-type $\mathtt{rt}_{NIL}$ corresponding to the common pointer-type $\mathtt{pt}_{NIL}$ (the type of the `NIL` object). Every record-type `rt` is an extension type of $\mathtt{rt}_{NIL}$, and every type-bound procedure `P` gets an equally-named "empty" implementation bound to $\mathtt{rt}_{NIL}$. Statements containing a type assertion are split into two parts: the type assertion and the term without the assertion. Finally, for type-bound procedures and type-bound procedure calls we add the receiver parameter as an ordinary parameter to the parameter list.

Goal of Type Analysis. The goal of type analysis for SimpleOberon programs is to compute for every occurrence of a pointer variable the set of dynamic types, which can actually occur at runtime as precisely as possible. Central for accomplishing this is the effect of elementary statements on the type of a variable. This is summarized in Figure 3.

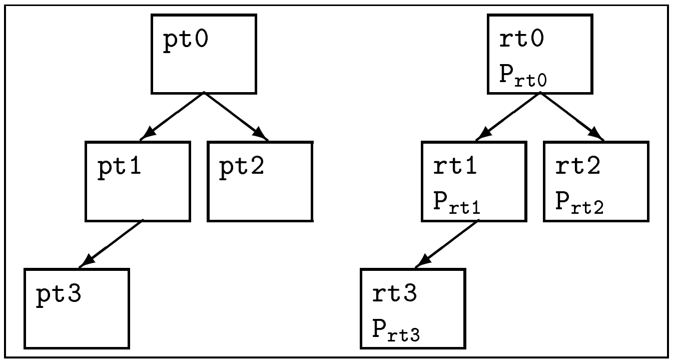

Figure 2: Type extension and type-bound procedures.

3 Framework of Abstract Interpretation

Abstract interpretation has proved to be a powerful and theoretically well-founded framework for static program analysis (cf. [2, 8]). In this article we use the framework developed by Knoop and Steffen (cf. [5, 7]), and its object-oriented extension by Knoop and Schreiber given in [6] and [12]. We represent SimpleOberon-programs by means of directed edge-labeled *flow graphs* $G_0, \dots, G_k$, where every G_i represents an ordinary or type-bound procedure (the main program is considered a parameterless procedure). The system $G = \{G_0, \dots, G_k\}$ is called a *program model.* The edges of G represent both the statements and the control flow of the underlying procedures, while the nodes represent just program points (cf. Figure 4). The function callee maps every edge representing a procedure call to the set of procedures it may invoke. For the object-oriented setting we have: if e represents an ordinary procedure call, callee(e) yields the singleton set containing this procedure. If it represents a type-bound procedure call, callee(e) yields the set of all procedures which are possibly called at runtime, i.e., the set of all equally-named procedures bound to some extension type of the static type of the pointer variable at the call site or to $\texttt{rt}_{NIL}$. Dually, the function caller yields the set of all call sites of a procedure G_i. A program model G does not explicitly represent the control flow caused by procedure calls. Therefore, we additionally consider the *interprocedural program model* G^* of G, which results from G by replacing every edge $e \in E_C$ by *call edges* leading from source(e) to the start node of every procedure of callee(e), and by *return edges* connecting the end nodes of these procedures with dest(e). These edges are labeled by assignments reflecting the parameter transfer (cf. [4, 6, 12]).

Abstract Semantics. The idea of abstract interpretation is to replace the “full” semantics of a program by a simpler, abstract version, which is tailored to deal with a specific problem. The (global) abstract semantics is typically induced by a *local abstract semantics* $[\![\ ]\!]' : E^* \to (C \to C)$, which gives abstract meaning to every edge of the interprocedural program model G^* in terms of a transformation on a complete lattice $(C, \sqcap, \sqsubseteq, \bot, \top)$. Its elements are assumed to represent the data flow information of interest. Fundamental for dealing with local variables of recursive procedures is the introduction of *stacks* of lattice elements and of *return functions* $R(e) : C \times C \to C$, $e \in E^*_R$, as in [7]. Intuitively, abstract stacks model the ordinary runtime stacks, the return functions the effect of returning from a procedure call, which requires to maintain the effects on global variables but to reset the effects on local ones.

