ELSEVIER Science of Computer Programming 40 (2001) 189–211

Science of Computer Programming

www.elsevier.nl/locate/scico

An empirical study into COBOL type inferencing ☆

Arie van Deursen [1], Leon Moonen [1] *

CWI, P.O. Box 94079, 1090 GB Amsterdam, Netherlands

Accepted 2 February 2001

Abstract

In a typical COBOL program, the data division consists of 50% of the lines of code. Automatic type inference can help to understand the large collections of variable declarations contained therein, showing how variables are related based on their actual usage. The most problematic aspect of type inference is *pollution*, the phenomenon that types become too large, and contain variables that intuitively should not belong to the same type. The aim of the paper is to provide empirical evidence for the hypothesis that the use of *subtyping* is an effective way for dealing with pollution. The main results include a tool set to carry out type inference experiments, a suite of metrics characterizing type inference outcomes, and the experimental observation that only one instance of pollution occurs in the case study conducted. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Software maintenance; Static program analysis; Variable usage; Case study

1. Introduction

In this paper, we will be concerned with the variables occurring in a COBOL program. The two main parts of a COBOL program are the *data division*, containing declarations for all variables used, and the *procedure division*, which contains the statements performing the program's functionality. Since it is in the procedure division that the actual computations are made, one would expect this division to be *larger* than the data division. Surprisingly, we found that in a typical COBOL system this is not the case: the data division often comprises more than 50% of the lines of code. We even encountered several programs in which 90% of the lines of code were part of the data

☆ Revised version of the paper Understanding COBOL systems using inferred types, in: S. Woods (Ed.), Proc. 7th Internat. Workshop on Program Comprehension, IEEE Computer Society Press, Silver Spring, MD, May 1999.

* Corresponding author.

E-mail address: leon.moonen@cwi.nl (L. Moonen).

[1] http://www.cwi.nl/~{arie,leon}

0167-6423/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0167-6423(01)00015-6

division.[2] (As we will see in this paper, one reason for this is that COBOL does not distinguish between type and variable declarations.)

These figures have two implications. First of all, they suggest that only a subset of all declared variables are actually used in a COBOL program. If 90% of the lines are variable declarations, it is unlikely that the remaining 10% will use all these variables. Indeed, in the systems we studied, we have observed that less than 50% of the variables declared are used in the procedure division.[3]

These figures also indicate that maintenance programmers need help when trying to understand the data division part. Just reading the data division will involve browsing through a lot of irrelevant information. Thus, the minimal help is to see which variables are in fact used, and which ones are not. In addition to that, the maintenance programmer will want to understand the relationships that hold between variables. In COBOL, some of these relations can be derived from the data division, such as whether a variable is part of a larger record, whether it is a redefine (alias) of another variable, or whether it is a predicate on another variable (level 88).

But not all relevant relations between variables are available in the data division. When do two different variables hold values that represent the same business entity? Can a given variable ever receive a value from some other given variable? What values are permitted for this variable? Is the value of this variable ever written to file? Is the value of this variable passed as output to some other program? What values are actually used for a given variable? What are the operations permitted on a given variable?

In strongly typed languages, questions like these can be answered by inspecting the *types* that are used in a program. First, a type helps to understand what set of values is permitted for a variable. Second, types help to see when variables represent the same kind of entities. Third, they help to hide the actual representation used (array versus record, length of array, etc.), allowing a more abstract view of the variable. Last but not least, types for input and output parameters of procedures immediately provide a "signature" of the intended use of the procedure.

Unfortunately, the variable declarations in a COBOL data division suffer from a number of problems that make them unsuitable to fulfill the roles of types as discussed above. In COBOL, it is not possible to separate type definitions from variable declarations. This has three unpleasant consequences. First, when two variables need the same record structure, this structure is *repeated*. Second, whenever a data division contains a repeated record structure, the lack of type definitions makes it difficult to determine whether that repetition is accidental (the two variables are not related), or whether it is intentional (the two variables should represent the same sort of entity). Third, the absence of explicit types leads to a lack of abstraction, since there is no way to hide the actual representation of a variable into some type name.

[2] For three different systems, each approx. 100,000 LOC, we found averages of 53%, 43%, and 58%, respectively.

[3] For the Mortgage system under study in this paper, on average 58% of the variables declared in a program were never used, the percentages ranging from 2.6% for the smallest up to 95% for the largest program.

In short, the problem we face with COBOL programs is that types are needed to understand the myriads of different variables, but that the COBOL language does *not* support the notion of types.

In [5], we have proposed a solution to this problem. Instead of deriving type information from the data division, we perform a *static analysis* on the programs to *infer* types from the usage of variables in the procedure division. The basic idea of type inference is simple: if the value of a variable is assigned or compared to another variable, we want to infer that these two variables should have the same type. However, just inferring a type *equivalence* for every assignment will not do. As an example, a temporary string value could receive values from names, streets, cities, etc., which should all have different types. Via transitivity of equivalence, however, all variables assigned to that string variable would receive the same type. This phenomenon, that a type equivalence class becomes too large, and contains variables that intuitively should not belong to the same type, is called *pollution*. In order to avoid pollution, we have proposed to introduce *subtyping* for assignments rather than type *equivalence* [5].

In this paper, we will carefully study the problem of pollution, and test the hypothesis that it is handled by deriving subtypes rather than equivalences. This is done by presenting statistical data illustrating the presence of pollution, and the effectiveness of subtyping for dealing with it. In particular, we look at the interplay between subtyping and equivalence (for example, consider two types T_A and T_B, and $T_A \preccurlyeq T_B$, and $T_B \preccurlyeq T_A$, we get $T_A \equiv T_B$ —how does this affect pollution?).

Moreover, we will discuss how *relational algebra* can be used for implementing COBOL type inferencing. Relational algebra has recently been proposed as a valuable tool for various reverse engineering and program understanding activities, such as architecture recovery [8,12]. It is based on Tarski's relational operators [23], such as union, subtraction, relational composition, etc. The use of relational algebra helps us to completely separate COBOL-specific source code analysis from calculating with types. Moreover, it enables us to specify type relationships at an appropriate level of abstraction.

All experiments are done on Mortgage, a real-life COBOL/CICS system from the banking environment. This system consists of 100,000 lines of code; with all copybooks (include files) expanded (unfolded), it consists of 250,000 lines of code. It conforms to the COBOL-85 standard, which is the most widely used COBOL version. Compared to a COBOL code base of 3 million lines we have available, Mortgage contains fairly representative COBOL code (it is neither the worst nor the best code).

