Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Types and Concept Analysis for Legacy Systems

T. Kuipers, L.M.F. Moonen

Software Engineering (SEN)

SEN-R0017 July 31, 2000

Report SEN-R0017
ISSN 1386-369X

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

Types and Concept Analysis for Legacy Systems

Tobias Kuipers

Leon Moonen
http://www.cwi.nl/~{kuipers,leon}/
{kuipers,leon}@cwi.nl

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

ABSTRACT
We combine type inference and concept analysis in order to gain insight into legacy software systems. Type inference for COBOL yields the types for variables and program parameters. These types are used to perform mathematical concept analysis on legacy systems.

We have developed ConceptRefinery, a tool for interactively manipulating concepts. We show how this tools facilitates experiments with concept analysis, and lets reengineers employ their knowedge of the legacy system to refine the results of concept analysis.

1998 ACM Computing Classification System: D.2.2, D.2.3, D.2.7., D.3.4, F.3.1, I.2.2.
Keywords and Phrases: software maintenance, program understanding, program analysis, type inference, concept analysis.
Note: To appear in *Proceedings of the 8th International Workshop on Program Comprehension (IWPC 2000)*, June 4-11, 2000 in Limerick, Ireland.
Note: Work carried out under projects SEN-1.1, *Software Renovation* and SEN-1.5, *Domain-Specific Languages*.

1. INTRODUCTION

Most legacy systems were developed using programming paradigms and languages that lack adequate means for modularization. Consequently, there is little explicit structure for a software engineer to hold on to. This makes effective maintenance or extension of such a system a strenuous task. Furthermore, according to the *Laws of Program Evolution Dynamics*, the structure of a system will decrease by maintenance, unless special care is taken to prevent this [1].

Object orientation is advocated as a way to enhance a system's correctness, robustness, extendibility, and reusability, the key factors affecting software quality [18]. Many organizations consider migration to object oriented platforms in order to tackle maintenance problems. However, such migrations are hindered themselves by the lack of modularization in the legacy code.

A software engineer's job can be relieved by tools that support remodularization of legacy systems, for example by making implicit structure explicitly available. Recovering this information is also a necessary first step in the migration of legacy systems to object orientation: identification of candidate objects in a given legacy system.

The use of concept analysis has been proposed as a technique for deriving (and assessing) the modular structure of legacy software [7, 16, 24]. This is done by deriving a concept lattice from the code based on data usage by procedures or programs. The *structure* of this lattice reveals a modularization that is (implicitly) avaliable in the code.

For many legacy applications written in COBOL, the data stored and processed represent the core of the system. For that reason, many approaches that support identification of objects in legacy code take the data

structures (variables and records) as starting point for candidate classes [5, 11, 19]. Unfortunately, legacy data structures tend to grow over time, and may contain many unrelated fields at the time of migration. Furthermore, in the case of COBOL, there is an additional disadvantage: since COBOL does not allow *type definitions*, there no way to recognize, or treat, groups of variables that fulfill a similar role. We can, however, *infer* types for COBOL automatically, based on an analysis of the *use* of variables [8]. This results in types for variables, program parameters, database records, literal values, and so on, which are used during analysis.

In this paper, we use the derived type information about the legacy system as input to the concept analysis. This way, the analysis is more precise than when we use variables or records as inputs. The concept analysis is used to find candidate classes in the legacy system. External knowledge of the system can be used to influence the concepts that are calculated through ConceptRefinery, a tool we have implemented for this purpose.

All example analyses described are performed on `Mortgage`, a relation administration subsystem of a large mortgage software system currently in production at various banks. It is a 100.000 LOC COBOL system and uses VSAM files for storing data. The `Mortgage` system is described in more detail in [6, 9].

2. TYPE INFERENCE FOR COBOL

COBOL programs consist of a *procedure division*, containing the executable statements, and a *data division*, containing declarations for all variables used. An example containing typical variable declarations is given in Figure 1. Line 6 contains a declaration of variable `STREET`. Its physical layout is described as *picture* `X(18)`, which means "a sequence of 18 characters" (characters are indicated by picture code `X`). Line 18 declares the numerical variable `N100` with picture `9(3)`, which is a sequence of three digits (picture code `9`).

The variable `PERSON` in line 3 is a record variable. Its record structure is indicated by level numbers: the full variable has level `01`, and the subfields `INITIALS`, `NAME`, and `STREET`, are at level `03`. Line 12 declares the array `A00-POS`: it is a single character (picture `X(01)`) occurring 40 times, i.e., an array of length 40.

When we want to reason about types of variables, COBOL variable declarations suffer from a number of problems. First of all, it is not possible to separate *type definitions* from *variable declarations*. As a result, whenever two variables have the same record structure, the complete record construction needs to be repeated.[1] Such practices do not only increase the chance of inconsistencies, they also make it harder to understand the program, since a maintainer has to check and compare all record fields in order to decide that two records indeed have the same structure.

In addition, the absence of type definitions makes it difficult to group variables that are intended to represent the same kind of entities. On the one hand, all such variables will share the same physical representation. on the other hand, 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.

Besides these problems with type *definitions*, COBOL only has limited means to indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, COBOL uses *sections* or *paragraphs* to represent procedures. Neither sections nor paragraphs can have formal parameters, forcing the programmer to use global variables to simulate parameter passing.

To remedy these problems, we have proposed to infer types for COBOL automatically, by analyzing their *use* in the procedure division. In the remainder of this section, we summarize the essentials of COBOL type inferencing: a more complete presentation is given in [8]. First, we describe the *primitive types* that are distinguished. This is followed by a description of the *type relations* that 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.

