Understanding COBOL Systems using Inferred Types*

Arie van Deursen

CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~arie/`

Leon Moonen

University of Amsterdam, Kruislaan 403
1098 SJ Amsterdam, The Netherlands
`http://adam.wins.uva.nl/~leon/`

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 conclusion that only one instance of pollution was found in the case study conducted.*

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 division.[1]

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.[2]

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, ...), 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

*Copyright 1999 IEEE. Published in the International Workshop on Program Comprehension 1999 (IWPC'99) May 5-7, 1999 in Pittsburgh, PA, USA. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

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

[2] 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.

division suffer from a number of problems that make them unsuitable to fulfil 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.

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 [4], we have proposed a solution to this problem. Instead of deriving type information from the data division, we *infer* types from the usage of variables in the procedure division. The basic idea is simple: if two variables are used in an assignment or comparison, we want to infer that these two variables should have the same type. In this paper, we will take a closer look at type inferencing, proposing a new way of implementation by means of *relational algebra*.

Moreover, we will carefully study the problem of *pollution*, which occurs when types become too large, containing variables that intuitively should belong to different types. In [4] we argued that deriving *subtypes* rather than equivalences handles the problem of pollution. In our current paper we test this hypothesis, 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, if $A \preccurlyeq B$ and $B \preccurlyeq A$, we get $A \equiv B$ — how does this affect pollution?).

All experiments are done on Mortgage, a real-life COBOL/CICS system from the banking environment. With all copybooks (include files) expanded (unfolded), it consists of 250,000 lines of code; unexpanded it consists of 100,000 lines of code.

2. Type Inference

In this section, we summarise the essentials of COBOL type inferencing: a more complete presentation is given in [4].

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 variables must have unique names in a COBOL program, they can be used as labels within a type to ensure uniqueness. We qualify variable names with program names to obtain uniqueness at the system level. We use T_A to denote the primitive type of variable A.

Equivalence By looking at the *expressions* occurring in statements, an *equivalence relation* between primitive types can be inferred. We distinguish three cases: (1) For relational expressions such as $v = u$ or $v \leq u$, an equivalence between T_v and T_u is inferred. (2) For arithmetic expressions such as $v + u$ or $v * u$, an equivalence between T_u and T_v is inferred. (3) For two different array accesses $a[v]$ 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*.

Subtyping By looking at the *assignment statements*, a *subtype relation* between primitive types can be inferred. 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.

Union types From a COBOL *redefine clause*, a *union type relation* between primitive types can be inferred. 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*.

System-Level Analysis In addition to inferring type relations within individual programs, we also infer type relations at the system-wide level. Such relations 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. Furthermore, we infer that the types of the actual parameters of a program call (listed in the `USING` clause) are subtypes of the formal parameters (listed in the linkage section), and that variables read from or written to the same databases have equivalent types.

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, it can be used for finding *enumeration types*.

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 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.

3. Tool Architecture

The set of tools we use for applying type inference to COBOL systems is shown in Figure 1. It separates source code analysis, inferencing and presentation, making it easier to adapt the

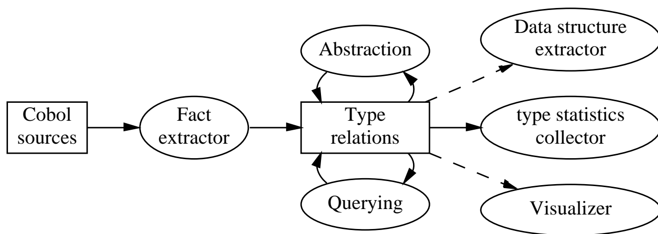

Figure 1. Overview of the type inference tool set.

Relation	dom	rng	description
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
subtypeOf	T_v	T_u	type T_v is subtype of T_u (variable v is assigned to u)
expression	T_v	T_u	variables of types T_v and T_u are used in the same expression
literalAssign	T_v	l	literal l is assigned to a variable of type T_v
literalTest	T_v	l	literal l is compared to a variable of type T_v
arrayIndex	T_a	T_i	variable of type T_i is used as index in array of type T_a
arrayLitIdx	T_a	l	literal l is used as index in array of type T_a

Table 1. Derived Facts

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 [1]. 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. The fact extractor 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, for example, in Rigi [9]. One of the tools we use for inferring type relations is `grok` [8], a calculator for *relational algebra* [12, 6]. 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 will mainly do this based on metrics, via the use of `gnuplot`. Other options include the generation of data structures in a language supporting explicit type definitions, and visualisation of type information via graphs.

