Type Inference for COBOL Systems*†

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

Types are a good starting point for various software reengineering tasks. Unfortunately, programs requiring reengineering most desperately are written in languages without an adequate type system (such as COBOL). To solve this problem, we propose a method of automated type inference for these languages. The main ingredients are that if variables are compared using some relational operator their types must be the same; likewise if an expression is assigned to a variable, the type of the expression must be a subtype of that of the variable. We present the formal type system and inference rules for this approach, show their effect on various real life COBOL fragments, describe the implementation of our ideas in a prototype type inference tool for COBOL, and discuss a number of applications.

1. Introduction

The many different variables occurring in a typical program, can generally be grouped into *types*. A type can play a number of roles:

- It is an indication of the set of values that is allowed for a variable;
- A type groups variables that represent the same kind of entities;
- A type helps to hide the actual representation (array versus record, length of array, ...) used;
- Types for input and output parameters of a procedure provide a "signature" of the expected use of that procedure.

Traditionally, types are associated with strongly-typed languages, in which explicit variable and type declarations help to detect programming errors at compile time instead of run time.

In this paper we will be concerned with a rather different use of types. In our opinion, types are a good starting point for various software reengineering activities. We argue that the use of types as described in this paper is in fact the underlying theory of the approach followed by a number of existing reverse engineering tools. For example, types can be used for migrating from a procedural to an object-oriented language, isolating reusable components from legacy sources, searching for potential year 2000 infections, or for searching code that will be affected by the introduction of the Euro: the single European currency.

Unfortunately, systems for which such reengineering activities are most necessary, are generally written in languages with a rather limited type system. This makes reengineering for such languages difficult. To solve this problem, we propose methods to *infer* a set of types from programs written in such languages automatically. These automatically inferred types can then be the starting point for objectification, year 2000 remediation, etc.

The language we deal with in this paper is COBOL. We show how to infer a set of types automatically from (a system of) COBOL programs. We present several varieties of our type system, taking sub-typing, byte representations and inter-program types into account. We describe how we made a prototype tool that performs type inference on COBOL code.

We have evaluated our approach using a case-study where we apply the ideas described above to Mortgage: a 100,000 LOC COBOL system from the banking area. The examples in this paper are taken from that system.

We conclude by describing a number of important applications of our technique in the area of software reengineering.

*This work was sponsored in part by bank ABN AMRO, software house Roccade, and the Dutch *Ministerie van Economische Zaken* (Department of Commerce) via SENTER Project #ITU95017 "SOS Resolver".

†Copyright 1998 IEEE. Published in the 5th Working Conference on Reverse Engineering 1998 (WCRE'98) October 12-14, 1998 in Honolulu, Hawaii, 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.

2. Approach and Motivation

At first sight, COBOL may *appear* to be a typed language. Every variable occurring in the statements of the procedure division, must be declared in the data division first. A typical declaration may look as follows:

```
01 TAB100.
   05 TAB100-POS    PIC  X(01) OCCURS 40.
   05 TAB100-FILLED PIC S9(03) VALUE 0.
```

Here, three variables are declared: `TAB100-FILLED`, which is an integer (picture "`9`") comprising three bytes initialised with value zero; `TAB100-POS`, which is a single character byte (picture "`X`") occurring 40 times, i.e., an array of length 40; and `TAB100` which is a record defined at level `01`, having the two variables with higher level numbers, namely `05`, as fields.

Unfortunately, the variable declarations in the data division suffer from a number of problems, making them unsuitable to fulfil the roles of types as listed in the beginning of this paper. First of all, since it is not possible to separate type definitions from variable declarations, when two variables for the same record structure are needed, the full record construction needs to be repeated. This violates the principle that the type hides the actual representation chosen.

Besides that, the absence of type definitions makes it difficult to group variables that represent the same kind of entities. Although it might well be possible that such variables have the same byte representation. Unfortunately, the converse does not hold: One cannot conclude that whenever two variables share the same byte representation, they must represent the same kind of entity.

In addition to these important problems pertaining to type definitions, COBOL only has limited means to accurately indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, in COBOL, sections or paragraphs that are used as procedures are typeless, and have no explicit parameter declarations.

In our approach, we use types to group variables that represent the same kind of entities. We start with the situation that every variable is of a unique primitive type. We then generate equivalences between these types based on their usage: if variables are compared using some relational operator, we infer that they must belong to the same type; and if an expression is assigned to a variable, the type of the variable must be that of the expression. We also propose a more refined scheme, in which a subtype relation between the types of the expression and the variable is inferred for assignments.

Furthermore, we use a similar approach to infer a minimal set of literal values that should be included in certain types. This information can be used to replace hard wired literal constants in a program with symbolic constants (i.e., replace them by variables that have the same initial value and are not changed in the program). Type information is important for such renovations since the constants for each type might need to be changed independently as a result of maintenance of the program.

Finally, from the minimal set of values of a given type and the usage of variables of that type, we infer whether such a type is an enumeration type: if variables of such a type only get assigned values from this set and there are no computations that might change that value then the type is an enumeration type.