2. Type inference

In this section, we summarize the essentials of COBOL type inferencing: a more complete presentation is given in [5]. We start by describing the *primitive types* that we distinguish. Then, we describe how *type relations* can be derived from the statements in a single COBOL program, and how this approach can be extended to *system-level*

analysis leading to inter-program dependencies. Finally, we show how the analysis can be extended to include types for *literals*, discuss the notion of *pollution*, and conclude with an example.

2.1. Primitive types

We distinguish three primitive types: (1) elementary types such as numeric values or strings; (2) arrays; and (3) records. Initially, every declared variable gets a unique primitive type. Since variable names qualified with their complete record name must be unique in a COBOL program, these names can be used as labels within a type to ensure uniqueness. Furthermore, we qualify variable names with program or copybook names to obtain uniqueness at the system level. We use T_A to denote the primitive type of variable A.

2.2. Type equivalence

By looking at the *expressions* occurring in statements, an *equivalence relation* between primitive types can be inferred. We distinguish three cases:

(1) *Relational expressions*: From a relational expression such as $v = u$ or $v \leqslant u$ an equivalence between T_v and T_u is inferred.
(2) *Arithmetic expressions*: From a arithmetic expression such as $v + u$ or $v * u$ an equivalence between T_v and T_u is inferred.
(3) *Array accesses*: From two different accesses to the same array, such as $a[v]$ and $a[u]$ an equivalence between T_v and T_u is inferred.

When we speak of a *type* we will generally mean an *equivalence class of primitive types*. For presentation purposes, we may also give names to types based on the names of the variables part of the type. For example, the type of a variable with the name `L100-DESCRIPTION` will be called `DESCRIPTION`-type.

2.3. Subtyping

By looking at the *assignment statements*, we infer a *subtype relation* between primitive types. Note that the notion of assignment statements corresponds to COBOL statements such as `MOVE`, `COMPUTE`, `MULTIPLY`, etc. From an assignment of the form $v := u$ we infer that T_u is a *subtype* of T_v, i.e., v can hold at least all the values u can hold.

2.4. Union types

From a COBOL *redefine clause*, we infer a *union type* relation between primitive types. When a given entry v in the data division redefines another entry u, we infer that T_v and T_u are part of the same *union type*.

2.5. System-level analysis

In addition to inferring type relations within individual programs, we derive type relations at the system-wide level. We infer that the types of the actual parameters of a program call (listed in the COBOL `USING` clause) are subtypes of the formal parameters (listed in the COBOL `LINKAGE` section), and that variables read from or written to the same file or table have equivalent types. Furthermore, we want to ensure that if a variable is declared in a copybook, its type is the same in all the different programs that copybook is included in. In order to do this, we derive relations that denote the origins of primitive types and the import relation between programs and copybooks. These relations are then used to link types via copybooks.

2.6. Literals

A natural extension of our type inference algorithm involves the analysis of literals that occur in a COBOL program. Whenever a literal value l is assigned to a variable v, we conclude that the value l must be a permitted value for the type of v. Likewise, when v and l are compared, l is considered a permitted value for the type of v. Literal analysis indicates permitted values for a type. Moreover, if additional analysis indicates that variables in this type are only assigned values from this set of literals, we can infer that the type in question is an *enumeration type*.

2.7. Pollution

The intuition behind type equivalence is that if the programmer would have used a typed language, he or she would have chosen to give a single type to two different COBOL variables whose types are inferred to be equivalent. We speak of *type pollution* if an equivalence is inferred which is in conflict with this intuition.

Typical situations in which pollution occurs include the use of a single variable for different purposes in different program slices; the use of a global variable acting as a formal parameter, to which a range of different variables can be assigned; and the use of a `PRINT-LINE` string variable for collecting output from various variables.

2.8. Example

Fig. 1 contains a COBOL fragment illustrating various aspects of type inferencing. The first half contains the declarations of variables, containing their physical types, i.e., how many bytes they occupy. The second half contains the actual statements from which type relations between variables are inferred.

Going from bottom to top, we first see (line 41) that variable `A00-FILLED` is compared to `N100`, from which we infer that they belong to the same type. From line 39, we then infer an additional type equivalence, adding `A00-MAX` to this equivalence class. We thus obtain one type, for three different variables. If we also take

```
1   / variables containing business data.
2    01    PERSON-RECORD.
3       03 INITIALS         PIC  X(05).
4       03 NAME             PIC  X(27).
5       03 STREET           PIC  X(18).
6       ...
7
8   / variables containing char array of length 40,
9   / as well as several counters.
10   01    TAB000.
11      03 A00-NAME-PART.
12         05 A00-POS        PIC  X(01)  OCCURS 40.
13      03 A00-MAX           PIC S9(03)  COMP-3 VALUE 40.
14      03 A00-FILLED        PIC S9(03)  COMP-3 VALUE ZERO.
15
16   ...
17
18  / other counters declared elsewhere.
19   01 N000.
20      03  N100             PIC S9(03)  COMP-3 VALUE ZERO.
21      03  N200             PIC S9(03)  COMP-3 VALUE ZERO.
22   ...
23
24  / procedure dealing with initials.
25   R210-VOORLT SECTION.
26      MOVE INITIALS TO A00-NAME-PART.
27      PERFORM R300-COMPOSE-NAME.
28
29  / procedure dealing with last names.
30   R230-NAME SECTION.
31      MOVE NAME TO A00-NAME-PART.
32      PERFORM R300-COMPOSE-NAME.
33
34  / procedure for computing a result based on the
35  / value of the A00-NAME-PART.
36  / Uses A00-FILLED, A00-MAX, and N100 for array indexing.
37   R300-COMPOSE-NAME SECTION.
38      ...
39      PERFORM UNTIL N100 > A00-MAX
40         ...
41         IF A00-FILLED = N100
42           ...
```

Fig. 1. Excerpt from a real-life COBOL program.

a look at the data division, we see that this equivalence is in accordance with their declared picture layouts (in lines 13, 14, and 20), which are all numeric data elements. However, we cannot infer such equivalences from just the pictures, as entirely unrelated data structures may share the same physical layout (for example, N200 in line 21).

As assignment example is given in line 31, where NAME is assigned to NAME-PART. Here we infer that the type of NAME is a *subtype* of NAME-PART. In line 26, another variable, INITIALS, is assigned to NAME-PART as well, giving rise to a second subtype relationship, now between INITIALS and NAME-PART. In this way, INITIALS and NAME share a common supertype (NAME-PART), but there is no direct relationship inferred between them. If we look at the declared physical layout we see that all three are strings of a different length (in lines 3,4 and 12). NAME-PART is the largest, capable of accepting values from both INITIALS and NAME.