Primitive Types The following three primitive types are distinguished: (1) *elementary types* such as numeric values or strings; (2) *arrays*; and (3) *records*. Every declared variable gets assigned 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. We qualify variable names with program

[1] In principle the COPY mechanism of COBOL for file inclusion can be used to avoid code duplication here, but in practice there are many cases in which this is not done.

```
1    DATA DIVISION.
2    / variables containing business data.
3     01 PERSON.
4      03 INITIALS     PIC  X(05).
5      03 NAME         PIC  X(27).
6      03 STREET       PIC  X(18).
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 0.
15     ...
16   / other counters declared elsewhere.
17    01 N000.
18     03 N100         PIC S9(03)  COMP-3 VALUE 0.
19     03 N200         PIC S9(03)  COMP-3 VALUE 0.
20
21   PROCEDURE DIVISION.
22   / procedure dealing with initials.
23    R210-INITIAL SECTION.
24     MOVE INITIALS TO A00-NAME-PART.
25     PERFORM R300-COMPOSE-NAME.
26
27   / procedure dealing with last names.
28    R230-NAME SECTION.
29     MOVE NAME TO A00-NAME-PART.
30     PERFORM R300-COMPOSE-NAME.
31
32   / procedure for computing a result based
33   / on the value of the A00-NAME-PART.
34   / Uses A00-FILLED, A00-MAX, and N100
35   / for array indexing.
36    R300-COMPOSE-NAME SECTION.
37     ...
38     PERFORM UNTIL N100 > A00-MAX
39      ...
40      IF A00-FILLED = N100
41       ...
```

Figure 1: Excerpt from one of the COBOL programs analyzed (with some explanatory comments added).

or copybook names to obtain uniqueness at the system level. In the remainder we will use T_A to denote the primitive type of variable A.

Type Equivalence From *expressions* that occur in statements, an *equivalence relation* between primitive types is inferred. We consider three cases: (1) *relational expressions:* such as $v = u$ or $v \leq u$, result in an equivalence between T_v and T_u; (2) *arithmetic expressions:* such as $v + u$ or $v * u$, result in an equivalence between T_v and T_u; (3) *array accesses:* two different accesses to the same array, such as $a[v]$ and $a[u]$, result in an equivalence between T_v and T_u.

When we speak of a *type*, we will generally mean an *equivalence class of primitive types*. For presentation purposes, we will 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.

Subtyping From *assignment statements*, a *subtype relation* between primitive types is inferred. 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.

System-Level Analysis In addition to type relations that are inferred within individual programs, we also infer type relations at the system-wide level: (1) 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). (2) Variables read from or written to the same file or table have equivalent types.

To ensure that a variable that is declared in a copybook gets the same type in all programs that include that copybook, 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]

Literals An extension of our type inference algorithm involves the analysis of literals that occur in a COBOL program. When a literal value *l* is assigned to a variable *v*, we infer that the value *l* must be a permitted value for the type of *v*. Likewise, when *v* and *l* are compared, value *l* is considered to be a permitted value for the type of *v*. Literal analysis infers for each type, a list of values that is permitted for that 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*.

Aggregate Structure Identification When the types of two records are related to each other, types for the fields of those records should be propagated as well. In our first proposal [8], we adopted a rule called *substructure completion*, which infers such type relations for record fields whenever the two records are identical (having the same number of fields, each of the same size). Since then, both Eidorff *et al.* [10] and Ramalingam *et al.* [22] have published an algorithm to split aggregate structures in smaller "atoms", such that types can be propagated through record fields even if the records do not have the same structure.

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 disjunct program slices; simulation of a formal parameter using a global variable to which a range of different variables are assigned; and the use of a `PRINT-LINE` string variable for collecting output from various variables.

The need to avoid pollution is the reason to introduce *subtyping* for assignments, rather than just type equivalences. In [9], we have described a range of experimental data showing the effectiveness of subtyping for dealing with pollution.

Example Figure 1 contains a COBOL fragment illustrating various aspects of type inferencing. It starts with a data division containing the declaration of variables. The second part is a procedure division containing statements from which type relations are inferred.

In line 40, variable `A00-FILLED` is compared to `N100`, which in line 38 is compared to `A00-MAX`. This results in an equivalence class between the primitive types of these three variables. Observe that these three variables are also declared with the same picture (in lines 13, 14, and 18).

In line 29, we infer from the assignment that the type of `NAME` is a *subtype* of the type of `NAME-PART`. From line 24, we infer that `INITIALS` is a subtype of of `NAME-PART` as well, thus making `NAME-PART` the common supertype of the other two. Here the three variables are declared with different pictures, namely strings of different lengths. In fact, `NAME-PART` is a global variable simulating a formal parameter for the `R300-COMPOSE-NAME` (COBOL does not support the declaration of parameters for procedures). Subtyping takes care that the different sorts of actual parameters used still have different types.

3. Concept Analysis

Concept analysis is a mathematical technique that provides a way to identify groupings of *items* that have common *features* [13]. It starts with a *context*: a binary table (relation) indicating the *features* of a given set of *items*. From that table, the analysis builds up so-called *concepts* which are maximal sets of items sharing certain features. The relations between *all* possible concepts in a binary relation can be given using a concise lattice representation: the *concept lattice*.

Recently, the use of concept analysis has been proposed as a technique for analyzing legacy systems [27]. One of the main applications in this context is deriving (and assessing) the modular structure of legacy software [7, 16, 24, 29]. This is done by deriving a concept lattice from the code based on data usage by procedures or programs. The *structure* of this lattice reveals a modularization that is (implicitly) available in the code. In [7], we used concept analysis to find groups of record fields that are related in the application domain, and compared it with cluster analysis.

In the remainder of this section we will explain concept analysis in more detail.

[2] Another (possibly more precise) approach would be to derive a common supertype for all versions that appear in different programs. Our case studies, however, showed no need for such an approach.

Items \ Features	P_1	P_2	P_3	P_4
NAME	×			
TITLE	×			
INITIAL	×			
PREFIX	×			
CITY		×		×
STREET			×	×
NUMBER				×
NUMBER-EXT				×
ZIPCD				×

Table 1: The list of items and their features

	items	features
c0	zipcd number-ext number street city prefix initial title name	
c1	zipcd number-ext number street city	p4
c2	street	p4 p3
c3	city	p4 p2
c4	prefix initial title name	p1
c5		p4 p3 p2 p1

Table 2: All concepts identified for Table 1.