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 T_2
literalType	T	l	type T contains l

Table 2. Inferred Relations

arrayIndexEquiv := arrayIndex^{-1} $\circ$ arrayIndex
subtypeEquiv := $\emptyset$
repeat
  subtypeEquiv := equiv(subtypeOf$+$ $\cap$
        (subtypeOf$+$)$^{-1}$)
  typeEquiv := equiv(arrayIndexEquiv $\cup$
        subtypeEquiv $\cup$ expression)
  subtypeOf := subtypeOf $\setminus$ typeEquiv
  subtypeOf := subtypeOf $\cup$ subtypeOf $\circ$ typeEquiv
        $\cup$ typeEquiv $\circ$ subtypeOf
until fixpoint of (typeEquiv, subtypeOf)
literalType := typeEquiv $\circ$ (literalTest $\cup$ literalAssign
        $\cup$ (arrayIndex^{-1} $\circ$ arrayLiteralIndex))

fun equiv(R) := $(R \cup R^{-1})*$

Figure 2. Outline of the resolution algorithm.

3.1. Derived Facts

The different kinds of facts derived from the COBOL sources are listed in Table 1. The contain and union relations are derived from the data division, the remaining ones from the procedure division.

Observe that the relations in this table 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 analysed by 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.

3.2. 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` [8].

The three key relations inferred are typeEquiv, subtypeOf, and literalType, summarised in Table 2. The inferred subtypeOf relation is a refinement of the subtypeOf relation directly extracted from the COBOL sources. For example, types

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

Table 3. Derived System-Level Relations

Relation	dom	rng	description
copyOf	T_p	T_c	T_p is a copy of T_c

Table 4. Inferred System-Level Relations

that are also equivalent are removed from subtypeOf.

Besides the relations in Table 2, 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 $A \preccurlyeq B$ and $B \preccurlyeq A$, we get $A \equiv B$), and transSubtypeOf for the transitive closure of subtypeOf.

The resolution algorithm is outlined in pseudo code in Figure 2. The operators used are those of relational algebra and can be mapped directly to `grok` operators. In the pseudo code we use function abstraction and a construction that loops over a body until a fixed point is reached. As these are not available in `grok`, in the actual implementation we decided to write out the functions explicitly and iterate a fixed number of times over the body of the loop (the number is determined heuristically).

3.3. 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 [4], 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 analyse 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 Table 3.

Next, we join 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 declared in a copybook (Table 4). 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 analysing all sources at once, but now using a *modular* approach. Such a modular approach allows us to analyse large industrial-scale systems that are too big to be handled in memory at once.

Example 3.1 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[3]. 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 will help us to understand how many primitive types exist which are *directly* related to other primitive types, and what effect such types have on pollution.

The database that is derived from the Mortgage sources contains 40,889 facts. An overview of these is shown in Table 5. 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.

Observe that the subtypeOf relation is more than 8 times as large as the expression relation, i.e., variables in a COBOL program are much more often moved around (assigned) than tested for their value.

[3] For Mortgage, the copyOf relation contains 121,915 tuples.

relation	tuples	percentage
formalParam	107	0.26%
union	129	0.32%
arrayLiteralIndex	263	0.64%
actualParam	593	1.45%
expression	644	1.57%
arrayIndex	1263	3.09%
copy	2581	6.31%
literalTest	3199	7.82%
subtypeOf	5504	13.46%
literalAssign	5507	13.47%
contain	10212	24.97%
decl	10887	26.63%
total	40889	100.00%

Table 5. Facts derived from Mortgage

4.1. Direct subtypes per type

A question of interest is how many different subtypes each primitive type has. We search for those types which have a large number of different subtypes, i.e., types of variables that get assigned values from many other variables.

In Figure 3 we show, for each program, the highest number of different subtypes that a single type has. In the figure, the programs are sorted ascending by size (lines of code). The numbers at the x-axis can be seen as their program IDs. 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. Finally, since the aim of this figure is to find those types that are directly responsible for a high number of different subtypes: therefore only the *direct* subtype relation was taken into account, rather than its transitive closure.

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 which will be explained later.

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 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 type of `Y800-PRINT-LINE`.

4.2. Direct supertypes per type

Another figure of interest consists of the number of *super* types per primitive type. This time, we particularly search for types with a large number of *supertypes*, i.e., types of variables that are assigned to many other variables.