In fact, NAME-PART is a global variable acting as a *formal parameter* for the procedure R300-COMPOSE-NAME (COBOL does not support the declaration of parameters for procedures). What we infer is that the type of the actual parameter is a subtype of the formal parameter. Just deriving equivalences from assignments would lead to *pollution*: it would give all the actual parameters, in this case the two different concepts "initials" and "first name", the same type.

2.9. Practical value

COBOL type inferencing provides a theory for grouping variables based on their usage. This is of great practical value for the understanding and (semi-automated) transformation of COBOL legacy systems. Example application areas are discussed in [5], and include the introduction of symbolic names for literal values (per type), extraction of system interfaces based on parameter types, migration to strongly typed languages such as Pascal, identification of candidate classes in legacy systems, and type-related modifications such as the Euro and year 2000 problem.

Another major application is to use type inferencing to support the migration of COBOL to the new COBOL standard, which is an object-oriented extension of COBOL-85 [13]. This new version of COBOL does support types, and offers the possibility of using type definitions. Type inferencing supports the detection of these types in existing COBOL programs, thus allowing old systems to benefit from the new language features.

3. Implementation using relational algebra

This section describes how we use relational algebra to implement type inference for COBOL systems. We start by giving an overview of the tool architecture. Then, we describe the facts that are derived from COBOL sources. We continue with a discussion of how these facts are combined and abstracted to infer more involved type relations. Finally, we describe the extension of this approach to the system level.

3.1. Tool architecture

The set of tools we use for applying type inference to COBOL systems is shown in Fig. 2. It separates source code analysis, inferencing and presentation, making it easier

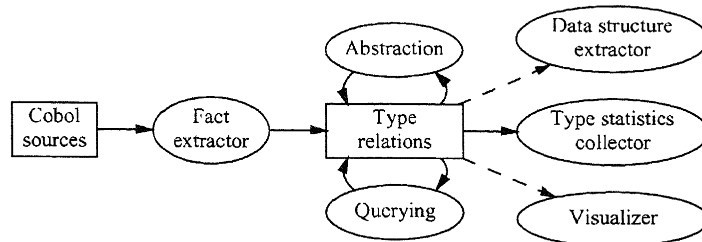

Fig. 2. Overview of the type inference tool set.

to adapt the toolset to different source languages or other ways of presenting the types found.

In the first phase, a collection (database) of *facts* is derived from the COBOL sources. For that purpose, we use a parser generated from the COBOL grammar discussed in [2]. The parser produces abstract syntax trees (ASTs) in a textual representation called the AsFix format. These ASTs are then processed using a Java package which implements the visitor design pattern [9]. The fact extractor itself is a refinement of this visitor which emits type facts at every node of interest (for example, assignments, relational expressions, etc.).

In the second phase, the derived facts are combined and abstracted to infer a number of conclusions regarding type relations. Both facts and conclusions are stored in a simple ASCII format, as also used in, for example, Rigi [17]. One of the tools we use for inferring type relations is `grok` [12], a calculator for *relational algebra* [23]. Relational algebra provides operators for relational composition, for computing the transitive closure of a relation, for computing the difference between two relations, and so on. We use it, for example, to turn the derived type facts into the required equivalence relation. In addition to relational algebra, we use Unix tools like `sort`, `uniq`, `awk`, etc., to manipulate the relation files.

In the final phase, we pass information about the type relations to the end-user. In this paper, we conduct an analysis of the effects of pollution, for which we collect and present a range of statistical data. Other options include the generation of data structures in a language supporting explicit type definitions, and visualization of type information via graphs.

3.2. Derived facts

The different kinds of facts derived from the COBOL sources are listed in Fig. 3. The contain and union relations are derived from the data division, the remaining ones from the procedure division.

Observe that the relations in this figure indicate the degree of language independence of type inferencing: it can be applied to any language from which these facts can be derived. Other languages like Fortran, C, or IBM 370 assembly, can be analyzed by

relation	dom	rng	description
assign	T_v	T_u	an expression of type T_v is assigned to a variable of type T_u
expression	T_v	T_u	variables of types T_v and T_u are used in the same expression
arrayIndex	T_a	T_i	variable of type T_i is used as index in array of type T_a
contain	T_r	T_f	structured type T_r contains T_f
union	T_v	T_u	types T_v and T_u are part of the same union type
literalAssign	T_v	l	literal l is assigned to a variable of type T_v
literalExp	T_v	l	literal l is compared to a variable of type T_v
arrayLitIdx	T_a	l	literal l is used as index in array of type T_a

Fig. 3. Derived facts.

relation	dom	rng	description
typeEquiv	T_1	T_2	type T_1 is equivalent to type T_2
subtypeOf	T_1	T_2	type T_1 is subtype of type T_2
literalType	T	l	type T contains literal l

Fig. 4. Inferred relations.

adding a parser and fact extractor for those languages. Furthermore, since the facts for different languages can easily be combined, this approach allows for the transparent analysis of multi-language systems where, for example, some parts are written in COBOL and other parts are written in assembly.

3.3. *Inferred relations*

The resolution process infers relations between types from the facts that were derived from the COBOL system. Our resolution process is based on relational algebra and is implemented using `grok` [12].

The three key relations inferred are typeEquiv, subtypeOf, and literalType, summarized in Fig. 4.

Besides the relations in Fig. 4, some auxiliary relations are inferred. These include: arrayIndexEquiv for equivalence of types through array access (if variables i and j are used as indexes for the same array A, their types should be equivalent), subtypeEquiv for type equivalence through subtyping (if $T_A \preccurlyeq T_B$ and $T_B \preccurlyeq T_A$, we get $T_A \equiv T_B$), and transSubtypeOf for the transitive closure of subtypeOf.

The resolution algorithm is outlined in pseudo code in Fig. 5. The operators used are those of relational algebra and can be mapped directly to `grok` operators. Note that

```
arrayIndexEquiv := arrayIndex⁻¹ ∘ arrayIndex
typeEquiv := arrayIndexEquiv ∪ expression
subtypeOf := assign
repeat
  subtypeEquiv := equiv(subtypeOf+ ∩ (subtypeOf+)⁻¹)
  typeEquiv := equiv(typeEquiv ∪ subtypeEquiv)
  subtypeOf := subtypeOf \ typeEquiv
  subtypeOf := subtypeOf ∪ subtypeOf ∘ typeEquiv ∪ typeEquiv ∘ subtypeOf
until fixpoint of (typeEquiv, subtypeOf)
literalType := typeEquiv ∘ (literalExp ∪ literalAssign
                            ∪ (arrayIndex⁻¹ ∘ arrayLiteralIndex))

fun equiv(R) := (R ∪ R⁻¹)*
```

Fig. 5. Outline of the resolution algorithm.

function abstraction and unbounded iteration are not available in `grok`. For this reason, in the actual implementation the functions were written out explicitly and bounded iteration is used. The number of iterations was determined heuristically; for the case study conducted, 5 iterations were sufficient. We were informed that addition of unbounded iteration is considered for future releases of `grok`.