3.1 Basic Notions

We start with a set M of *items*, a set F of *features*,[3] and a *binary relation* (table) $T \subseteq M \times F$ indicating the features possessed by each item. The three tuple (T, M, F) is called the *context* of the concept analysis. In Table 1 the items are the field names, and the features are usage in a given program. We will use this table as example context to explain the analysis.

For a set of items $I \subseteq M$, we can identify the *common features*, written $\sigma(I)$, via:

$$\sigma(I) = \{ f \in F \mid \forall i \in I : (i, f) \in T \}$$

For example, $\sigma(\{\texttt{ZIPCD}, \texttt{STREET}\}) = \{P_4\}$.

Likewise, we define for $F \subseteq F$ the set of *common items*, written $\tau(F)$, as:

$$\tau(F) = \{ i \in M \mid \forall f \in F : (i, f) \in T \}$$

For example, $\tau(\{P_3, P_4\}) = \{\texttt{STREET}\}$.

A *concept* is a pair (I, F) of items and features such that $F = \sigma(I)$ and $I = \tau(F)$. In other words, a concept is a maximal collection of items sharing common features. In our example,

$$(\{\texttt{PREFIX}, \texttt{INITIAL}, \texttt{TITLE}, \texttt{NAME}\}, \{P_1\})$$

is the concept of those items having feature P_1, i.e., the fields used in program P_1. All concepts that can be identified from Table 1 are summarized in Table 2. The items of a concept are called its *extent*, and the features its *intent*.

The concepts of a given table are partially ordered via:

$$(I_1, F_1) \leq (I_2, F_2) \;\Leftrightarrow\; (I_1 \subseteq I_2 \;\Leftrightarrow\; F_2 \subseteq F_1)$$

As an example, for the concepts shown in Table 2, we see that $\bot = c5 \leq c3 \leq c1 \leq c0 = \top$.

[3] The literature generally uses *object* for *item*, and *attribute* for *feature*. In order to avoid confusion with the objects and attributes from object orientation we have changed these names into items and features.

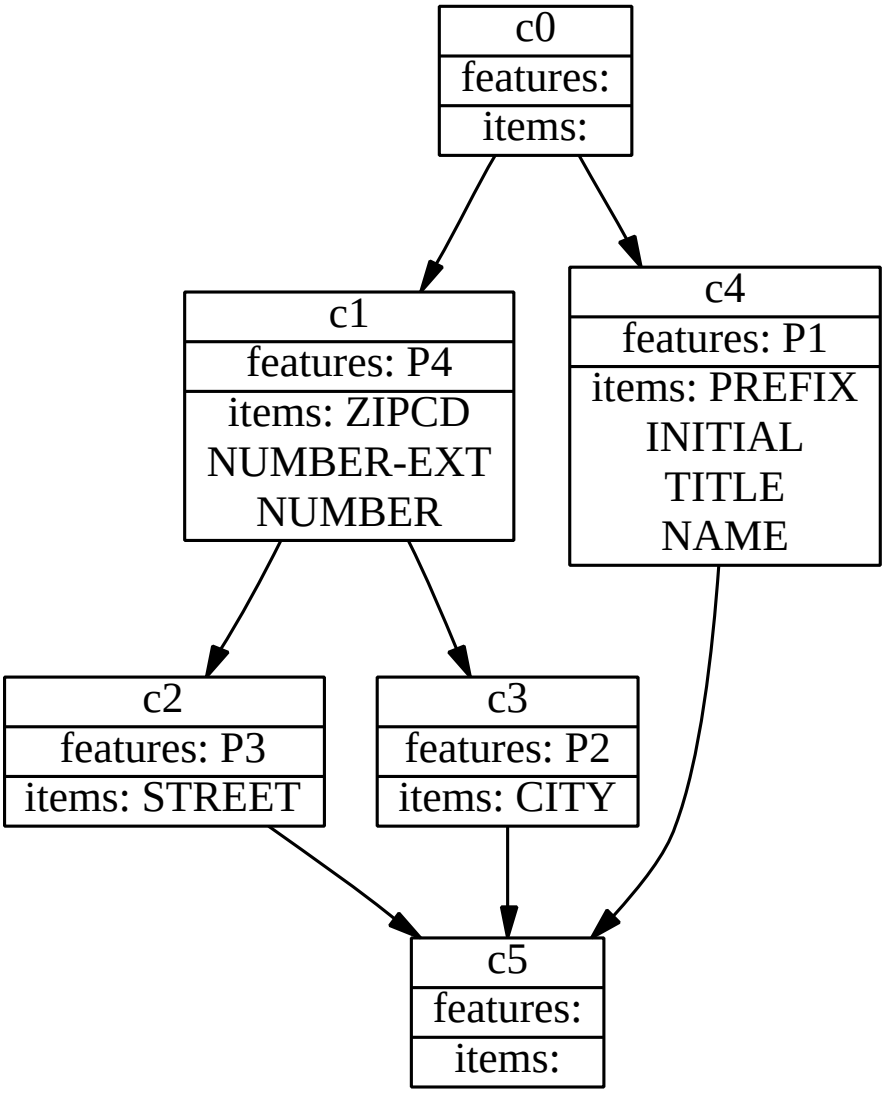


Figure 2: Lattice for the concepts of Table 2.

This partial order allows us to organize all concepts in a *concept lattice*, with *meet* $\wedge$ and *join* $\vee$ defined as

$$\begin{aligned}(I_1,F_1)\wedge(I_2,F_2) &= (I_1\cap I_2,\sigma(I_1\cap I_2)\\ (I_1,F_1)\vee(I_2,F_2) &= (\tau(F_1\cap F_2),F_1\cap F_2)\end{aligned}$$

The visualization of the concept lattice shows all concepts, as well as the relationships between them. For our example, the lattice is shown in Figure 2.