Figure 4 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 turns out to be a `CURSOR` type, used in a CICS interactive setting. 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, the 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. Most other

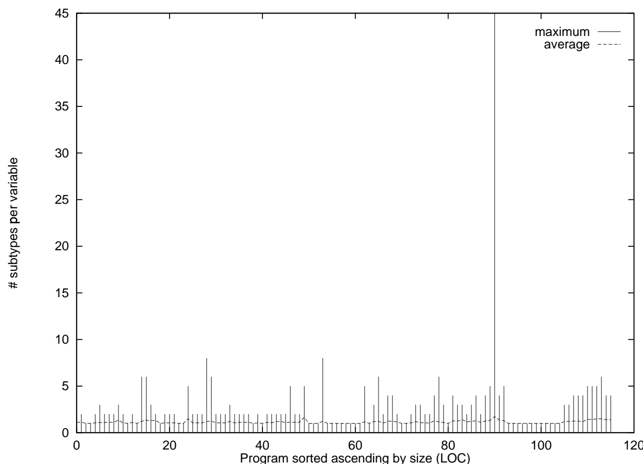

Figure 3. Direct subtypes per type.

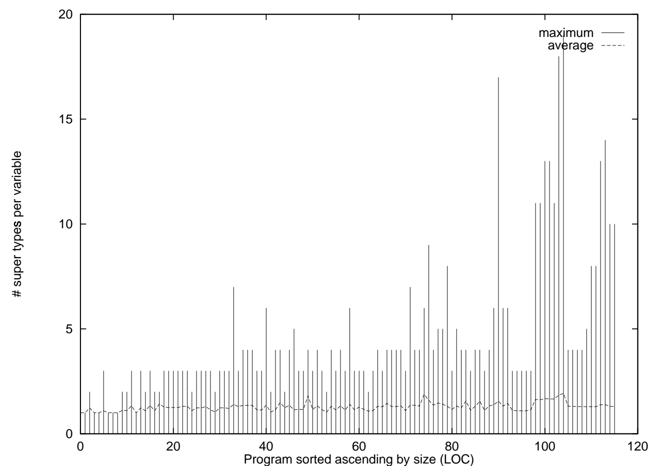

Figure 4. Direct supertypes per type

maxima higher than 6 in Figure 4 are due to such a `CURSOR` type.

One of the non-`CURSOR` cases is a type called `DESCRIPTION` which has 17 different supertypes. It is the type of an output field of a procedure for reading a value from a 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 Table 5 that there are 8 times fewer expression tuples than subtypeOf tuples. The resulting figure, however, is quite similar to Figure 4, 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 `DFHBMEOF`. 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 seconds. The resolution was done using a `grok` script implementing the algorithm in Figure 2 which takes 7 minutes for the case study at hand (on Sun Ultra 10, 300MHz, 576 M memory).

After resolution, the database contains 202,848 tuples. An overview of these is shown in Table 6. For a number of relations (such as arrayIndex or literalTest), the number of tuples in the resulting database is *smaller* than before since the substitution results in some tuples becoming duplicates. For others, such as subtypeOf, the number of tuples increases, via propagation of the equivalence relation.

relation	tuples	percentage
arrayLiteralIndex	107	0.05%
formalParam	107	0.05%
actualParam	191	0.09%
arrayIndexEquiv	509	0.25%
arrayIndex	537	0.26%
union	996	0.49%
literalTest	1614	0.80%
literalType	2577	1.27%
copy	2581	1.27%
literalAssign	3567	1.76%
contain	10212	5.03%
decl	10886	5.37%
subtypeOf	18362	9.05%
transSubtypeOf	21838	10.77%
expression	28368	13.98%
subtypeEquiv	42692	21.05%
typeEquiv	57704	28.45%
total	202848	100.00%

Table 6. Information inferred from Mortgage

5.1. Subtype relation

One of the goals of the resolution process is to improve the subtypeOf relation by removing tuples that are also equivalent. On the other hand the subtypeOf relation is also extended with information of the typeEquiv relation (e.g., if $A \preccurlyeq B$ and $B \equiv C$ then $A \preccurlyeq C$). The percentage of subtypes that are added or removed as a result of both modifications is shown in Figure 5.