3.4. System-level types

In order to do system-level type inference, the primitive types have to be unique for the whole system. As described in Section 2.5, this can be done by qualifying them with program names. Primitive types derived from copybooks that are included in the data division should be qualified using the copybook's name—this ensures that variables of those types will have the same type in all the programs that this copybook is included in.

However, this approach does not allow us to deal with system-level type inference without loading all COBOL sources in memory at once. We would need to analyze self-contained clusters of programs and copybooks, in order to qualify types with the correct names. Such clusters are likely to become as large as the complete system.

To facilitate complete separation of the analysis of copybooks and programs, we derive all information as before, and add extra facts from COBOL sources concerning the use of copybooks and declaration of types. The extra relations are described in Fig. 6.

Next, we compose the copy and decl relations, and infer a copyOf relation that indicates which types used in a program are actually "copies" of types that were

relation	dom	rng	description
decl	m	T_v	module m declares T_v
copy	m_1	m_2	module m_1 imports m_2
actualParam	$P.n$	T_v	nth actual parm. of P has type T_v
formalParam	$P.n$	T_v	nth formal parm. of P has type T_v

Fig. 6. Derived system-level relations.

copyOf	T_p	T_c	T_p is a copy of T_c

Fig. 7. Inferred system-level relations.

declared in a copybook (Fig. 7). This join is done on the imported module field m_2 of the copy relation with the module field m of the decl relation.

Finally, the copyOf relation between T_p and T_c is interpreted as a substitution on the derived relations replacing all occurrences of T_p by T_c. This substitution propagates type dependencies through copybooks.

At this point we have achieved the same database as we would have obtained by analyzing all sources at once, but now using a *modular* approach. Such a modular approach allows us to analyze large industrial-scale systems that are too big to be handled in memory at once.

Example 1. Suppose we derive the following information from programs P and Q:

subtypeOf P.A P.B *copy P Z* *decl Z Z.B*
subtypeOf Q.B Q.C *copy Q Z*

Program P and Q both use variable B and import copybook Z in which B is declared. Joining the copy and decl relations yields two copyOf facts:

copyOf P.B Z.B *copyOf Q.B Z.B*

After substituting these in subtypeOf, we get

subtypeOf P.A Z.B *subtypeOf Z.B Q.C*

Observe that, via transitivity of the subtypeOf relation, we can now infer that $P.A$ is a subtype of $Q.C$, a relation that could not have been found without the propagation through the copybook.

We have written a dedicated C program to perform the substitution since standard Unix tools like `sed` or `perl` could not handle the amount of substitutions involved.[4]

[4] For example, for Mortgage, the copyOf relation contains 121,915 tuples.

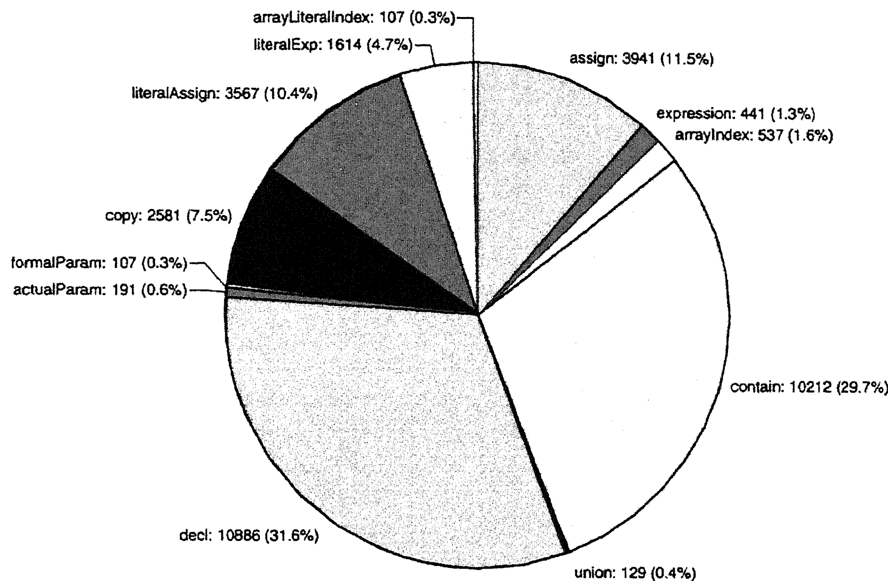

Fig. 8. Facts derived from Mortgage.

Time complexity of this program is $O(n \log n + m \log n)$ (where n is the number of tuples in copyOf, and m is number of tuples in the database), and its space requirements are $O(n)$.

4. Assessing derived facts

In this section we study the nature of the facts that can be directly derived from the COBOL sources, i.e., without applying the resolution step. This means that we only look at the intra-module dependencies, and only consider direct subtype relationships, not transitive ones. This will help us to understand to what extent individual programs are responsible for causing pollution. In the next section, we will look at inter-module dependencies, and relationships arising from taking the transitive closure.

The database that is derived from the Mortgage sources contains 34.313 unique facts. An overview of these is shown in Fig. 8. All duplicates were removed, thus, if variable v is assigned to variable u in two different statements in a certain program, this results in only one subtype relation between T_v and T_u. The majority of facts are from decl and contain. Type equivalence and subtype relationships are inferred from the remaining facts. An interesting observation is that the assign relation is almost 9 times as large as the expression relation. This means that variables in a COBOL program are much more often moved around (assigned) than tested for their value. In this section, we will particularly look at these assign-facts.

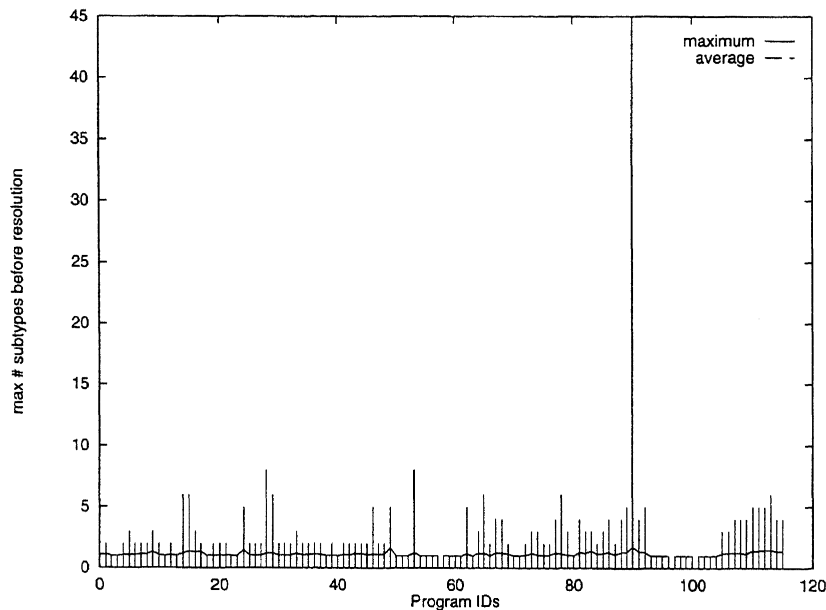

Fig. 9. Maximum number of subtypes per program before resolution.