The *global abstract semantics* of a program results from one of the following two globalization approaches: the "operational" *meet over all paths* (MOP) approach and the "denotational" *maximal fixed point* (MFP) approach. The MOP approach leads to the MOP solution, and globalizes a local abstract semantics by directly mimicing possible program executions: it "meets" (intersects) all information, which belong to a program path reaching the program point under consideration. The MOP solution does not specify an effective computation procedure in general. The MFP approach leads to the MFP solution in the sense of Kam and Ullman (cf. [3]). This approach induces an iterative computation procedure, which is effective, if the function lattice on C satisfies the descending chain condition, and if the local semantic functions $[\![\, e \,]\!]'$, $e \in E^*$, and the return functions $R(e)$, $e \in E^*_R$, are monotonic (cf. [5, 7]). In this setting the specification of a data flow analysis algorithm requires only four elementary components: the data domain, the local abstract semantics, the return functional, and the start information of interest.

Correctness and Coincidence. The following theorems give sufficient conditions for the correctness (or safety) and the coincidence (or precision) of the MFP solution with respect to the MOP solution. Along the lines of [7, 13] we get:

Theorem 3.1 (Correctness Theorem) *The MFP solution is a correct approximation of the MOP solution, i. e., $\forall\, c_0 \in C.\ \forall\, n \in N.\ \mathit{MFP}_{([\![\ ]\!]', c_0)}(n) \sqsubseteq \mathit{MOP}_{([\![\ ]\!]', c_0)}(n)$, if the functions $[\![\, e \,]\!]'$, $e \in E^*$, and $R(e)$, $e \in E^*_R$, are all monotonic.*

Theorem 3.2 (Coincidence Theorem) *The MFP solution and the MOP solution coincide, i. e., $\forall\, c_0 \in C.\ \forall\, n \in N.\ \mathit{MFP}_{([\![\ ]\!]', c_0)}(n) = \mathit{MOP}_{([\![\ ]\!]', c_0)}(n)$, if the functions $[\![\, e \,]\!]'$, $e \in E^*$, and $R(e)$, $e \in E^*_R$, are all distributive.*

4 Type Analysis

Resolving the Interplay of Pointer Variables and Type-bound Procedures
Central for type analysis of Oberon-2-like programs is to resolve the complex interplay of pointer variables and type-bound procedures: the dynamic types of pointer variables and the procedures called by type-bound procedure calls depend mutually on each other. Like in [6] and [12], we resolve these interdependencies by decomposing the analysis into two components dealing with (1) the computation of dynamic types of pointer variables, and (2) the computation of potentially called procedures. Both steps are repeated until a common fixed point is reached, i.e., both the sets delivered by (1) and (2) are invariant under further applications of the component analyses. Both components rely on information computed by the other. Fortunately, this (apparent) deadlock-situation can be resolved by means of the function callee. Based on the static type declarations of the program, it provides a safe approximation of the sets of potentially called procedures. This information is initially fed into the type analysis of step 1 returning an approximation of the sets of dynamic types of pointer variables according to this information. Vice versa, the type information on pointer variables induces now an improved approximation of the sets of potentially called procedures. The repetition stops, if the new approximation provided by step 2 coincides with the former one.

Statement	Semantics	Effect on the dynamic type of the pointer variable pv
NEW(pv)	**Create:** Creates a new variable of the record-type corresponding to the static pointer-type of pv.	The dynamic type of pv is equal to its static type.
pv:=pv′	**Assign:** (i) Assigns pointer variable pv′ to pv.	The dynamic type of pv is set to the dynamic type of pv′.
pv:=NIL	(ii) Assigns the special value NIL to pointer variable pv.	The dynamic type of pv is set to $\mathtt{pt}_{NIL}$.
pv(pt)	**Test:** (i) Type assertion (type guard). Tests the dynamic type of pv.	No effect on the dynamic type of pv, if it is equal to some extension type of pt. Otherwise the program execution is aborted.
pv IS pt	(ii) Type test. Evaluates to true, if the dynamic type of pv is some extension type of pt, otherwise it evaluates to false.	No effect on the dynamic type of pv.

Figure 3: Semantics of elementary statements.