3. Notation

In this paper, we will consider the following primitive types:

Definition 1 *The set T of* primitive types *is defined by the following productions:*

N	$::=$	*Natural numbers*	
I	$::=$	*Set of* identifiers	
B	$::=$	*Set of* byte sorts	
P	$::=$	B^+	(*Pictures of bytes*)
T	$::=$	$\textbf{\textit{elem}}(I,P)$	*(Elementary variable)*
	$\mid$	$\textbf{\textit{rec}}(I,T^+)$	*(Record type)*
	$\mid$	$\textbf{\textit{array}}(I,T,N)$	*(Array type)*

In other words, we distinguish type constructors for elementary data types, for records, and for arrays (with a given length). All types have a name as their first component. The precise choice of the set of byte sorts B can be chosen at will: for our purposes, it consists of the COBOL byte markers such as `X` (character byte), `9` (decimal digit), etc., as occurring in COBOL picture clauses.

We will use T_A to refer to the primitive type that can be derived for a given variable `A` from the data division of a program in which `A` is used.

Below we define the language constructs that are used to describe the type inference rules in the rest of this paper.

Definition 2 *The set S of* syntactic constructs *is defined by the following productions:*

L	$::=$	*Set of literals*	
V	$::=$	I	*(Identifier)*
	$\mid$	$I(E)$	*(Array access)*
E	$::=$	L	*(Literal value)*
	$\mid$	V	*(Variable)*
	$\mid$	E_1 *a-op* E_2	*(Arithmetic operator)*
C	$::=$	E	*(Expression)*
	$\mid$	E_1 *rel-op* E_2	*(Relational operator)*
	$\mid$	$V := E$	*(Assignment)*
S	$::=$	$C+$	*(Syntax)*

The set L corresponds to literals such as numbers and strings, V are variable and array accesses, and E are arithmetic expressions. The set C consists of the set of constructs that

are needed for our purposes: arithmetic expressions, relational expressions, and assignments. It contains only those language constructs that affect the type inference algorithm. The top or start set S is just a collection of constructs from C.

Following [4], we will use so-called *judgements* to express relations between syntactic constructs, and types. Let Γ be a *type environment*, i.e., a mapping from identifiers to types. We will distinguish the following five judgements:

- $\Gamma \vdash \diamond$

  Γ is a well-formed type environment.

- $\Gamma \vdash E : T$

  Expression E is of type T.

- $\Gamma \vdash S : T_1 \equiv T_2$

  An equivalence relation indicating that given construct S, types T_1 and T_2 are the same.

- $\Gamma \vdash S : T_1 \preccurlyeq T_2$

  A partial order indicating that given construct S, type T_1 is a subtype of type T_2.

- $\Gamma \vdash S : L \in T$

  Given construct S, literal L is an element of type T.

The sections to come will include a number of *inference rules* indicating for what particular language constructs these judgements hold.

4. Inference Rules

In this section we describe a method to find an *equivalence* relation between the primitive types within a single module (COBOL program). Later, we will extend this method to system level types and refine the results using subtypes.

4.1. The Data Division

Every variable declared in one of the various sections of the data division of a COBOL program corresponds to a type from the set T of primitive types in a straightforward manner. For simple variables, the `PIC` clause is used to obtain the sequence of byte sorts. `OCCUR` clauses result in arrays, and record definitions yield (nested) record types. To avoid name clashes between fields with the same name coming from different records, variables should be qualified using the full nested record structure. This is a trivial translation that can be done in a preprocessing phase on the incoming COBOL code. As an example, Figure 1 shows the type environment resulting from the COBOL variable declarations shown in Section 2. Observe that every COBOL variable

```
TAB100           ↦  record(TAB100,
                       array(TAB100-POS,
                         elem(TAB100-POS[],X),40)
                       elem(TAB100-FILLED,S9999)),
TAB100-POS       ↦  array(TAB100-POS,
                       elem(TAB100-POS[],X),40),
TAB100-POS[]     ↦  elem(TAB100-POS[],X),
TAB100-FILLED    ↦  elem(TAB100-FILLED,S9999)
```

Figure 1. Type environment derived from COBOL fragment from Section 2.

obtains a unique type. In order to focus the presentation on the most relevant issues, we postpone the treatment of `REDEFINE`s until Section 7.

4.2. Types for Expressions

An arithmetic expression is constructed from variables, constants, and arithmetic operators such as $+, -, *, \ldots$. We derive the type of such an expression by distinguishing the following cases:

1. *Variable access*: If e is a variable, array access, or record field access, its type is the one obtained from analysing the data division.

2. *Arithmetic operators*: Let e be an arithmetic expression of the form e_1 *a-op* e_2. We then infer several types for this expression: every type of e_1 and e_2 is also a type of e.

The rules formalising this are shown in Figure 2. As an example, an expression consisting of just the variable `A` will have one type, T_A, the primitive type derived for `A` from the data division. An expression `A + B` will have two different types: it is both of type T_A as well as of type T_B.

One might think that the type of an expression can be any of the types of the identifiers occurring in that expression. In general, however, this is not the case: an expression can contain an array access, for example `A(I+1) + B(J+1)`, but the type of variables occurring in the access (namely `I` and `J`) are not part of the type of the full expression.