4.1. Direct subtypes per type

A variable that receives values from many different other variables is a potential cause for pollution. Therefore, in this section we will search for those types that have many different subtypes, i.e., types of variables that are assigned values from many other variables.

In Fig. 9 we show, for each program, the highest number of different subtypes that a single type has. The numbers at the x-axis can be seen as program IDs—they are given in order of increasing program size. As an example, the program with ID-number 20 (one of the smaller programs) has a pulse of length 2 associated with it, i.e., the type with the most different subtypes just has 2 different subtypes.

The dashed line indicates the *average* number of subtypes per type. It shows that most types have just 1 or 2 subtypes. To compute the average number of subtypes per type, only those types that have at least one subtype were taken into account (hence this average will always be larger than 1), ignoring types that were not used at all, or only in expressions. The overall average number of subtypes is 1.18.

Most programs do not contain types with more than 5 subtypes; one program contains a type with an exceptionally large number of 45 different subtypes. If we look at the COBOL code underlying these data, we can understand the high maximum of 45. This involves the type of a variable called P800-LINE, which is a string of length 132. It acts as the formal parameter of a section called Y800-PRINT-LINE. Whenever data

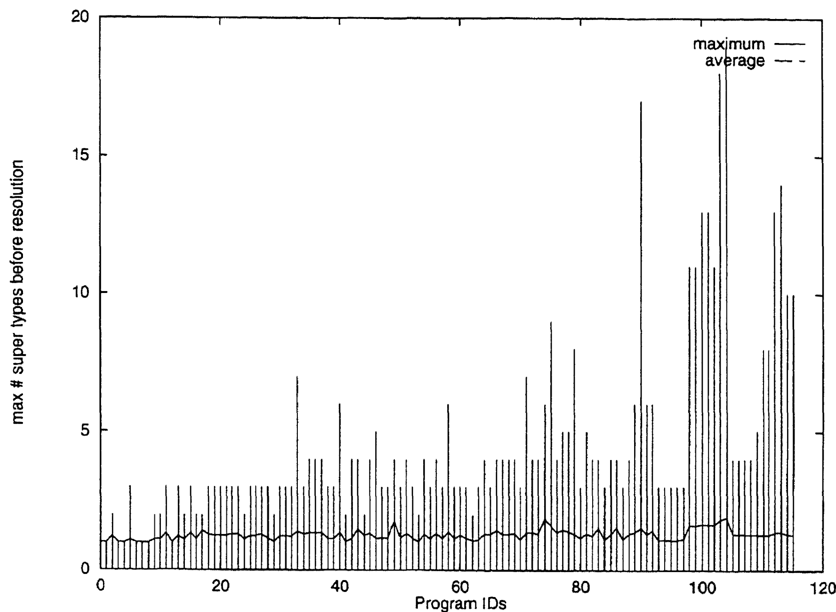

Fig. 10. Maximum number of supertypes per program before resolution.

is to be printed, it is moved into that variable and the Y800-PRINT-LINE section is called. Type inference concludes that the types of all the variables that are printed this way, are subtypes of the type of Y800-PRINT-LINE.

4.2. Direct supertypes per type

Another figure of interest consists of the number of *supertypes* per primitive type, i.e., types of variables that are assigned to many other variables. Fig. 10 shows the number of supertypes per type. Again, most types that have a supertype have one or two supertypes, the average being 1.32. Most of the maxima are below 6, but a number of programs contain types with many more supertypes, for example with 17, 18, or 19 different ones.

If we look at the COBOL source code, we can explain the role of these types. The type with 19 supertypes occurs in a (fairly large) program with ID-number 104, and turns out to be the type of a CURSOR variable, used in a CICS interactive setting. We will refer to this type as CURSOR-type. The variable of this type navigates through the screen positions of a terminal. It is compared with, and copied into a number of different variables representing screen positions of certain fields, such as the position where to enter the name of a person. All these positions together, each declared with numeric picture, share one subtype: the CURSOR-type. Thus, number 19 is not due to pollution, but rather provides meaningful information for understanding the program,

namely that all these types share the values of their common CURSOR-subtype. This CURSOR-mechanism is used by many different programs, and thus is the explanation for most of the maxima higher than 6 occurring in Fig. 10.

One of the non-CURSOR cases occurs in the program with ID-number 90. It concerns a so-called DESCRIPTION-type which has 17 different supertypes. It is the type of an output field of a procedure for reading a value from a particular database. The particular database contains a wide variety of data, and depending on some of the input parameters, different sorts of data are returned. Each of these becomes a supertype of the DESCRIPTION-type.

4.3. *Type equivalence*

In addition to looking at the subtype relations, we can look at the direct type equivalence relations we derive, i.e., we look at types that occur in the same relational or arithmetic expressions. The statistics derived needed for this is based on fewer input tuples, as we know from Fig. 8 that there are almost 9 times fewer expression tuples than assign tuples. The resulting figure, however, is quite similar to Fig. 10, so we omitted the figure in the paper.

If we look at the maxima, they are again 19, 18, and lower. As with the supertypes, one of the types responsible for this is the CURSOR-type. A variable of this type is compared with 18 other variables. Therefore, we conclude that the types of these 18 variables must be the same as the CURSOR-type. The resulting type represents a screen position.

Another type that is equivalent to many other types is the so-called DFHBMEOF-type. This is the type of a special CICS variable which has a constant value for a certain control character. After reading the input entered from a screen, the status characters for the strings that were read are compared with this CICS variable. The types of those status characters are thus equivalent to the type of that CICS variable in our approach.

5. Assessing inferred relations

In this section we examine the relations that result from applying the resolution step. This will help us to understand the merits of resolution and how it affects type pollution.

Before executing the resolution process, we prepare the derived facts for system-level analysis. The copyOf relation that is inferred from the copy and decl relations contains 121,915 tuples. The propagation of copyOf information in the derived database takes 6 s. The resolution was done using a grok script implementing the algorithm in Fig. 5 which takes 7 min for the case study at hand (on Sun Ultra 10, 300 MHz, 576 M memory).