In such visualizations, the nodes only show the "new" items and features per concept. More formally, a node is labeled with an item *i* if that node is the *smallest* concept with *i* in its extent, and it is labeled with a feature f if it is the *largest* concept with f in its intent.

For a thorough study of the foundations of concept analysis we refer the reader to [13].

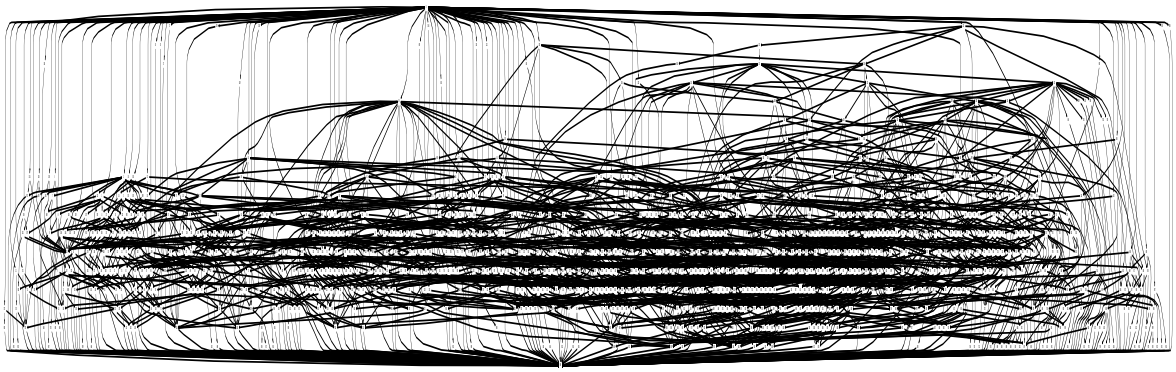


Figure 3: Types as items, program using type as feature

4. Combine Types and Concepts

In [7] concept analysis was used to find structure in a legacy system. The variables of a COBOL system were considered items, the programs features, and the "variable used in program" property as a relation. Table 1 is an example of such a relation, and Figure 2 show the corresponding lattice. This lattice can be seen as a candidate object oriented design of the legacy system. The concepts are individual classes and related concepts can be seen as subclasses or class associations.

The identification of variables in different programs was performed by comparing variable names, and variable declarations. If two variables shared a particular substring they were considered equal. This works well for systems that employ a coding standard which forces similar names for similar variables but fails horribly for systems where variable names are less structured. In this paper this problem is solved by taking the *types* (as described in Section 2) of these variables, and relating them to programs in various ways.

4.1 Data for Concept Analysis

Before describing the concept experiments performed, first the relations derived from the legacy source will be explained. The four extracted relations are varUsage, typeEquiv, transSubtypeOf and formalParam. varUsage is the relation between a program and the variables that are used in that program. typeEquiv is the relation between a type name (the name of a type equivalence class) and a variable that is of this type. transSubtypeOf is the relation between a type and the transitive closure of all its supertypes, i.e. between two types where the second is in the transitive closure of all the supertypes of the first. formalParam is the relation between a program and the types of its formal parameters. An overview of these relations is given in Table 3.

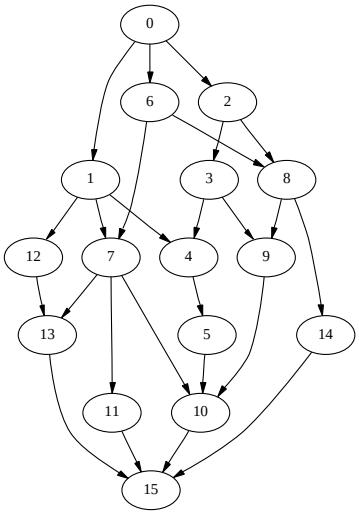
In the remainder of this section the set of all programs, variables, and types in a system will be denoted P, V, and T, respectively.

4.2 Experiments Performed

Type Usage The first experiment performed is exactly the experiment performed in [7], as described earlier. The type usage per program is taken as the context relation, instead of *variable* usage. This results in a lattice where the programs that use exactly the same set of types will end up in the same concept, programs that use

Relation name	Name of relation element	
varUsage	program	variable
typeEquiv	type	variable
transSubtypeOf	sub	super
formalParam	program	type

Table 3: Derived and inferred relations



	items	features
0		
1	1.firstnm[] 1.telb-area 1.telb-nr 1.telp-area 1.telp-nr 1.contact-prs2 1.contact-prs1 1.sex 1.plbirth 1.cvlcd 1.openg	31c 31
2		10c
3		10
4	4.city-bank 4.pay 4.acctnr-bank 4.pay-type 4.paysoc-name 4.acctnr 1.city-bank 1.acctnr-bank 1.pay-type 1.paysoc-name 1.acctnr	
5	4.mortseqnr 4.mortgnr	09 09c
6	1.variable[]	22
7	1.region[] 1.street[] 1.addr-type[] 1.zipcd[] 1.countrycd[] 1.house-ext[] 1.citycd[] 1.birthdt 1.type 1.count	
8		
9	1.mod-dat	
10	4.relnr 1.p-relnr	
11	1.city[] 1.house[]	89c 89
12	1.prefix 1.initl 1.titlcd	36c 36
13	1.fixed 1.name	
14	1.startdt[]	12c
15		

Figure 4: Concepts involving relevant programs

less types will end up in a concept below, and programs that use more types will end up in a concept above that concept.

In order to arrive at the type usage concept lattice the varUsage table is taken as a starting point. For each variable, its type is selected from typeEquiv such that the result is a set of relations $\{(p,t) \in P \times T | (p,v) \in \mathsf{varUsage}, (t,v) \in \mathsf{typeEquiv}\}$. Then the types are considered items, and the programs features and the concept analysis is performed. For the example Mortgage system, the resulting concept lattice is shown in Figure 3. The list of items and features is not shown for (obvious) lack of space.