I. The First Type Analysis: Emphasizing Efficiency

First we consider the type-extension (inheritance) tree as the lattice of interest as proposed by Corney and Gough for an intraprocedural setting (cf. [1]). We adapt the local semantic functions accordingly in order to model their algorithm in our approach and enhance it interprocedurally. The definition of the local abstract semantics and the return functions requires a safe approximation of the procedures potentially called by type-bound procedure calls. This information is made available by the function callee_{ta} provided by the component of step 2. Initially, callee_{ta} is given by callee, which is based on static, and hence, safe type information. In the specification below, the index "ta" being a shorthand for "type analysis" reminds of this fact. As noted in Section 3 we specify a data flow algorithm by 4 components.

(1) Specification of the First Type Analysis. Let $PT^{\top}$ be the set of all pointer types including the special type $\top_{ta}$ with $\forall \mathtt{pt} \in PT.\ \top_{ta} \mid\!>^* \mathtt{pt} \mid\!>^* \mathtt{pt}_{NIL}$, where $\mid\!>$ denotes the extension type relation and $\mid\!>^*$ its reflexive, transitive closure (cf. [14]). The function $\oplus$ maps a set of pointer-types to their "mostly extended" common basis-type.

Data domain. Let $(C_i, \sqcap, \sqsubseteq, \bot, \top) = (PT^{\top}, \oplus, <\!\mid, \mathtt{pt}_{NIL}, \top_{ta})$. As C_i is a complete lattice this also holds for $(C_1 \times C_2 \times \ldots \times C_n, \sqsubseteq)$, which in the following is abbreviated by $(C, \sqcap, \sqsubseteq, \bot, \top)$. The projection function $\downarrow_{\mathtt{pv}}$ maps every lattice element $c \in C$ to the component belonging to the pointer-variable pv. The possible types of pv wrt. $c \in C$ are given by $\{\mathtt{pt} \mid \mathtt{pt} \mid\!>^* c\downarrow_{pv}\}$.

Local abstract semantics. The local abstract semantics is given by: $\forall e \in E^*_{ta}.\forall \mathtt{pv} \in PVar.$

$$[\![\, e \,]\!](c) = \begin{cases} c[\mathsf{statTyp}(\mathtt{pv})\backslash \mathtt{pv}] & \text{if } e \equiv \mathtt{NEW}(\mathtt{pv}) \\ c[c\downarrow_{\mathtt{pv}'} \backslash \mathtt{pv}] & \text{if } e \equiv \mathtt{pv} := \mathtt{pv}' \\ c[\mathtt{pt}\backslash \mathtt{pv}] & \text{if } e \equiv \mathtt{pv}(\mathtt{pt}) \text{ and } c\downarrow_{\mathtt{pv}} \notin \mathsf{extTyp}(\mathtt{pt}) \\ c[\mathtt{pt}_{NIL}\backslash \mathtt{pv}] & \text{if } e \equiv \mathtt{pv} := \mathtt{NIL} \\ c & \text{otherwise} \end{cases}$$

Here, $\mathsf{statTyp(pv)}$ denotes the static type of $\mathtt{pv}$, $\mathsf{extTyp(pt)}$ the set of extension types of $\mathtt{pt}$, and $c[\,.\,\backslash\,.\,]$ a substitution defined by: Let $\mathtt{pt} \in PT^{\top}, \mathtt{pv} \in PVar, c \in C$:

$$\forall \mathtt{pv}' \in PVar.\ (c[\mathtt{pt}\backslash\mathtt{pv}])\downarrow_{\mathtt{pv}'} = \begin{cases} \mathtt{pt} & \text{if } \mathtt{pv} = \mathtt{pv}' \\ c\downarrow_{\mathtt{pv}'} & \text{otherwise} \end{cases}$$

Return functions. The return functions are defined by: $\forall e \in E^*_{Rta}.\ R(e)(c_1, c_2) = c_3$ with

$$\forall \mathtt{pv} \in PVar.\ c_3\downarrow_{\mathtt{pv}} = \begin{cases} c_1\downarrow_{\mathtt{pv}} & \text{if } \mathtt{pv} \in \mathsf{LocVar}(\mathsf{pg}(\mathsf{source}(e))) \\ c_2\downarrow_{\mathtt{pv}} & \text{otherwise} \end{cases}$$

LocVar maps a procedure to its set of local pointer variables and value parameters, pg maps a node to the flow graph G_i containing it, and source maps an edge to its source node.
Start information. It is given by $c_0 \in C$ with $\forall \mathtt{pv} \in PVar.\ c_0\downarrow_{\mathtt{pv}} = \mathtt{pt}_{NIL}$.