After resolution, the database contains 202,848 tuples. An overview of these is shown in Fig. 11. For a number of relations (such as arrayIndex or literalExp), the number

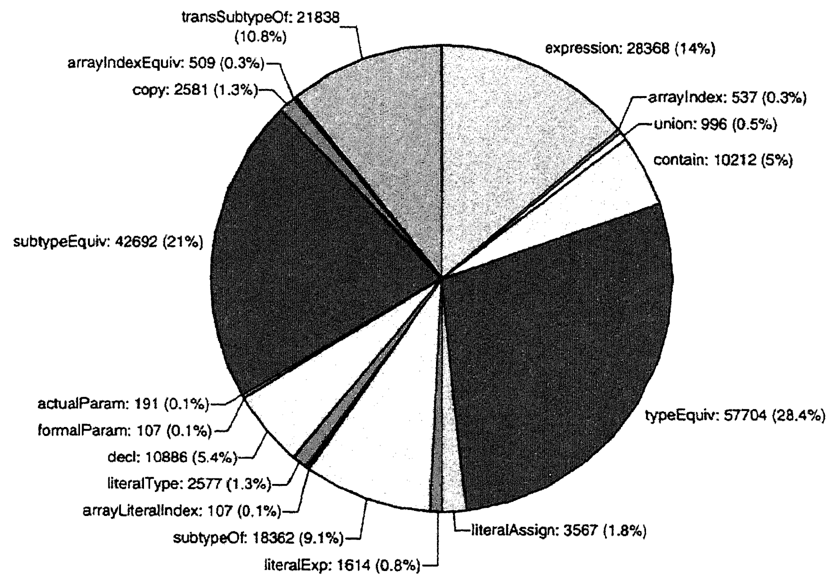

Fig. 11. Information inferred from Mortgage.

of tuples in the resulting database is *smaller* than before since the substitution results in some tuples become duplicates. For others, such as subtypeOf, the number of tuples increases, via propagation of the equivalence relation.

5.1. Subtype relation

One of the goals of the resolution process is to improve the subtypeOf relation by removing tuples for which we have more specific information, namely that they are part of the typeEquiv relation. On the other hand, the subtypeOf relation is also extended with information of the typeEquiv relation. For example, if $T_A \preccurlyeq T_B$ and $T_B \equiv T_C$ then also $T_A \preccurlyeq T_C$. The percentage of subtypes that are added or removed as a result of both modifications is shown in Fig. 12.

In this figure we see that for most programs, resolution reduces the number of subtypes. The average reduction in these programs is 18.4% with a maximum of 47.1%. There are however a couple of programs in which the number of subtypes grows. The average growth in these programs is 54.5% and the maximum is 393.8%. Inspection of these programs shows that the cause of these large numbers is again the CURSOR that was earlier described in Sections 4.2 and 4.3. The reason for this is that CURSOR is the subtype of a lot of types (say set S), *and* it is equivalent to a number of types (say set E). Since the resolution process ensures that all types in set E become subtypes of all the types in set S, the resulting database contains a rather large number of subtypes ($|S| \times |E|$ to be precise) just because of this CURSOR.

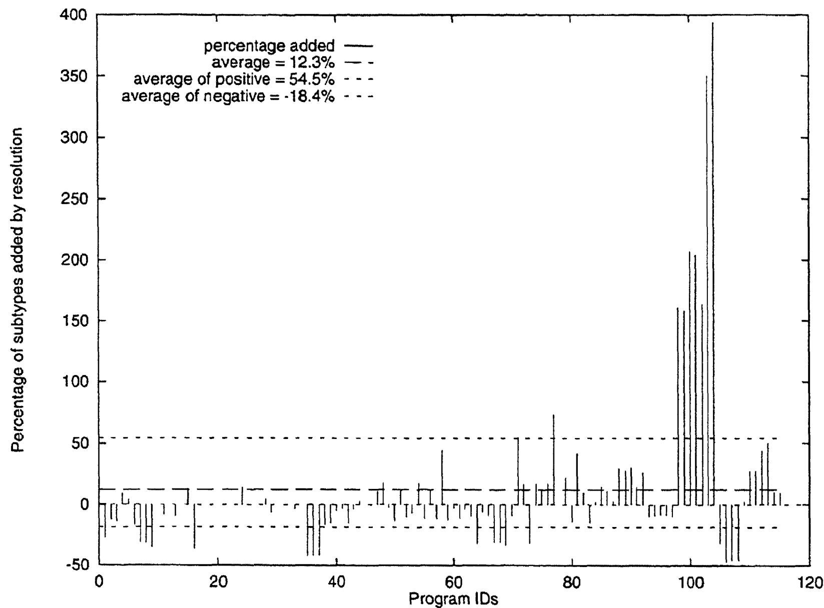

Fig. 12. Subtypes added by resolution.

As not all variables are used in comparisons (recall that in COBOL it is very common to just move variables), other types with many sub- or supertypes (such as DESCRIPTION and P800-LINE) but which are never used in comparisons, play no role of importance here.

5.2. *Type equivalence*

The typeEquiv partitions types into equivalence classes. An overview of all classes that occur in Mortgage and their sizes is presented in Fig. 13.

Fig. 13(a) contains the classes if resolution is only done on a per-program basis, i.e., without taking system-wide propagation via copybooks and program calls into account. On this program level, resolution does not have a big influence on these equivalence classes. The explanation for this is that the classes at the program level are small and tightly connected, so all relations are already found by analyzing the code (e.g., if 3 variables are equivalent, they will all be compared to each other so the transitive closure does not find new tuples). The maxima are still 19 and 18 and the average class size is 3. Furthermore, approx. 90% of the classes have less than 5 equivalent elements.

Things get more interesting at the system level presented in Fig. 13b. The maximum class size jumps to 201, followed by 118 but the total number of different classes drops to 191, one-third of the number of classes before resolution. Again, approximately 90% of the classes have less than 5 equivalent elements.

class size	# of classes	percent of total
2	373	63.4%
3	99	16.8%
4	53	9.0%
5	10	1.7%
6	8	1.4%
7	29	4.9%
8	1	0.2%
10	1	0.2%
11	5	0.9%
12	1	0.2%
13	4	0.7%
18	2	0.3%
19	2	0.3%
sum	588	100.0%

(a) program level

class size	# of classes	percent of total
2	135	70.7%
3	22	11.5%
4	5	2.6%
5	4	2.1%
6	9	4.7%
7	6	3.1%
8	2	1.0%
10	1	0.5%
11	1	0.5%
12	1	0.5%
13	1	0.5%
24	1	0.5%
39	1	0.5%
118	1	0.5%
201	1	0.5%
sum	191	100.0%

(b) system level

Fig. 13. Size and frequency of equivalence classes.