Observe that we take advantage of the fact that in COBOL all arithmetic operators take arguments that must have the same type, namely a numeric type. If COBOL would contain other operators, for example involving both strings and numeric arguments, these operands should not receive the same type. Support for such operators could easily be added to our system by refining the inference rules for operators.

Furthermore, there are no rules for literal expressions (constants): At this stage we are only interested in finding out type information about *variables*.

$$\frac{(\Gamma_1, i \mapsto t, \Gamma_2) \vdash \diamond}{(\Gamma_1, i \mapsto t, \Gamma_2) \vdash i : t} \quad \text{Variable Types}$$

$$\frac{\Gamma \vdash e_1 : t_1}{\Gamma \vdash e_1 \ \textit{a-op} \ e_2 : t_1} \quad \text{A-Op Left}$$

$$\frac{\Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \ \textit{a-op} \ e_2 : t_2} \quad \text{A-Op Right}$$

Figure 2. Rules to infer types for variable access and arithmetic expressions.

4.3. The Procedure Division

Now that we know how to derive types for variables and arithmetic expressions, we can define how to infer relations between the types of the syntactic constructs from S. We distinguish the following cases:

1. *Arithmetic expression*: If $s \in S$ is an arithmetic expression, as we have seen in the previous section, the types of its operands are defined to be equivalent.

2. *Relational operator*: If $s \in S$ is a relational operator, such as $>, <, =, ...$, the types of the operands are defined to be equivalent.

3. *Assignment*: If $s \in S$ is an assignment of the form $v := e$ (recall that this corresponds to COBOL statements such as `MOVE`, `COMPUTE`, `MULTIPLY`, ...), we define that the types of e and v are equivalent.

4. *Array access*: If S contains two constructs that both have array accesses to the same variable, say $v(e_1)$ and $v(e_2)$, then the types of the index expressions are defined to be equivalent. Note that this includes any pair of accesses to the same array v in a program.

The rules formalising these cases are shown in Figure 3. Note that the Array Index rule uses a *context* variable of the form $S[...]$, which represents the source tree S with a subtree left open. We refer to [6] for more details.

As an example, let us infer the type relations for the expression `A + B < D`. The subexpression `A + B` leads, via rule A-Exp, to an equivalence between T_A and T_B. As was shown in the previous section, this subexpression has both type T_A and type T_B. These two types are used when inferring the relations between the types of the complete expression: Following rule Rel-Op, any type of `A + B` is equivalent to the type of `D`. Hence we infer two more equivalences, namely between T_A and T_D as well as between T_B and T_D. Thus, the expression `A + B < D` results in three equavalences: $T_A \equiv T_B$, $T_A \equiv T_D$, and $T_B \equiv T_D$.

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \ \textit{a-op} \ e_2 : t_1 \equiv t_2} \quad \text{A-Exp}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \ \textit{rel-op} \ e_2 : t_1 \equiv t_2} \quad \text{Rel-Op}$$

$$\frac{\Gamma \vdash v : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash v := e : t_1 \equiv t_2} \quad \text{Assignment}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash S[v(e_1)][v(e_2)] : t_1 \equiv t_2} \quad \text{Array Index}$$

Figure 3. Rules to infer equivalences between types, given arithmetic and relational expressions, assignments, and arrays.

4.4. Example

For practical purposes, the most important result of the type inference procedure are the equivalence classes for types. As an example, consider Figure 4, which shows a COBOL fragment manipulating strings. At first sight, the exact relationship between the seven declared variables will be unclear. Applying our type equivalence procedure to this fragment, will infer that `N100`, `TAB100-MAX`, and `TAB100-FILLED` all belong to the same type, due to the statements

```
MOVE TAB100-MAX TO N100.
```

and

```
MOVE N100 TO TAB100-FILLED
```

The equivalence class of these three types corresponds to the index type of the `TAB100-POS` array.

The information that these three variables belong to the same type, can be graphically displayed in an editor (for example by giving them the same colour) which would help the programmer to understand relationships between variables when browsing the program. Moreover, this information can be used when migrating a COBOL application to a typed language. The typical Pascal type for this equivalence class would be a range from 1 to 40 used as array index type.

Other applications of type information in reverse engineering are described in Section 11.

5. System-Level Types

The previous section describes a way of finding sets of equivalent primitive types within a single module (COBOL program). Given the type relations per program, we can infer further type equivalences based on inter-program relations in the following manner:

```
01  N000.
  05 N100           PIC S9(03) COMP-3.
  ...
01  TAB000.
  05 TAB100-NAME-PART
     10 TAB100-POS  PIC  X(01) OCCURS 40.
  05 TAB100-MAX     PIC S9(03) COMP-3 VALUE 40.
  05 TAB100-FILLED  PIC S9(03) COMP-3 VALUE ZERO.
...
R300-COMPOSE-NAME SECTION.
  MOVE TAB100-MAX TO N100.
  MOVE ZERO       TO TAB100-FILLED.

  PERFORM UNTIL N100 EQUAL ZERO
    IF TAB100-POS (N100) EQUAL SPACE
         SUBTRACT 1 FROM N100
    ELSE
         MOVE N100 TO TAB100-FILLED
         MOVE ZERO TO N100
    END-IF
  END-PERFORM.
```

Figure 4. COBOL **fragment for manipulating strings.**

- Make all identifiers unique per program, by qualifying them with the program name.