Actually, the definition of the type analysis is rather straightforward. Thus, we only discuss the case dealing with type assertions `pv(pt)` in the definition of the local semantic functional in more detail. Note, in case of $c\downarrow_{\mathtt{pv}} \in \mathsf{extTyp(pt)}$, the type assertion holds, and consequently the local abstract semantics has no effect. On the other hand, if $c\downarrow_{\mathtt{pv}} \in \mathsf{extTyp(pt)}$ is violated, the test on the assertion fails indicating a possible program abortion at runtime. However, instead of immediately finishing the analysis with indicating this failure, we proceed by propagating the type information $\mathtt{pt}$ provided by the assertion. Thus, the analysis is continued using the information originally intended by the programmer, which allows us to provide him a program being completely annotated with type information, and not just partially up to detecting the first point of a possible failure. This is important in practice. Moreover, the information on possible failures is retained by this proceeding. After termination, the existence of possible failures can simply be checked by comparing the type assertions with the set of dynamic types computed for the source node of the edge the type assertion under consideration is attached to.

Correctness and Termination of the First Type Analysis. The first part of Theorem 4.1 follows immediately from the finiteness of $PVar$ and $PT^{\top}$. The second part can easily be proved by means of the definitions of the local semantic functions and return functions.

Theorem 4.1 (Descending Chain Condition and Distributivity)
1. The lattice C satisfies the descending chain condition.
2. All local semantic functions and return functions of the type analysis are distributive.

Theorem 4.1 directly yields the effectivity and termination of the instantiated fixed point algorithm (cf. [12]). Moreover, because of part 2, the Coincidence Theorem 3.2 is applicable yielding the precision of the results of the type analysis with respect to the MOP solution.

Theorem 4.2 (Coincidence) *The MFP solution of the first type analysis and its MOP solution coincide.*

(2) Computing Potentially Called Procedures of Type-bound Procedure Calls

By means of the type information computed by step 1, we get a new approximation of the sets of procedures potentially called by type-bound procedure calls. It is given by:

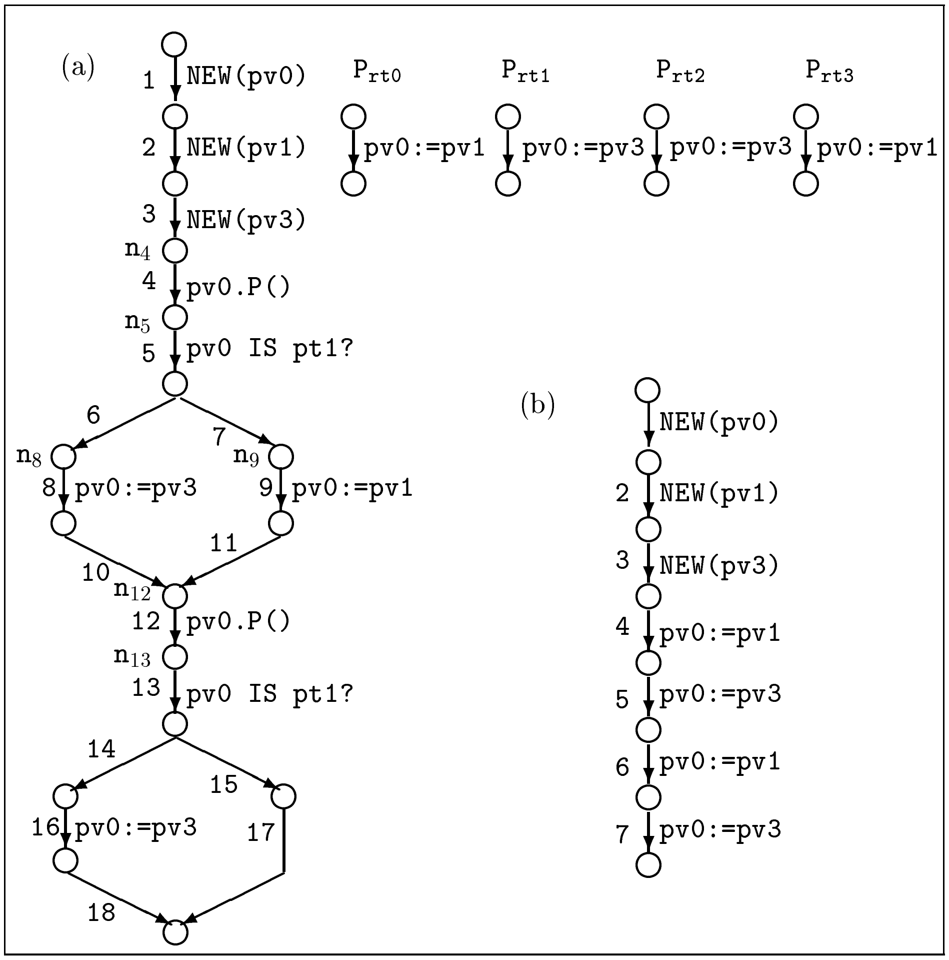

Figure 4: Illustrating example.

$$\forall e \in E_{TPC}.\ \mathsf{pp}_{ta}(e) = \mathsf{pwts}(\mathsf{tpc\text{-}tp}(e), \mathsf{dynTyp}_{ta}(\mathsf{tpc\text{-}id}(e), \mathsf{source}(e)))$$

where E_{TPC} is the set of edges labeled with a type-bound procedure call pv.P(), tpc-id and tpc-tp are two functions mapping an edge $e \in E_{TPC}$ to pv, and the set of procedures with identifier P bound to some extension type of the static type of pv, respectively. $\mathsf{pwts}(\{G_1, \ldots, G_m\}, \{\mathtt{pt}_1, \ldots, \mathtt{pt}_l\})$, finally, denotes the set of type-bound procedures of $G_1, \ldots, G_m$ being bound to some type of $\mathtt{pt}_1, \ldots, \mathtt{pt}_l$. Intuitively, pp_{ta} induces a "dynamic" counterpart of callee being more precise as it profits from type analysis information, whereas callee (cf. Section 3) is based on static type information only. The function callee_{ta} can now be (re-)fed into step 1 offering the type analysis the chance of computing an improved approximation of the sets of dynamic types of pointer variables. After the termination of the global analysis, i.e., after reaching a fixed point for both steps, the sets of procedures possibly called by type-bound procedure calls are given by:

$$\forall e \in E_{TPC}.\ \mathsf{potProc}(e) = \mathsf{callee}_{ta_{fix}}(e)$$

where $\mathsf{callee}_{ta_{fix}}$ denotes the final value of callee_{ta}.

Correctness. In essence, the correctness of the complete two-step approach is a consequence of the validity of the inclusion $\forall e \in E_{TPC}.\ \mathsf{actProc}(e) \subseteq \mathsf{callee}(e)$ where actProc denotes the sets of type-bound procedures which can *actually* be called at runtime.

Node	First ("efficient") solution	Second ("precise") solution
n_4	{pt0, pt1, pt2, pt3}	pt0
n_5	{pt1, pt3}	pt1
n_8	{pt1, pt3}	pt1
n_9	{pt1, pt3}	$\emptyset$
n_{12}	{pt1, pt3}	pt3
n_{13}	{pt1, pt3}	pt1

Figure 5: Type analysis results: the possible types of pointer variable pv0.

Illustrating Example: First Type Analysis

We demonstrate the type analysis algorithm by means of the example of Figure 4(a), which is assumed to obey the type hierarchy of Figure 2. We assume that pv$i$ is a pointer variable of the type POINTER TO pt$i$, $\mathtt{P}_{\mathtt{rt}i}$ are equally-named type-bound procedures corresponding to the record types rt$i$, and n_i is an abbreviation for the node $\mathsf{source}(e_i), e_i \in E$.