Inspection of the derived equivalence classes shows that the class with 201 elements contains all elements that are equivalent to the `CURSOR`-type. All `CURSOR` classes occurring in different programs are taken together, as the underlying `CURSOR` variable is declared in a copybook. When we look at the code we see that the elements in this class are typically used in a relational expression with the `CURSOR`-type, although in some cases they are both a sub- and supertype of it and therefore inferred to be equivalent.

The next biggest class has 118 elements and represents a type holding some CICS status information. It contains all elements equivalent to the `DFHBMEOF`-type described in Section 4.3, again coming from a copybook.

The class with 39 elements represents the index type for some array type. The elements in this class were typically found using the rule for array index equivalence. It contains the primitive types of variables that were used to access arrays in loops and those that were used for checking array bounds. Here the array variable was declared in a copybook.

The last class we will discuss here is the one with 24 elements. This class represents the so-called RELATION-ID-type and is worth mentioning since it contains a form of pollution that is not solved by subtyping. The spurious type is the so-called MORTGAGE-ID-type which is unrelated to the RELATION-ID-type according to the business logic. The reason that they end up in the same class is that both types are used as parameter of a "function" that does a sanity check on the number (11-check) *and* return the corrected number when necessary. In the call both types become subtypes of the input type of that function. After the call, the output is moved back so the output type becomes a subtype of RELATION-ID and MORTGAGE-ID. Since the input and output type for this function is the same, RELATION-ID becomes a subtype MORTGAGE-ID and vice versa so they are considered to be equivalent.

We can solve such pollution by deriving an additional cast relation during fact extraction. Whenever a variable of a supertype is assigned to a variable of a subtype, we derive that the supertype is casted into the subtype. Furthermore, we can use data flow analysis to derive what are the input parameters of a function and what are the output parameters of a function. This mechanisms also allows us to deal with explicit casts as, for example, can occur in C programs.

6. Related work

A principal source of inspiration to us was Lackwit, a tool for understanding C programs by means of type inference [20]. Lackwit performs a type analysis of variables based on their usage. The analysis results are used to find abstract data types, detect abstraction violations, identify unused variables, and to detect certain types of errors. New in our work is not only the significantly different source language, but also the use of subtyping for dealing with pollution, and the use of type inference to classify literals. Another paper discussing type inference for C is by Sniff and Reps [22], who use inferred types to generalize C functions to C++ function templates.

Our approach is also related to various tools for the analysis and correction of the year 2000 problem where date *seeds* are tracked through the statements in a program [11,14,18]. In year 2000 analysis, preventing pollution (called *classification noise* in [14]) is an important issue. We have not been able to find papers that propose the use of subtyping to do this. Our current paper adds a strong empirical basis for using subtyping to reduce pollution.

Recently, two papers appeared which rely on type theory to deal with the year 2000 problem [7,21]. These papers do not address the problem of pollution, but do contain an interesting algorithm for propagating type information through the elements of aggregate data structures such as arrays or records. Our approach essentially treats each aggregate as a single scalar value. If, however, two entire records are moved, types can also be propagated through the individual fields. Such moves may even cross field boundaries if the two records differ in record layout, or if records are aliased using COBOL's redefine statement. Ramalingam et al. [21] and Eidorff et al.

[7] provide an algorithm that finds a minimal splitting of all aggregates such that types can be correctly propagated for the resulting "atoms". In our earlier paper [5], we proposed a weaker method using an inference rule called *substructure completion*, which just ensures that type equivalences between structurally equivalent aggregates are propagated to the components. As discussed later, we plan to combine this algorithm with our type inferencing approach to see if we can further improve the accuracy.

Chen et al. [3] describe a (semi)-automatic approach for COBOL *variable classification*. They distinguish a fixed set of categories, such as input/output, constant, local variable etc., and each variable is placed into one or more of these classes. They provide a set of rules to infer this classification automatically, essentially using data flow analysis. Their technique is orthogonal to ours: the types we infer can be used for both local and global variables, for variables that are used for databases access and for those that are not, etc.

Newcomb and Kotik [19] describe a method for migrating COBOL to object orientation. Their approach takes all level 01 records as starting point for classes. Records that are structurally equivalent, i.e., matching in record length, field offset, field length, and field picture, but possibly with different names, are called "aliases". According to Newcomb and Kotik, "for complex records consisting of 5–10 or more fields, the likelihood of false positives is relatively small, but for smaller records the probability of false positives is fairly large" [19, p. 240]. Our way of type inferencing may help to reduce this risk, as it provides a complementary way of grouping such 01 level records together based on *usage*.

Wegman and Zadeck [24] describe a method to detect whether the value of a variable occurring at a particular point in the program is constant and, if so, what that value is. Merlo et al. [16] describe an extension of this method that allows detection of all constants that can be the value of a particular variable occurrence. This differs from our approach which finds all constants that can be assigned to *any* variable of a given type. Furthermore, the methods described in both papers take the flow of control into account where as our approach is flow-insensitive (control flow is completely ignored). Consequently, their results are more precise (e.g., we report constants that are used in dead code) but their approach is also more expensive.

Gravley and Lakhotia [10] identify enumeration types that are modeled using `define` preprocessor directives. Their approach is orthogonal to ours since they group constants which are *defined* "in the same context" (i.e., close to each other in the program text) whereas we group constants based on their *usage* in the source code.

Concerning the tool used for implementing type inference, there is second suite of relational algebra tools available from Philips, as described by Feijs et al. [8]. An alternative to the use of relational algebra, is to view type inferencing as a graph traversal problem. A graph querying formalism such as GReQL [15] can then be used to compute the closures of several relations. A second alternative is to use one of several program analysis frameworks. Of particular interest is BANE, the Berkely ANalysis Engine as described by Aiken et al. [1]. BANE provides constraint specification and

resolution components, which can be to experiment with program analyses in which properties of types are expressed as constraints.

7. Concluding remarks

7.1. Contributions

In this paper, we carried out an empirical study into the relations between variables established by COBOL type inference. We argued that such relations are necessary in a COBOL setting: COBOL programs contain a large number of variable declarations (50% of a program's lines of code consist of variable declarations), but only half of these variables are actually used. Inferred types help to understand how variables are used and how they are related to each other.

The empirical study aimed at finding out how the problem of *pollution* is handled by the use of subtyping. Pollution occurs when a counter-intuitive type equivalence is found for two variables. Since it is impossible to check by hand the hundreds of type equivalences classes found by type inferencing, we devised a suite of numeric measurements directing us to potential pollution spots.