  Variables declared in copybooks that are included in the data division should be qualified using the copybook's name — in this way variables declared in copybooks included in multiple programs will have the same type.

- In a program call, the actual parameters (the COBOL `USING` clause) are assigned to the formal parameters (the COBOL linkage section), resulting in an inferred equivalence between their types.

- Read and write operations of different variables to the same database result in an inferred equivalence between the variable's types.

A fairly typical call is shown in Figures 5 and 6. Regarding type equivalence, a first observation is that in the call statements, `RAR001-FIXED` is an array of 274 bytes. In other statements (not shown), it is assigned to variables declared as a record also consist of 274 bytes. This is typical COBOL programming style, and done to keep the interface of the call statement simple. Our type inference approach will find equivalences between these byte arrays and full records. This allows us to retrieve the complete (complex) interface that programs actually use for their inter-program communication.

A second observation is that the `L100-ENTITY` parameter is in fact a record. The parameter passing is treated as an assignment from `L100-ENTITY` to `L001-ENTITY`. This, in turn is used to infer a type equivalence between these two records. When looking at the example, however,

```
LINKAGE SECTION.
01 L001-FUNCTION         PIC S9(05) COMP-3.
01 L001-RAR001-FIXED     PIC  X(274).
01 L001-FORMATTED-NAME   PIC  X(46).
01 L001-ENTITY.
   05 L001-ENTITY-NR     PIC S9(11) COMP-3.
   05 L001-ENTITY-TYPE   PIC  X(01).
01 L001-STATUS           PIC S9(05) COMP-3.
```

Figure 5. Linkage section of callee (formal parameters of program `RA36`**).**

```
01 L000.
   05 L100-RA36.
      10 L100-FUNCTION       PIC S9(05) COMP-3.
      10 L100-RAR001-FIXED   PIC X(274).
      10 L100-FORMATTED-NAME PIC X(046).
      10 L100-ENTITY.
         15 L100-ENTITY-NR   PIC S9(11) COMP-3.
         15 L100-ENTITY-TYPE PIC  X(01).
      10 L100-STATUS         PIC S9(05) COMP-3.
...
CALL 'RA36' USING
   L100-FUNCTION  L100-RAR001-FIXED
   L100-FORMATTED-NAME L100-ENTITY L100-STATUS.
```

Figure 6. Call to program `RA36`**, together with actual parameters.**

we immediately see that the fields of these records, namely `ENTITY-NR` and `ENITY-TYPE`, should also be of the same type. This, however, is not inferred by the rules given so far.

Clearly, this is a situation which can occur not only at the inter-program level, but also within programs. What we need is a rule which says that if two structure types are inferred to be equivalent, and if these types have the same structure (without looking at the names), we can infer an additional equivalence between the sub-level types.

To formalise this, we first need the notion of *representation* (i,p,t,n are variables ranging over I,P,T,N, respectively):

Definition 3 *We define* $rep : T \rightarrow P$*, which gives the byte representation of a type inductively by*

$$
\begin{aligned}
rep(\textbf{\textit{elem}}(i,p)) &= p \\
rep(\textbf{\textit{rec}}(i,t_1...t_n)) &= rep(t_1)...rep(t_n) \\
rep(\textbf{\textit{array}}(i,t,n)) &= rep(t)^n
\end{aligned}
$$

The rules in Figure 7 then deal with inferring equivalence for subconstructs. The Fields rule states that if we know that two records are inferred to be equivalent, and if we know that they have exactly the same number of fields, and every two fields have the same representation, then we can infer that the fields must be equivalent as well.

The Arrays rule states that if we know that two arrays are inferred to be equivalent, and if we know that their elements have the same representation, then we can infer that these elements must be equivalent as well.

$$\frac{(j = 1...n)\ \ (\forall_{k:1...n} : rep(f_k) = rep(f'_k)) \quad \Gamma \vdash S : \mathbf{rec}(i, f_1, ..., f_n) \equiv \mathbf{rec}(i', f'_1, ..., f'_n)}{\Gamma \vdash S : f_j \equiv f'_j} \quad \text{Fields}$$

$$\frac{(rep(t) = rep(t')) \quad \Gamma \vdash S : \mathbf{array}(i, t, n) \equiv \mathbf{array}(i', t', n)}{\Gamma \vdash S : t \equiv t'} \quad \text{Arrays}$$

Figure 7. Rules for substructure completion.

$$\frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash S : t_1 \preccurlyeq t_2}{\Gamma \vdash e : t_2} \quad \text{Subsumption}$$

Figure 8. Rule for reasoning with the has-type relation in combination with the subtype relation.

$$\frac{\Gamma \vdash v : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash v := e : t_2 \preccurlyeq t_1} \quad \text{Sub-Assignment}$$

Figure 9. Subtype inference rule for assignments.

6. Assessment of Type Equivalence

The rules provided so far describe how an equivalence relation between primitive types can be derived from a COBOL program. These rules are intuitive, and in general they provide meaningful equivalences. There are, however, a number of problematic situations for which inferring type equivalences is not satisfactory.