The global analysis starts with fixing the initial approximation of the sets of procedures possibly called by type-bound procedure calls. Initially, it returns for both type-bound procedure calls pv0.P() at the edges 4 and 12 the procedures $\mathtt{P}_{\mathtt{rt0}}$, $\mathtt{P}_{\mathtt{rt1}}$, $\mathtt{P}_{\mathtt{rt2}}$ and $\mathtt{P}_{\mathtt{rt3}}$. This information fixes the program model for the first application of step 1 of the global analysis computing a first approximation of the set of potential types of pointer variables. For the occurrences of variable pv0 at nodes n_4, n_5 and n_{12} it returns the type sets {pt0, pt1, pt2, pt3}, {pt1, pt3} and {pt1, pt3}, respectively. Repeating step 2, we get $\mathtt{P}_{\mathtt{rt0}}$, $\mathtt{P}_{\mathtt{rt1}}$, $\mathtt{P}_{\mathtt{rt2}}$ and $\mathtt{P}_{\mathtt{rt3}}$ as possibly called procedures for the call site at edge 4, $\mathtt{P}_{\mathtt{rt1}}$ and $\mathtt{P}_{\mathtt{rt3}}$ at edge 12. This information fixes a new program model for the second application of step 1. We obtain {pt0, pt1, pt2, pt3} as possible types of the variable pv0 at node n_4, {pt1, pt3} at node n_5, and {pt1, pt3} at node n_{12}. After updating the sets of possibly called procedures, which yields that every procedure $\mathtt{P}_{\mathtt{rt0}}$ to $\mathtt{P}_{\mathtt{rt3}}$ can be called at edge 4 and $\mathtt{P}_{\mathtt{rt1}}$ or $\mathtt{P}_{\mathtt{rt3}}$ at edge 12, the fixed point is reached, and the global analysis stops. Figure 5 summarizes the possible dynamic types of pv0 at several nodes after the termination of the whole analysis process.

II. The Second Type Analysis: Emphasizing Precision

Different type analyses can easily be obtained by only modifying the first step of the global analysis procedure. In fact, changing or modifying the local abstract semantics suffices for obtaining new type analyses or variants of a type analysis in order to meet the individual requirements of a user concerning precision and efficiency. In the previous solution, two effects limit the precision: the inaccuracy resulting from the use of the "mostly extended" common basic type at "meet" points and the nondeterministic treatment of branches. In the following we present a second analysis addressing both problems: it (1) uses a "finer" lattice and (2) considers branches deterministic in the fashion of [4] by means of introducing *filter* functions in order to evaluate type tests.

Specification of the Second Type Analysis.

Data domain. The data domain of our second type analysis is given by the powerset lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top) = (\mathcal{P}(\mathcal{F}), \cup, \supseteq, \mathcal{F}, \emptyset)$ where $F = [PVar \to PT]$ denotes the set of all functions from pointer variables to pointer types including $\mathtt{pt}_{NIL}$.

Local abstract semantics. It is given by $[\![\ ]\!] : E^*_{ta} \to (\mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F}))$, which is defined by $\forall e \in E^*_{ta}. \forall P \in \mathcal{P}(\mathcal{F}).\ [\![\, e \,]\!](P) = \{[\![\, e \,]\!]^\circ(f) \mid f \in P\}$ where $[\![\ ]\!]^\circ : E^*_{ta} \to (F \to F)$ is given by:

$$\forall \mathsf{pv} \in PVar.\ [\![\, e \,]\!]^{\circ}(f) = \begin{cases} f[\mathsf{statTyp}(\mathtt{pv})\backslash\mathtt{pv}] & \text{if } e \equiv \mathtt{NEW(pv)} \\ f[f(\mathtt{pv}')\backslash\mathtt{pv}] & \text{if } e \equiv \mathtt{pv} := \mathtt{pv}' \\ f[\mathtt{pt}\backslash\mathtt{pv}] & \text{if } e \equiv \mathtt{pv(pt)} \text{ and } f(\mathtt{pv}) \notin \mathsf{extTyp}(\mathtt{pt}) \\ f[\mathtt{pt}_{NIL}\backslash\mathtt{pv}] & \text{if } e \equiv \mathtt{pv} := \mathtt{NIL} \\ f & \text{otherwise} \end{cases}$$