In this figure we see that for most programs, resolution reduces the number of subtypes (i.e., resolution cleans up the subtypeOf relation). The average reduction in these programs is 18.4% with a maximum of 47.1%. There are however a cou-

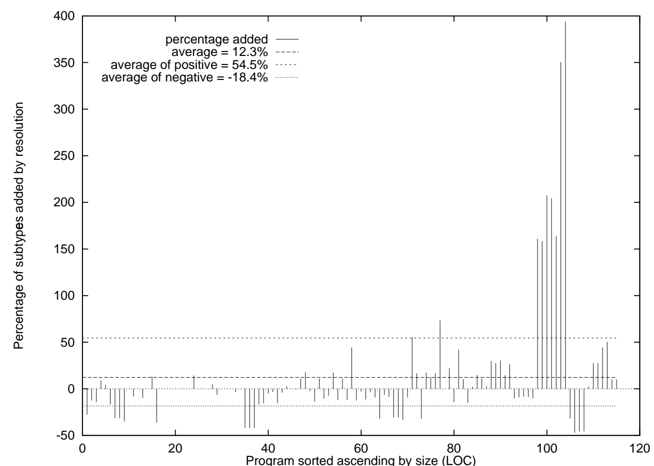

Figure 5. Subtypes added by resolution

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

Table 7. Size and frequency of equivalence classes

ple 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 type that was earlier described in Sections 4.2 and 4.3. The CURSOR type is the subtype of a lot of types (say set *S*), *and* it is equivalent to a number of types (say set *E*). The resolution process ensures that all types in set *E* become subtypes of the types in set *S*.

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. On the 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 analysing 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). An overview of all classes that occur in Mortgage and their sizes is presented in Table 7a. 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 Table 7b. The maximum class size jumps to 201, followed by 118 but the total number of different classes drops to 190, one third of the number of classes before resolution. Again, approx. 90% of the classes have less than 5 equivalent elements.

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 type DFHBMEOF 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 19 elements. This class represents the RELATION-ID type and is worth metioning since it contains a form of pollution that is not solved by subtyping. The spurious type is a 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* returns 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- and what are the output- parameters of a function. This mechanism also allows us to deal with explicit casts as, for example, can occur in C programs.

6. Related Work

Type inference for COBOL is related to earlier work on type inference for C [10], (semi)-automatic classification of COBOL variables [2], and various approaches for detecting and correcting year 2000 problems [7]. Applications include the use of type inferencing to identify objects in legacy code [3]. A more detailed overview of related work is given in [4]. Unique in our approach is the use of subtyping for dealing with pollution. Our current paper adds a strong empirical basis for this

approach.

Recently, two new related papers have appeared. One of these uses type theory as the basis for a year 2000 conversion tool, using an approach like type inference to determine date-related variables [5]. The other carefully analyses the structure of COBOL records. It decomposes aggregates such as records and arrays into simpler components based on the access patterns specific to a given program [11]. Based on an additional analysis of the COBOL picture clauses, records are split into "atoms". This analysis of pictures is orthogonal to our approach, and looks like a valuable addition.

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 [8]. Moreover, we devised a modular approach to infer types for variables playing a system-wide role. Thanks 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 further statistical data on other (larger) case studies.

Moreover, we plan to experiment with ways of *visualising* type relations and to investigate additional ways of querying the derived facts and performing the resolution step. In particular, we plan to look at the relational tool set discussed by [6].

A natural extension of our work is to conduct experiments with *metrics* based on variable relations. Can the data we collected also be used to predict attributes of software, such as maintainability? This might be a valuable addition to the more common metrics which are mostly focused on complexity induced by statements and control structures.

References

[1] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *4th Working Conf. on Reverse Engineering; WCRE'97*, pages 144–155. IEEE, 1997.

[2] X. P. Chen, W. T. Tsai, J. K. Joiner, H. Gandamaneni, and J. Sun. Automatic variable classification for COBOL programs. In *18th Ann. int. Computer Software and Applications Conference; COMPSAC'94*, pages 432–437. IEEE, 1994.

[3] A. van Deursen and T. Kuipers. Finding objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE'99*. ACM, 1999. To appear.

[4] A. van Deursen and L. Moonen. Type inference for COBOL systems. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the fifth Working Conference on Reverse Engineering*, pages 220–230. IEEE Computer Society, 1998.

[5] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte. Anno Domini: From type theory to Year 2000 conversion tool. In *26th Annual Symposium on Principles of Programming Languages, POPL'99*. ACM, 1999.

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

[7] J. Hart and A. Pizzarello. A scaleable, automated process for year 2000 system correction. In *Proceedings of the 18th International Conference on Software Engineering ICSE-18*, pages 475–484. IEEE, 1996.

[8] R. Holt. Structural manipulations of software architecture using Tarski relational algebra. In M. Blaha, A. Quilici, and C. Verhoef, editors, *5th Working Conference on Reverse Engineering, WCRE'98*, pages 210–219. IEEE Computer Society, 1998.

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

[10] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *19th International Conference on Software Engineering; ICSE-97*. ACM, 1997.

[11] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *26th Annual Symposium on Principles of Programming Languages, POPL'99*. ACM, 1999.

[12] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.