First of all, it may be the case that one variable is being used for different purposes in different slices of the program. For example, a variable `TMP` may be assigned the 8-digit variable `PHONE-NR` in one slice, and an 8-digit `DATE` in another. The rules provided so far will infer equivalences for both assignments. By transitivity of equivalence, we then get that `PHONE-NR` and `DATE` are of the same type.

A similar situation can occur in a procedure call. In COBOL, this can happen in a program `CALL`, where the variables in the `USING` clause are the actual parameters, and those in the `LINKAGE SECTION` the formal ones. Alternatively, a `PERFORM` statement can be used, in which case global variables can be used as formal parameters (for an example, see the next section). With the rules given so far, all actual and formal parameters of a procedure will obtain the same type. This may lead to undesirable situations, if the procedure, for example, deals with strings in general, and is given actual parameters of different sorts such as `STREET` or `CITY`.

Another situation that does occur in practice is that a single variable, for example `ZEROES`, is assigned to many different variables during the initialisation phase. Alternatively, one variable, for example `PRINT-LINE`, can receive values from many different variables occurring in a sequence of assignments involving output operations. Again, this will give all these variables the same type.

In all these situations, the inference rules lead to too many equivalences, to which we will refer as *type pollution*. In the next section, we discuss how *subtyping* can be used to address this problem.

7. Subtypes

A type is an indication for a set of permitted values. If the set of permitted values for type T_1 is a subset of the values of type T_2, type T_1 is said to be a *subtype* of T_2, written $T_1 \preccurlyeq T_2$. Subtyping makes a type system more flexible, since an element of a type can be considered also as an element of any of its supertypes, thus allowing an element to be used flexibly in many different contexts [4, Section 6].

The rule for reasoning about type assertions in the presence of subtyping is shown in Figure 8. In addition to that, we need rules to explicitly infer a subtype relationship between two types. Assignments are the natural place for this: If v is assigned an expression e, the type of v should at least contain the values of e, i.e., the type of e is a subtype of the type of v. The rule formalising this is shown in Figure 9. With subtyping this rule should be used instead of the "Assign" rule from Figure 3, which infers a straight type equivalence.

Inferring subtypes has some important practical benefits. Consider, for example, the fragment of Figure 10, which invokes the procedure `R300-COMPOSE-NAME` two times. Since COBOL procedures cannot have parameters, the variable `TAB100-NAME-PART` is used to simulate an input parameter. In the first `PERFORM` statement, it is given the value of `RAR001-INITIALS`, in the second the value of `RAR001-NAME`.

Looking at the names and declarations, one can clearly see that the type of `RAR001-NAME`, a string of length 27 representing a person's last name, and the type of `RAR001-INITIALS`, a string of length 5 representing a person's initials, should be different. However, when inferring type equivalences for assignments, they would become equal, by transitivity via variable `TAB100-NAME-PART`. With subtyping, we do not infer such an equivalence, but infer that they should both have a common supertype, namely the type of `TAB100-NAME-PART` (which has length 40). As described above, similar situations can occur with variables that are used for collecting lines to be printed, temporary variables, etc.

```
01 RAR001-RECORD
   03 RAR001-VAST
       05 RAR001-NAME      PIC X(27).
       05 RAR001-INITIALS  PIC X(05).
      ...
R210-INITIALS SECTION.
   MOVE RAR001-INITIALS TO TAB100-NAME-PART
   PERFORM R300-COMPOSE-NAME
   EXIT.

R230-NAME SECTION.
   MOVE RAR001-NAME TO TAB100-NAME-PART
   PERFORM R300-COMPOSE-NAME
   EXIT.
```

Figure 10. Two calls to a procedure (section) called `R300-COMPOSE-NAME` (see Figure 4). Variable `TAB100-NAME-PART` (in fact a parameter of that section) obtains a supertype, receiving values from both `RAR001-NAME` and `RAR001-INITIALS`.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \text{ } \mathit{rel\text{-}op} \text{ } l : l \in t} \quad \text{Right literal}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash l \text{ } \mathit{rel\text{-}op} \text{ } e : l \in t} \quad \text{Left literal}$$

$$\frac{\Gamma \vdash v : t}{\Gamma \vdash v := l : l \in t} \quad \text{Literal assignment}$$

$$\frac{\Gamma \vdash S : l \in t_1 \quad \Gamma \vdash S : t_1 \equiv t_2}{\Gamma \vdash S : l \in t_2} \quad \text{Equivalent types}$$

$$\frac{\Gamma \vdash S : l \in t_1 \quad \Gamma \vdash S : t_1 \preccurlyeq t_2}{\Gamma \vdash S : l \in t_2} \quad \text{Subtypes}$$

Figure 11. Main rules for inferring minimal literal containment in types.

Using subtyping, `REDEFINE`s can be handled by a simple extension of our type language. In COBOL, `REDEFINE` clauses are used to define data structures that are known as variant records in Pascal (or unions in C); these can be dealt with by adding a *union type* constructor to the set of primitive types T. During analysis of the data division, the type generated for a number of redefined variables is the union type constructed from the types of the individual variables. Furthermore, a rule is added which infers a subtype relation between the components of a union type and the complete union type. The remaining type inference rules stay the same. For more information on union types, we refer to [4].