We manually inspected, and explained in the paper, the results from these measurements. Of all inferred type equivalence classes, only one contains a clear case of pollution: in Section 5.2 we discuss how type casts could help to address this problem.

To conduct our experiments, we developed a tool environment permitting all sorts of experiments. An important new element is the use of relational algebra to do the inference of type conclusions from derived type facts. Moreover, we devised a modular approach to infer types for variables playing a system-wide role. Owing to this modular approach, system-level type analysis scales up to large systems.

7.2. Future work

Now that we have all machinery for conducting large scale type inferencing experiments in place, and now that we understand which data to collect, we are in a position to apply type inference to more COBOL systems. We intend to do this, and collect statistical data on other case studies as well.

A question of interest is how we can further improve the accuracy of our type inferencing approach by deconstructing aggregates into "atoms" of the appropriate size, following the algorithm of [7,21]. An important problem to be solved is how to combine this algorithm with *subtyping*, in order to minimize the danger of pollution.

At the moment, we are conducting experiments with new ways of *presenting* type relations [6]. One way is to visualize type relations as graphs. We are integrating such graphs with the COBOL documentation generator covered in [4]. This generator provides an abstract view of COBOL systems, highlighting essential relationships between programs, databases, screens, etc. Types play an important role in this form of

documentation, as they help to characterize the interfaces of COBOL modules, or the interplay of variables occurring in the COBOL programs.

References

[1] A. Aiken, S. Fähndrich, S. Foster, Z. Su, A toolkit for constructing type- and constraint-based program analyses, in: Second Internat. Workshop on Types in Compilation, TIC'98, Lecture Notes in Computer Science, vol. 1473, Springer, Berlin, 1998, pp. 78–96.

[2] M.G.J. van den Brand, A. Sellink, C. Verhoef, Generation of components for software renovation factories from context-free grammars, in: Fourth Working Conf. on Reverse Engineering, WCRE'97, IEEE Computer Society, Silver Spring, MD, 1997, pp. 144–155.

[3] X.P. Chen, W.T. Tsai, J.K. Joiner, H. Gandamaneni, J. Sun, Automatic variable classification for COBOL programs, in: 18th Ann. Internat. Computer Software and Applications Conf. COMPSAC'94, IEEE Computer Society, Los Alamitos, CA, 1994, pp. 432–437.

[4] A. van Deursen, T. Kuipers, Building documentation generators, Internat. Conf. on Software Maintenance, ICSM'99, IEEE Computer Society, 1999, pp. 40–49.

[5] A. van Deursen, L. Moonen, Type inference for COBOL systems, in: M. Blaha, A. Quilici, C. Verhoef (Eds.), Fifth Working Conf. on Reverse Engineering, WCRE'98, IEEE Computer Society, Silver Spring, MD, 1998, pp. 220–230.

[6] A. van Deursen, L. Moonen, Exploring legacy systems using types, in: Seventh Working Conf. on Reverse Engineering, WCRE'00, IEEE Computer Society Press, Silver Spring, MD, 2000.

[7] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sorensen, M. Tofte, Anno Domini: From type theory to Year 2000 conversion tool, in: 26th Ann. Symp. on Principles of Programming Languages, POPL'99. ACM, New York, 1999.

[8] L. Feijs, R. Krikhaar, R. van Ommering, A relational approach to support software architecture analysis, Software Practice Exp. 28 (4) (1998) 371–400.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.

[10] J.M. Gravley, A. Lakhotia, Identifying enumeration types modeled with symbolic constants, in: Third Working Conf. on Reverse Engineering, WCRE'96, IEEE Computer Society Press, Silver Spring, MD, 1996, pp. 227–236.

[11] J. Hart, A. Pizzarello, A scaleable, automated process for year 2000 system correction, in: 18th Internat. Conf. on Software Engineering, ICSE-18, IEEE, New York, 1996, pp. 475–484.

[12] R. Holt, Structural manipulations of software architecture using Tarski relational algebra, in: M. Blaha, A. Quilici, C. Verhoef (Eds.), Fifth Working Conf. on Reverse Engineering, WCRE'98, IEEE Computer Society, Silver Spring, MD, 1998, pp. 210–219.

[13] ISO, Programming language COBOL: Proposed Revision of ISO 1989:1985. International Standardization Organization, 2000, Committee Draft 1.8, ISO/IEC CD 1.8 1989 : yyyy(E).

[14] K. Kawabe, A. Matsuo, S. Uehara, A. Ogawa, Variable classification technique for software maintenance and application to the year 2000 problem, in: P. Nesi, F. Lehner (Eds.), Second European Conf. on Software Maintenance and Reengineering, CSMR'98, IEEE Computer Society, Silver Spring, MD, 1998, pp. 44–50.

[15] B. Kullbach, A. Winter, Querying as an enabling technology in software reengineering, in: Third European Conf. on Software Maintenance and Reengineering, CSMR'99, IEEE Computer Society, Silver Spring, MD, 1999, pp. 42–50.

[16] E. Merlo, J.F. Girard, L. Hendren, R. De Mori, Multi-valued constant propagation analysis for user interface reengineering, Internat. J. Software Eng. Knowledge Eng. 5 (1) (1995) 5–23.

[17] H.A. Müller, M.A. Orgun, S.R. Tilley, J.S. Uhl, A reverse engineering approach to subsystem structure identification, J. Software Maintenance 5 (4) (1993) 181–204.

[18] M.G. Nanda, P. Bhaduri, S. Oberoi, A. Sanyal, An application of compiler technology to the year 2000 problem, Software Practice Experience 29 (4) (1999) 359–377.

[19] P. Newcomb, G. Kottik, Reengineering procedural into object-oriented systems, in: Second Working Conf. on Reverse Engineering, WCRE'95, IEEE Computer Society, Silver Spring, MD, 1995. pp. 237–249.

[20] R. O'Callahan, D. Jackson, Lackwit: A program understanding tool based on type inference, in: 19th Internat. Conf. on Software Engineering, ICSE-97, ACM/IEEE, New York, 1997.
[21] G. Ramalingam, J. Field, F. Tip, Aggregate structure identification and its application to program analysis, in: 26th Annu. Symposium on Principles of Programming Languages, POPL'99, ACM, New York, 1999.
[22] M. Siff, T. Reps, Program generalization for software reuse, in: ACM SIGSOFT Symp. on the Foundations of Software Engineering, FSE'96, Software Engineering Notes 21(6), 1996, pp. 135–146.
[23] A. Tarski, On the calculus of relations, J. Symbolic Logic 6 (1941) 73–89.
[24] M. Wegman, K. Zadeck, Constant propagation with conditional branches, ACM Trans. Programming Languages Systems 13 (2) (1991) 18–210.