Filtering This picture may not be as insightful as we might hope. A way to decrease the complexity of this picture is by filtering out data before performing the concept analysis. A selection of *relevant* programs from all programs in a COBOL system can be made as described in [6]. COBOL systems typically contain a number of programs that implement low-level utilities such as file I/O, error handling and memory management. These programs can in general be left out of the analysis, particularly when we are only interested in the general structure of the system.

Filtering out insignificant variables is also possible. Typically, certain records in a COBOL system contain all data that has to do with customers (and therefor is probably relevant) while other records may only be used as temporary storage.

Suppose a list of relevant programs is selected and only the data that originated from a certain set of records is deemed interesting. The first step, filtering out the uninteresting programs, is easy. All tuples from varUsage that have an irrelevant program as their program element are simply ignored. Suppose P_{rel} with $P_{rel} \subseteq P$ is the set of relevant programs which is derived in some way. Then all types that are related to the interesting variables need to be determined. Suppose V_{rel} with $V_{rel} \subseteq V$ is the set of all relevant variables. From the relation typeEquiv all types that are related to a relevant variable are selected. If T_{rel} with $T_{rel} \subseteq T$ is the set of all relevant types: $\{t \in T | (t,v) \in \mathsf{typeEquiv}, v \in V_{rel}\}$ Then the type equivalent variables that are used in the selected relevant programs are selected: $\{(v,p) \in V_{rel} \times P_{rel} | (v',p) \in \mathsf{varUsage}, (t,v') \in \mathsf{typeEquiv}, t \in T_{rel}\}$

The result of the experiments with filtered data are much more comprehensible than those without filtering, basically because there are less concepts to try to understand. Figure 4 shows the concept lattice for the same system as in Figure 3, but with irrelevant programs filtered out according to [6]. The relevant data are the fields of the two records describing the persistent data in the system.

The lattice in Figure 4 contains some unexpected combinations. Concept 7 for instance, contains items that have to do with locations and addresses, but also a birth date. Close inspections reveals that this is not a case of type pollution, but these variables are really used in both program 31(c) (from concept 1) and program 22 (from concept 6). A possible explanation could be that these programs send birthday cards.

It is important to have some way to validate these lattices externally, to perfect the filter set. For our example system, one program implements a utility routine through which a lot of variables are passed, causing one type to contain a remarkable large number of variables. When we filtered out that program, the resulting lattice was much more intuitive.

Parameter Types Experiments have been performed on another concept analysis context; the context that has programs as items and the types of their formal parameter as features. When concept analysis is performed on this data set, all programs that share exactly the same set of parameter-types end up in the same concept. If two programs share some parameter-types, but not all, the shared parameter types will end up in the same concept. These will then form an excellent basis for developing an object oriented view on the system, as the shared types can be seen as the attributes of a class sharing programs as methods.

In its simplest version the items and features for these concepts are computed by just taking formalParam and ignoring the subtype relationship.

As was described in Section 2, the relation between actual parameter types and formal parameter types is inferred as a subtype relation. If the subtype relationship is ignored, then variables can only be identified as having the same type in different programs, when they are "passed" through a copybook. That is, if a variable is included in two different programs from the same copybook, it is considered type equivalent in the two programs. Obviously, this is not the intuition we have when looking at formal parameters, where we would like to know how the types used in the calling program propagate to the called program. Therefor, subtyping *is* considered as type equivalence when looking at parameter types.

The context for parameter type usage per program while considering supertypes as equivalent is derived as follows: $\{(p,v) \in P \times V | (p,t) \in \mathsf{formalParam}, (((t',t) \in \mathsf{transSubtypeOf} \wedge (t',v) \in \mathsf{typeEquiv}) \vee (t,v) \in \mathsf{typeEquiv})\}$.

As described in the previous section, data may be filtered on either relevant programs or relevant data elements. In that case the context is arrived at as follows: $\{(p,v) \in P_{rel} \times V_{rel} | (p,t_1) \in \mathsf{formalParam}, (((t',t) \in \mathsf{transSubtypeOf} \wedge (t',v) \in \mathsf{typeEquiv}) \vee (t,v) \in \mathsf{typeEquiv})\}$ for some externally determined value of P_{rel} and V_{rel}.

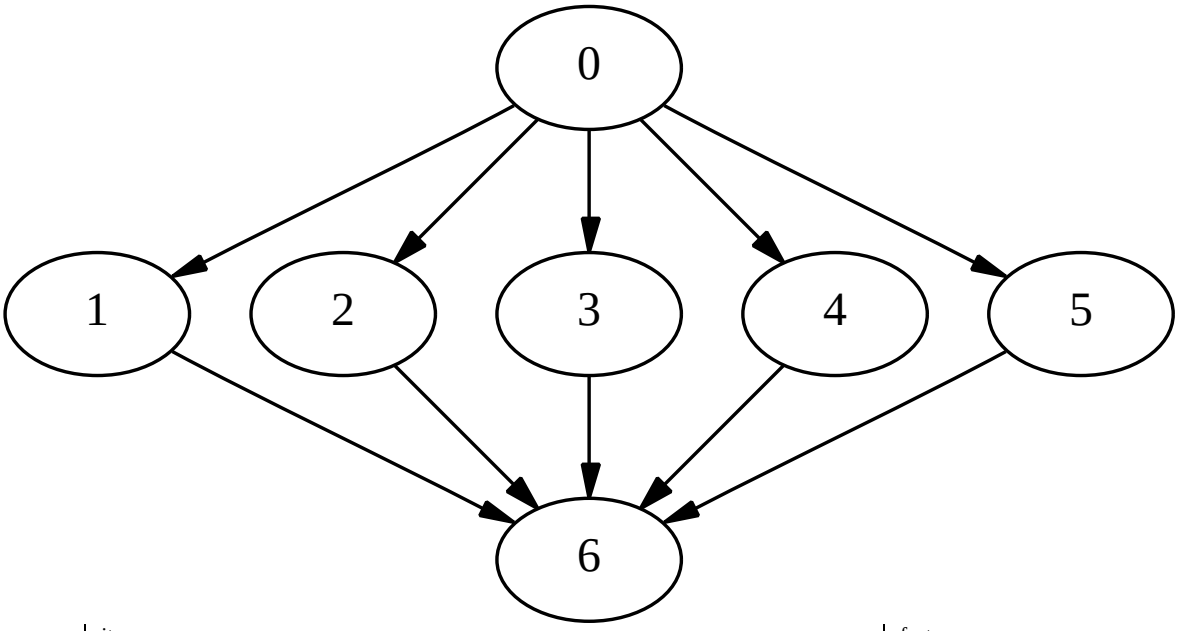
An example of a concept lattice showing program as items and the types they use as formal parameters as features (when supertypes are considered type equivalent) filtered for the same set of relevant variables as Figure 4 is shown in Figure 5.