8. Literal Analysis

A natural extension of our type inference algorithm involves the analysis of literals that occur in a COBOL program. The basic idea is that whenever a variable v is assigned a literal value l, or compared with l, then the type of v should at least contain the literal l. Moreover, whenever we infer that two types must be equivalent, elements contained in one should be contained in the other. Figure 11 formalises these ideas.

An example use of this literal analysis is in the code below:

```
EVALUATE RAR001-NATURE
   WHEN 001 GO TO R180-100
   WHEN 002 GO TO R180-100
   WHEN 003 GO TO R180-100
   WHEN 013 GO TO R180-100
   WHEN OTHER GO TO R180-999
END-EVALUATE.
```

Here `NATURE`, is a number indicating the sort of entity a large record describes. Depending on this sort, different actions are taken. In our case-study of the Mortgage system, our technique was able to find all constants that are used for all variables of type `NATURE`.

Consider the following piece of code:

```
IF RAR001-NATURE EQUAL 8
   IF RAR008-NUMBER  EQUAL 1234 AND
      RAR008-ZIPCODE EQUAL '5678AB'
   ...
```

Here, a selection is made based on a specific address that is included in the code[1]. Our analysis will help to identify such "special values" for a particular type, which provides insight in the nature and actual usage of that type.

The literal type information can also be used to improve the replacement of hard wired literal constants in a program with *symbolic constants*. The algorithm is simple: replace the constants by fresh variables that are initialised to the given literal value and are not changed in the program. For example, the tool set of Sneed [19] has an option called *reassign* for such constant replacements. His approach is to introduce only one symbolic constant which is substituted for all occurrences of the literal constant (e.g. all occurrences of the literal '18' are replaced by `CONST-18` and a new data item '`01 CONST-18 PIC 99 VALUE 18.` is added to the data division).

This approach has the disadvantage that the value of such constants can never be changed during the remaining life time of the reverse engineered program because the literal values that were replaced could have been from different types. For example: consider a program with two literal values '18', one is used to check the number of passengers on a boat, the other is used to check their age. Either of these values might need to be modified during maintenance and by replacing them both by the same symbolic constant `CONST-18` such changes can not be made.

[1] The actual address has been changed to protect the innocent.

The types we infer for literals allow a much more refined renovation: they can be used to replace all occurrences of a literal constant *of a given type* with a symbolic constant for that type. As a result, the constants can be modified independently of each other.

Note that generating names for these constants is no problem, they can either be derived from the name of the type or a fresh prefix can be generated for each new type, similar to the `CONST-18` example above.

The results of the literal type inference described above provide an indication of the minimal set of values that should be included in a given type equivalence class. From this set of values of a given type and the usage of variables of that type, we infer whether such a type is an *enumeration type*, i.e., if variables of such a type only get assigned values from this set and there are no computations that might change that value then the type is an enumeration type.

9. Implementation

We have implemented our ideas in a tool performing type inference on COBOL code. The tool reads COBOL source code and its outputs are the types, typed literal elements, and enumeration types that occur in that code. The architecture of the tool is shown in Figure 12. The boxes represent data, the ellipses represent processes and the arrows depict the flow of data through the system. The solid objects in the figure describe the basic type inference tool. The dashed and dotted objects refer to the extension of our system with literal type detection (dashed) and enumeration type detection (dotted) described in Section 8.

We start with the step *extract primitive types* which finds a set P of primitive types given the data division of the source code. This set is stored in a type environment for the variables of the data division.

We then perform the *derive type relations* step, which combines the primitive types and the usage of variables in the procedure division. The result is a set of relations, which can either be equivalences ($T_1 \equiv T_2$) or partial orderings ($T_1 \preccurlyeq T_2$) for subtyping. For example, the COBOL statement `MOVE A TO B` results in the relation $T_A \preccurlyeq T_B$.

The *type resolution* step infers the types by computing $P/\equiv$: the partition of the set of primitive types that is induced by the derived equivalence relation. Thus the inferred types are the equivalence classes of primitive types modulo $\equiv$. The derived subtyping order $\preccurlyeq$ on primitive types can be used to compute a subtyping order on the inferred types: if $T_1 \preccurlyeq T_2$ then $[T_1]_\equiv \preccurlyeq [T_2]_\equiv$.

Obviously, it is not possible to fully automatically find a meaningful name (or representative) from a set of primitive types. However, we found that it is possible to derive a suggestion for the type name by lexical analysis of the names of the variables that are of a given derived type. Our case study shows that in almost all cases these variables have a common substring. We suggest to use this string as base for the type-name.

Platform We have implemented the architecture using the ASF+SDF Meta-Environment [12, 6, 1]. Furthermore, some pre- and post-processing was done using standard Unix tools like `perl`.

The ASF+SDF Meta-Environment is an interactive development environment for the algebraic specification of formal (programming) languages. It takes a syntax definition of a language and an algebraic specification that describes operations on programs written in that language. From these two, the system generates a programming environment that contains scanners, parsers and syntax-directed editors for the language, and tools that perform the specified operations on programs written in that language [6].