$f[\,.\,\backslash\,.\,]$ is a substitution defined as follows: If $f \in F, \mathtt{pv} \in PVar, \mathtt{pt} \in PT$, then $f[\mathtt{pt}\backslash\mathtt{pv}]$ is the unique function of F defined by:

$$\forall \mathtt{pv}' \in PVar.\ f[\mathtt{pt}\backslash\mathtt{pv}](\mathtt{pv}') = \begin{cases} \mathtt{pt} & \text{if } \mathtt{pv} = \mathtt{pv}' \\ f(\mathtt{pv}') & \text{otherwise} \end{cases}$$

Return functions. The return functional $R : E^*_{R_{ta}} \to (\mathcal{P}(\mathcal{F}) \times \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F}))$ is defined by: $\forall e \in E^*_{R_{ta}}. \forall P_1, P_2 \in \mathcal{P}(\mathcal{F})$.

$$R(e)(P_1, P_2) = \{R'(f_1, f_2) \mid f_2 = [\![\, e \,]\!] \circ [\![\, \mathsf{source}(e) \,]\!] \circ [\![\, e_C \,]\!](f_1), f_1 \in P_1, f_2 \in P_2\}$$

where e_C denotes the call edge corresponding to $e \in E^*_{R_{ta}}$. The function $R' : (F \times F) \to F$, finally, is defined by: $\forall\, (f_1, f_2) \in F \times F.\ R'(f_1, f_2) = f_3$ where

$$\forall \mathtt{pv} \in PVar.\ f_3(\mathtt{pv}) = \begin{cases} f_1(\mathtt{pv}) & \text{if } \mathtt{pv} \in \mathsf{LocVar}(\mathsf{pg}(\mathsf{source}(e))) \\ f_2(\mathtt{pv}) & \text{otherwise} \end{cases}$$

Start information. The start information is given by the singleton set $F_0 = \{f_0\} \in \mathcal{P}(\mathcal{F})$, where the function f_0 is defined by $\forall \mathtt{pv} \in PVar.\ f_0(\mathtt{pv}) = \mathtt{pt}_{NIL}$.

We introduce filter functions, which relies on the following simple program transformation: remove every type test `pv IS pt` and insert `pv IS pt` on the true and $\neg$ (`pv IS pt`) on the false branch. In essence, introducing filter functions amounts to expanding the local semantic functional to edges labeled by branch information. For the presented type analysis this is accomplished as follows:

$$\forall e \in E. \forall P \in \mathcal{P}(\mathcal{F}).[\![\, e \,]\!](\mathcal{P}) = \begin{cases} \{f \mid f(\mathtt{pv}) \in \mathsf{extTyp}(\mathtt{pt}), f \in P\} & \text{if } e \equiv \mathtt{pv\ IS\ pt} \\ \{f \mid f(\mathtt{pv}) \notin \mathsf{extTyp}(\mathtt{pt}), f \in P\} & \text{if } e \equiv \neg(\mathtt{pv\ IS\ pt}) \\ \{[\![\, e \,]\!]^{\circ}(f) \mid f \in P\} & \text{otherwise} \end{cases}$$

Note that the local abstract semantics of a type test performs in fact like a filter: functions $f \in F$ are propagated along the true-branch, if the test `pv IS pt` evaluates to true, i.e., if the dynamic type of `pv` is `pt` or an extension type of `pt`. Otherwise the function propagation of the filter is blocked. This holds dually for the false-branch.

Correctness and Termination of the Second Type Analysis. We can prove:

Theorem 4.3 (Descending Chain Condition and Monotonicity)
1. *The lattice $\mathcal{P}(\mathcal{F})$ satisfies the descending chain condition.*
2. *All local semantic functions and return functions of the type analysis are monotonic.*

Theorem 4.4 (Correctness) *The MFP solution of the second type analysis is a safe approximation of its MOP solution.*

Illustrating Example: Second Type Analysis

The usage of filter functions requires to replace the labels of edge 5 and edge 13 by `skip`, and to attach the labels `pv0 IS pt1` to the edges 6 and 14 and ¬ (`pv0 IS pt1`) to the edges 7 and 15 assuming that edges 6 and 14 represent the corresponding true-branches and edges 7 and 15 the false-branches.