In this lattice, concept 3 is remarkable, because it contains by far the most programs. This turns out to be caused by the fact that these programs all use "record" as input parameter. Inspection of the source reveals that "record" is a rather large record, and that only some fields of this record are actually used in the programs. It is subject of future work to look at these types of parameters in more detail.

5. Refinement of Concepts

When concept analysis is used for analyzing software systems, there will be a point where a user might want to modify an automatically derived concept lattice. For example, consider the applications of concept analysis to remodularization of legacy systems. A maintainer that performs such a task is likely to have knowledge of the system that is being analyzed. Based on that knowledge, he or she might have certain ideas to improve the modularization indicated by the derived lattice by combining or ignoring certain parts of that lattice.

To facilitate the validation of such ideas, we have developed ConceptRefinery, a tool which allows one to manipulate parts of a concept lattice while maintaining its consistency. ConceptRefinery defines a set of generic structure modifying operations on concept lattices, so its use is not only restricted to the application domain of remodularization or reverse engineering. Figure 6 shows the application of ConceptRefinery on the data of Table 1.



	items	features
0		
1	36	fixed
2	31 10 09	p-relnr
3	36c 35030u 31c 10c 05010r 01410u 01330u 01230u	record
4	41	city[]
5	40	street[]
6		

Figure 5: Programs as items, parameters as features

5.1 Operations on concept lattices

We allow three kinds of operations on concept lattices. The first is to combine certain items or certain features. When we consider the context of the concept analysis, these operations amount to combining certain rows or columns in the table and recomputing the lattice.

The second operation is to ignore certain items or features. When we consider the analysis context, these operations amount to removing certain rows or columns and recomputing the lattice.

The third operation is combining two concepts. This operation has the following rationale: when we consider concepts as class candidates for an object-oriented (re-)design of a system, the standard concept lattice gives us classes where all methods in a class operate on all data in that class. This is a situation that rarely occurs in a real world OO-design and would result a large number of small classes that have a lot of dependencies with other classes. The combination of two concepts allows us to escape from this situation.

On the table underlying the lattice the combination of two concepts can be computed by adding all features of the first concept to the items of the second and vice versa.

5.2 Relation with source

When a concept lattice that was previously derived from a legacy system is manipulated, the relation between that lattice and the code will be weakened:

- Whenever features, items or concepts are combined, the resulting lattice will represent an abstraction of the source system.
- Whenever features or items are ignored, the resulting lattice will represent a part of the source system.

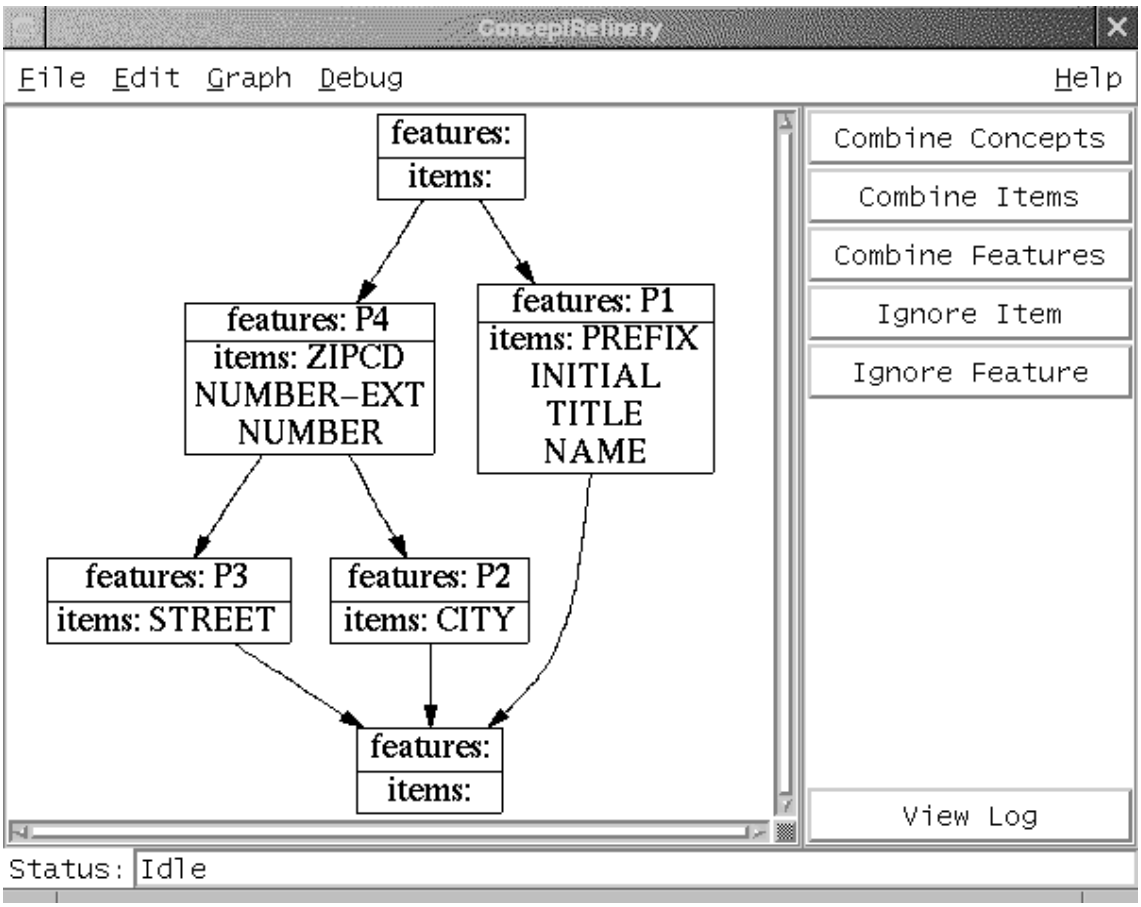


Figure 6: Screendump of ConceptRefinery.