To get an environment for analysing COBOL, we have instantiated the ASF+SDF Meta-Environment with a COBOL grammar [3] and generated *native patterns* and *traversal functions* from this grammar [2, 18]. This gives us a tool that provides a default pass over the full abstract syntax tree of COBOL programs. This default pass can be specialised for particular constructs which allows us to focus only on the COBOL constructs that are important for our problem. In a single traversal of the source code we extract the primitive types, and derive the relations between types. Since the ASF+SDF Meta-Environment uses algebraic specifications, we were able to use the type-inference rules presented in the Figures 2, 3, 7, 8, 9, and 11 almost literally.

10. Case Study

In order to assess the effect of type inference on real life systems, we studied an existing legacy system called Mortgage[2], a COBOL/CICS application of 100,000 lines of code. It consists of an on-line (interactive) part, as well as a batch part, and it is in fact a subsystem of a larger (1 MLOC) system.

We used the implementation of type inference described in the previous section to infer the equivalence classes as well as the subtype relations between them. To enable us to assess the resulting types, we visualised the type relations as directed graphs in which variables are nodes, and arrows and lines represent subtype and equivalence relations respectively. Inspection of these graphs revealed the following issues.

First, assignments are the predominant factor responsible for creating type relations. In other words, COBOL programs contain more `MOVE` statements then (conditional) expressions.

[2] This system was also used as case study in [21, 7].

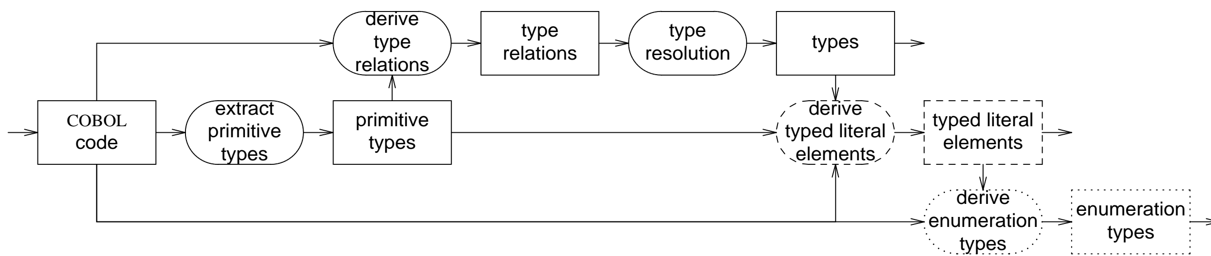

Figure 12. Tool architecture.

Second, the sets of related (via subtyping or equivalence) variables are fairly small. For example, many variables are only once assigned to another variable. We encountered only very few cases in which there were more than 25 different variables involved. This is due to the fact that the types inferred reflect the actual *use* of variables. This gives an interesting comparison with languages that are strongly typed. In such languages, one would declare many different variables of type "int", which may be used for many different purposes. Type *inferencing* finds different types for all these purposes, based on their actual use (see also [17]).

A question of interest is to what extent type pollution (inferring too many equivalences) as discussed in Section 6 is present in Mortgage, and whether the proposed solution, subtyping, is adequate. For most of the variables, pollution is not an issue, i.e., subtyping can be safely replaced by type equivalence. However, all forms of pollution as discussed in Section 6 do occur in Mortgage. Typical cases include the use of a single `MOVE` statement to initialise many different variables, the use of alpha-numeric string variables to represent various types of strings, and the use of sections that use global variables to simulate formal parameters permitting values of different types (different sorts of keys, for example). In all these cases, subtyping provides the proper solution.

Many constants in Mortgage deal with enumeration types. Not all enumeration types in Mortgage contain a consecutive series of numbers: in some cases during maintenance certain numbers may have been removed; in other cases this indicates that a particular program deals with specific enumerated cases only.

Another group of constants occurring in Mortgage deals with *program names*, and are used in statements that invoke other modules, but in which the name of the module is contained in a variable. Our constant analysis helps to identify the possible values of such variables, which is necessary, for example, if one wants to derive the call graph of such programs.

In addition to the qualitative statements listed above, it would be useful to have some quantitative data on types as well, and to collect these for many different systems. We are in the process of collecting data such as the average and maximum of the size of equivalence sets, the number of types related via subtyping, and the number of supertypes per type; the number of equivalence relations divided by the number of subtype relations; and the percentage of declared variables that is never used (which may be up to 10%).

11. Concluding Remarks

Applications Type inference for COBOL systems has many applications. We have presented one, literal analysis, in considerable detail in Section 8. Here we discuss a selection of other applications.

One of the most direct applications of type inferencing is in tool support for year 2000 and Euro conversions. Type inferencing will find a number of types, and matching on names or record structures in these types will classify certain types as "year", "month", "two-digit date", "currency", etc. Indeed several of the published year 2000 solutions [10, 11] search for date-infections by propagating date-seeds via an equivalence relation between variables that is very similar to inferred type equivalence. Moreover, type inferencing can be used to realize the *static date analyser* discussed in [8].

An application using all types rather than just the date-related ones is migrating COBOL systems to a typed language, such as Pascal or C.