Let us now consider the proceeding for the second type analysis. Initially, at both type-bound procedure calls the set of procedures $\{\mathtt{P_{rt0}}, \mathtt{P_{rt1}}, \mathtt{P_{rt2}}, \mathtt{P_{rt3}}\}$ is used as initial approximation of potentially called procedures. Step 1 of the global analysis delivers for the variable `pv0` at node n_4 the type `pt0`, and at both nodes n_5 and n_{12} the types `pt1` and `pt3`. This gives a new approximation for the called procedures. After step 2, we get $\mathtt{P_{rt0}}$ as called procedure for edge 4 and $\mathtt{P_{rt1}}$, $\mathtt{P_{rt3}}$ for edge 12. Repeating step 1 we obtain `pt0` as possible type of the variable `pv0` at node n_4, `pt1` at node n_5 and `pt3` at node n_{12}. In contrast to the first solution, we need a third repetition of the process yielding that $\mathtt{P_{rt0}}$ is callable at edge 4 and $\mathtt{P_{rt3}}$ at edge 12. After computing the dynamic types of the pointer variables and updating the sets of possible called procedures the fixed point is reached.

As in this example the second type analysis gives in general more precise results as the first one (see Figure 5). On the other hand, it is less efficient, which is essentially a consequence of the larger maximal chain length of the data flow lattice determining above all the worst case complexity of the type analysis algorithm.

Optimizations

Type information as computed in the previous section can immediately be used for a variety of optimizations. Particularly straightforward and important are (1) the elimination of unnecessary type tests and type assertions, (2) the replacement of dynamically bound procedure calls by statically bound procedure calls, and (3) the inlining of procedures. This is illustrated in Figure 4(b). It shows the program of Figure 4(a) after eliminating unnecessary type tests (the type of `pv0` at edge 5 and edge 13 is always `pt1`), and inlining of $\mathtt{P_{rt0}}$ at edge 4 and $\mathtt{P_{rt3}}$ at edge 12. As a side-effect this enables a classical optimization, the elimination of *dead code*. In this example, the edges 4, 5 and 6 represent useless code. In fact, it is worth noting that type information is the key for transfering classical analysis and optimization procedures of imperative languages to the object-oriented setting.

5 Conclusion

Precise type information is the backbone of generating efficient code for object-oriented programs. In this article we demonstrated the flexibility of a generic abstract interpretation based approach for type analysis of strongly typed Oberon-2-like object-oriented languages. Fundamental was to resolve the complex interdependencies between pointer variables and type-bound procedures by decomposing the analysis into two steps mutually fed with the result of their counterpart. The genericity of the framework supports the construction of type analyses of different precision and efficiency, supporting thus the construction of user-customized solutions, as demonstrated by two type analyses focusing on efficiency and precision, respectively.

References

[1] D. Corney and J. Gough. Type test elimination using typeflow analysis. In *Proceedings of the 1st International Conference on Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1994.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Symposium on Principles of Programming Languages*, pages 238–251. ACM, 1977.

[3] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[4] J. Knoop and W. Golubski. Abstract interpretation: A uniform framework for type analysis and classical optimization of object-oriented programs. In *Proc. of the 1st International Symposium on OO Technology "The White OO Nights"* (*WOON'96*), pages 126–142, 1996.

[5] J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Lang.*, 4(4):211–246, 1996.

[6] J. Knoop and F. Schreiber. Analysing and Optimizing Strongly Typed Object-oriented Languages: A Generic Approach and its Application to Oberon-2. In *Proc. of the 2nd International Symposium on OO Technology "The White OO Nights"* (*WOON'97*), pages 252–266, 1997.

[7] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 1992.

[8] K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103 – 129, 1993.

[9] H. Mössenböck. *Object-Oriented Programming in Oberon-2.* Springer Verlag, 2nd ed., 1995.

[10] N. Oxhøj, J. Palsberg, and M.-I. Schwartzbach. Making type inference practical. In *Proceedings of the 6th European Conference on Object-oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349. Springer-Verlag, 1992.

[11] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 9th ACM SIGPLAN Annual Conference on OO Programming Systems, Languages and Applications*, pages 324–340, 1994.

[12] F. Schreiber. Datenflußanalyse und Optimierung objektorientierter Programme. Master's thesis, Universität Passau, 1997.

[13] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 – 233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[14] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10:204–214, 1988.