The choice to allow such a weakening of this relation is motivated by the fact that we would rather be able to understand only part of a system than not being able to understand the complete system at all. However, in order for ConceptRefinery to be useful in a real-world maintenance situation, we have to take special care to allow a maintainer to relate the resulting lattice with the one derived directly from the legacy code. This is done by maintaining a concise log of modifications.

6. Implementation

We have developed a prototype toolset to perform concept analysis experiments. An overview of this toolset is shown in Figure 7. The toolset separates source code analysis, computation and presentation. Such a three phase approach makes it easier to adapt to different source languages, to insert specific filters, or to use other ways of presenting the concepts found [6, 8].

In the first phase, a collection 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 that are processed using a Java package which implements the visitor design pattern. The fact extractor is a refinement of this visitor which emits facts at every node of interest (for example, assignments, relational expressions, etc.).

From these facts, we infer types for the variables that are used in the COBOL system. This step uses the COBOL type inferencing tools presented in [9]. The derived and inferred facts are stored in a MySQL relational database [31].

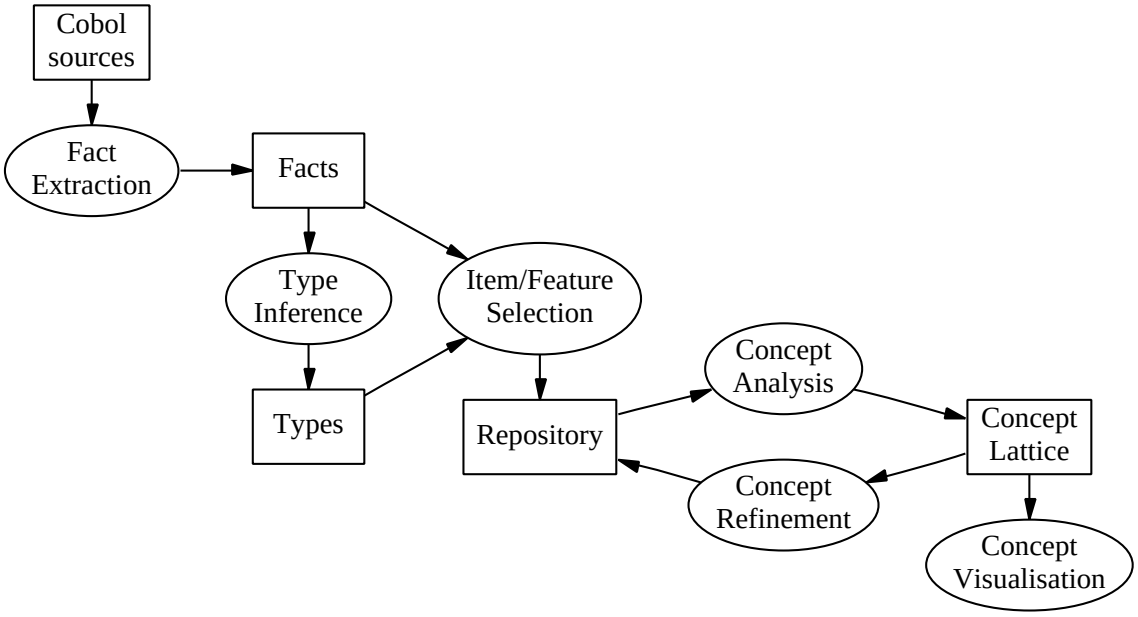


Figure 7: Overview of the toolset.

In the next phase, a selection of the derived types and facts is made. Such a selection is an SQL queries that results in a table describing items and their features. A number of interesting selections were described in Section 4. The results of these selections are stored in a repository. Currently, this is just a file on disk.

In the final phase, the contents of the repository are fed into a concept analysis tool, yielding a concept lattice. We make use of the concept analysis tool that was developed by C. Lindig from the University of Braunschweig.[4] The concept lattice can be visualized using a tool that converts it to input for dot [12], a system for visualizing graphs. The lattices in Figures 2, 5, 3 and 4 were produced this way.

Furthermore, the lattice can be manipulated using ConceptRefinery. This tool allows a user to select items, features or concepts and perform operations on that selection. These operations result in updates of the repository. We distinguish the following manipulations and describe the actions that are carried out on the repository: (1) *combining items or features* is done by merging corresponding columns or rows in the repository; (2) *ignoring items or features* is done by removing corresponding columns or rows in the repository; (3) *combining concepts* is done by adding all features of the first concept to the items of the second and vice versa. The user interface of ConceptRefinery is shown in Figure 6. On the left hand side a visualization of the concept lattice is given. The items, features or concepts that need to be modified can be selected in this lattice. The right hand side shows all available operations. ConceptRefinery is implemented in Tcl/Tk [21] and Tcldot: an extension for Tcl/Tk that incorporates the directed graph facilities of dot into Tcl/Tk and provides a set of commands to control those facilities.

7. Related Work

Several methods have been described for modularizing legacy systems. A typical approach is to identify procedures and global variables in the legacy, and to group these together based on attributes such as use of the same global variable, having the same input parameter types, returning the same output type, etc. [3, 17, 20, 23]. A unifying framework discussing such *subsystem classification techniques* is provided by Lakhotia [15].