One step further is migrating COBOL to an object oriented language. A typical route is to use *subsystem classification techniques* [13] for that purpose, which aim at decomposing a large system into, potentially reusable, components or classes. This is generally by applying a numerical clustering algorithm to group syntactic units based on various interconnection relations. One way is to group procedures based on the types they use. As Lakhotia [13] remarks, however, this technique cannot be used if the source language does not support types. Type inferencing makes these techniques available for the COBOL domain as well.

A rather different potential application of type inferencing is during software maintenance: if types are inferred both before and after the modifications, a presentation of the

difference between the inferred type sets to the programmer may help to detect inconsistencies and potential errors: for example, if the new typing scheme unifies two old types that are perceived as different, the modification made may contain an error.

Related Work A principal source of inspiration to us was Lackwit, a tool for understanding C programs by means of type inference [17]. New in our work is not only the significantly different source language: Also new is the inference of subtyping for assignments, and the use of type inference to classify literals.

The approach of Kawabe *et al.* [11] uses an equivalence relation between variables to deal with the year 2000 problem, which is similar to our inferred type equivalence. They pay a lot of attention to *noise reduction*, but have no solution similar to our subtyping approach. They formulate their work in terms of COBOL, and do not provide a formal type system. They discuss year 2000 as an application.

Chen *et al.* [5] describe a COBOL *variable classification* mechanism. They distinguish a fixed set of categories, such as input/output, constant, local variable etc. They provide a set of rules to infer these automatically, essentially using data flow analysis. Their technique is orthogonal to ours: types we infer can be used in local or global variables, for database output or not, etc.

Newcomb and Kotik [16] 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 considered "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." [16, p. 240]. Our way of type inferencing provides a complementary way of grouping such 01 level records together, and will help to reduce this risk of false positives for small records.

Wegman and Zadeck [20] 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.* [14] 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 [9] identify enumeration types that are modelled using symbolic constants. Their approach is orthogonal to ours since they group constants which are *defined* in the same context whereas we group constants based on their *usage* in the source code.

Future Work We are currently in the process of extending our work in the following ways:

- Inference of input and output parameters for COBOL sections and paragraphs, by means of data flow analysis [15]. This information can then be used to refine the inferred subtype relations.
- Extension of the empirical results, in order to further demonstrate the usefulness of type inferencing, and to assess the validity of the choices made. In particular, we want to apply our technique to other COBOL systems and collect quantitative data on the inferred types.
- We are working on applying type inferencing to component extraction, following [13, 7].
- Extension to new languages, most notably Fortran and IBM 370 assembler.
- Visualisation of the inferred equivalence and subtype relations, the typed literal and enumerations types on the level of COBOL programs as well as visualisation of (the usage of) system-level types in complete COBOL systems.

Contributions In this paper we have proposed a formal system for inferring types from COBOL programs, which we explained by means of a number of real-life COBOL fragments. We formulated rules for inferring type equivalence classes, and we discussed how subtype relations can be inferred to refine the analysis and deal with, for example, variables representing lines to be printed or variables simulating input parameters. We discussed a number of applications, most notably the use of type inference to introduce variables for literals occurring in statements. We have implemented the type inference rules in the ASF+SDF Meta-Environment [12, 6] and successfully applied this tool to a real life, 100,000 lines of code COBOL system.

Acknowledgements We would like to thank Paul Klint (CWI) and Joost Visser (University of Amsterdam) for reading earlier drafts of this paper, and the anonymous WCRE'98 referees who were able to make useful comments and suggestions in spite of the font problem with the version they reviewed.

References

[1] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

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

[3] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In A. Sellink, editor, *Theory and Practice of Algebraic Specifications; ASF+SDF'97*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[4] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[5] 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, Los Alamitos, CA, 1994. IEEE Computer Society.

[6] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[7] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998.

[8] A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In *Proceedings 4th Working Conference on Reverse Engineering; WCRE'97*, pages 124–133. IEEE Computer Society, 1997.

[9] J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In *Third Working Conference on Reverse Engineering; WCRE'96*, pages 227–236. IEEE Computer Society Press, 1996.

[10] 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.

[11] K. Kawabe, A. Matsuo, S. Uehara, and A. Ogawa. Variable classification technique for software maintenance and application to the year 2000 problem. In P. Nesi and F. Lehner, editors, *Conference on Software Maintenance and Reengineering*, pages 44–50. IEEE Computer Society, 1998.

[12] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[13] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

[14] E. Merlo, J. F. Girard, L. Hendren, and R. De Mori. Multi-valued constant propagation analysis for user interface reengineering. *International Journal of Software Engineering and Knowledge Engineering*, 5(1), March 1995.

[15] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In A. Sellink, editor, *Theory and Practice of Algebraic Specifications; ASF+SDF'97*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[16] P. Newcomb and G. Kottik. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering; WCRE'95*, pages 237–249. IEEE Computer Society, 1995.

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

[18] M. P. A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Fifth Working Conference on Reverse Engineering; WCRE'98*. IEEE Computer Society, 1998.

[19] H. Sneed. Architecture and functions of a commercial reengineering workbench. In P. Nesi and F. Lehner, editors, *Conference on Software Maintenance and Reengineering*, pages 2–10. IEEE Computer Society, 1998.

[20] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):18–210, 1991.

[21] T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *4th Working Conference on Reverse Engineering*, pages 24–32. IEEE Computer Society, 1997.