Many of these approaches rely on features such as scope rules, return types, and parameter passing, available in languages like Pascal, C, or Fortran. Many data-intensive business programs, however, are written in languages like COBOL that do not have these features. As a consequence, these class extraction approaches have

[4] The tool "`concepts`" is available from `http://www.cs.tu-bs.de/softech/people/lindig/`.

not been applied successfully to COBOL systems [5]. Other class extraction techniques have been developed specifically with languages like COBOL in mind. They take specific characteristics into account, such as the structure of data definitions, or the close connection with databases [5, 11, 19]. The interested reader is referred to [7] for more related work on object identification.

Concept analysis has been proposed as a technique for analyzing legacy systems. Snelting [27, 28] provides an overview of various applications. Applications in this context include reengineering of software configurations [26], deriving and assessing the modular structure of legacy software [16, 24], object identification [7], and reengineering class hierarchies [29].

The extract-query-view approach adopted in our implementation is also used by several other program understanding and architecture extraction tools, such as Ciao [4], Rigi [30], PBS [25], and Dali [14].

New in our work is the addition of the combination of concept analysis and type inferencing to the suite of analysis techniques used by such tools. Our own work on type inferencing started with [8], where we present the basic theory for COBOL type inferencing, and propose the use of subtyping to deal with pollution. In [9], we covered the implementation using Tarski relational algebra, as well as an assessment of the benefits of subtyping for dealing with pollution. Type-based analysis of COBOL, for the purpose of year 2000 analysis, is presented by [10, 22]: both provide a type inference algorithm that splits aggregate structures into smaller units based on assignments between records that cross field boundaries. The interested reader is referred to [8, 9] for more pointers to related work on type inferencing.

8. Concluding remarks

In this paper we have shown that the combination of facts derived from legacy source code, together with types inferenced from those facts, forms a solid base for performing concept analysis to discover structure in legacy systems. This extends and combines our previous work on type inferencing for legacy systems and object identification using concept analysis. We implemented a prototype toolset for performing experiments. From these experiments, we can conclude that the combination of type inference and concept analysis provides more precise results than our previous concept analyses which did not involve types.

The combinations discussed in this paper are the following concept analysis contexts:

1. type usage per program
2. types of parameters per program

The latter analysis appears to be particularly suitable as a starting point for an object oriented redesign of a legacy system.

When performing concept analysis to gain understanding of a legacy system, it proves very helpful if the reengineer is able to manipulate the calculated concepts to match them with his knowledge of the system, or to remove parts he know to be irrelevant. We have implemented ConceptRefinery, a tool that allows a software engineer to consistently perform this kind of modifications while maintaining a relation with both the original calculated concepts, and the legacy source code.

8.1 Future work

We would like to extend ConceptRefinery to propose a grouping of concepts to the human engineer to consider when refining the lattice. To this end, we will to experiment with applying cluster analysis algorithms to the concept lattice.

We have discussed two particular concept analysis contexts in this paper. We would like to see whether we could use the results of one of these concept analyses to improve the results of the other. I.e. to take the concept found by looking at the parameter types of programs and somehow use those to mark relevant and irrelevant concepts from the variable usage analysis.

Acknowledgments The many pleasant discussions we had about this paper with Arie van Deursen are greatly appreciated. We thank Joost Visser for his comments on earlier drafts of this paper.

References

1. L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.
2. 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.
3. G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software—Practice and Experience*, 26(1):25–48, 1996.
4. Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In *International Conference on Software Maintenance; ICSM 95*, pages 66–75. IEEE Computer Society, 1995.
5. A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software*, 44(3):199–211, 1999.
6. 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.
7. A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.
8. A. van Deursen and L. Moonen. Type inference for COBOL systems. In *Proceedings of the fifth Working Conference on Reverse Engineering, WCRE'98*, pages 220–230. IEEE Computer Society, 1998.
9. A. van Deursen and L. Moonen. Understanding COBOL systems using types. In *Proceedings 7th Int. Workshop on Program Comprehension, IWPC'99*, pages 74–83. IEEE Computer Society, 1999.
10. 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 Symp. on Principles of Progr. Languages, POPL'99*. ACM, 1999.
11. H. Fergen, P. Reichelt, and K. P. Schmidt. Bringing objects into COBOL: MOORE - a tool for migration from COBOL85 to object-oriented COBOL. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS 14)*, pages 435–448. Prentice-Hall, 1994.
12. E. R. Gansner, E. Koutsofios, S. North, and K-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

13. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
14. R. Kazman and J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6:107–138, 1999.
15. A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.
16. C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *19th International Conference on Software Engineering, ICSE-19*, pages 349–359. ACM Press, 1997.
17. S. S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *International Conference on Software Maintenance; ICSM'90*, pages 266–271. IEEE Computer Society, 1990.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
19. 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.
20. C. L. Ong and W. T. Tsai. Class and object extraction from imperative code. *Journal of Object-Oriented Programming*, pages 58–68, March–April 1993.
21. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
22. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *26th Symp. on Principles of Progr. Languages, POPL'99*. ACM, 1999.
23. R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering, ICSE-13*, pages 83–92. IEEE, 1991.
24. M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance, ICSM97*. IEEE Computer Society, 1997.
25. S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. M. Cox. Browsing and searching software architectures. In *Int. Conf. on Software Maintenance, ICSM'99*, pages 381–390. IEEE Computer Society, 1999.
26. G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.
27. G. Snelting. Concept analysis — a new framework for program understanding. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998. SIGPLAN Notices 33(7).
28. G. Snelting. Software reengineering based on concept lattices. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00)*. IEEE Computer Society, 2000. To appear.
29. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Foundations of Software Engineering, FSE-6*, pages 99–110. ACM, 1998. SIGSOFT Software Engineering Notes 23(6).
30. K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, 1995.
31. R. J. Yarger, G. Reese, and T. King. *MySQL & mSQL*. O'Reilly, 1999. `http://www.mysql.org/`.