ELSEVIER
Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 60 (2006) 205–220
Science of Computer Programming
www.elsevier.com/locate/scico

Documenting software systems using types[☆]

Arie van Deursen[a,b,*], Leon Moonen[b,a]

[a] CWI, Software Renovation Research Group, SEN-1, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[b] Delft University of Technology, Software Evolution Research Lab, Faculty EEMCS, Mekelweg 4, 2628 CD, Delft, The Netherlands

---

Abstract

We show how hypertext-based program understanding tools can achieve new levels of abstraction by using inferred type information for cases where the subject software system is written in a weakly typed language. We propose TYPEEXPLORER, a tool for browsing COBOL legacy systems based on these types. The paper addresses (1) how types, an invented abstraction, can be presented meaningfully to software re-engineers; (2) the implementation techniques used to construct TYPEEXPLORER; and (3) the use of TYPEEXPLORER for understanding legacy systems, at the level of individual statements as well as at the level of the software architecture — which is illustrated by using TYPEEXPLORER to browse an industrial COBOL system of 100,000 lines of code.

Keywords: Software maintenance; Software exploration; Program understanding; Program analysis; Type inference; Documentation generation; Variable usage; Hypertext

---

1. Introduction

Software immigrants, employees that are added to an existing software project in order to conduct maintenance or development, are faced with the difficult task of understanding an existing software system [39]. Even the original developers of a system generally have a hard time understanding their own code as time between development and maintenance goes by. As a consequence, maintenance tasks become difficult, expensive, and error prone.

To reduce these problems, much research is being invested in the development of tools to assist in program understanding. One line of research focuses on the use of hypertext for program comprehension purposes [3,9,32, 35,38]. Within a hypertext, various layers of abstraction can be integrated, ranging from the system's architecture to the individual statements in the source code. The maintenance engineer can navigate easily between these, using both top-down and bottom-up comprehension strategies, as well as the "opportunistic" combination of these [24,35].

Such a hypertext can be seen as a (special form of) system documentation. Part of it will be hand-written, especially those sections addressing domain-specific issues, the system's requirements, or the rationale behind certain design

---

[☆] Revised version of the paper "Exploring Legacy Systems Using Types". In *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE Computer Society, 2000.

[*] Corresponding author at: Department of Software Engineering, P.O. Box 94179, 1090 GB Amsterdam, The Netherlands.
E-mail addresses: Arie.van.Deursen@cwi.nl (A. van Deursen), Leon.Moonen@acm.org (L. Moonen).
URLs: http://www.cwi.nl/~arie/ (A. van Deursen), http://www.cwi.nl/~leon/ (L. Moonen).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.10.006

decisions. However, documentation at the more technical level should be generated whenever possible, in order to keep it up to date and consistent with the sources at all times.

The fundamental problem with documentation generation (and in fact, the key challenge of reverse engineering) is to arrive at non-trivial levels of abstraction, going beyond just cross-referencing information and source code browsing. Our research aims at achieving such a level of abstraction by looking at the *types* that are used in a software system.

For typed languages, such as Java, C, and Pascal, using types for program comprehension is relatively straightforward: types are explicit, and can help to determine interfaces, function signatures, permitted values for certain variables, etc. Many of the existing software systems, however, are written in older languages with very weak type systems. In particular COBOL, the language in which at least 30% of the world's software is written, does not offer the possibility of type definitions. The question we ask ourselves is whether types nevertheless can help in understanding such COBOL systems.

The solution we propose is to *infer* types for COBOL automatically, based on an analysis of the *use* of variables [10]. This results in types for variables, program parameters, database records, literal values, and so on, which can be used to understand the relationships between, e.g., programs, copybooks, databases, screens.

In earlier work, we presented an algorithm and toolset for determining types in COBOL systems [10,11]. The current paper addresses the problems involved in integrating inferred types into hypertext-based program understanding tools. In particular, we will be concerned with the following three questions:

Presentation Types are an abstraction not directly present in the (legacy) system — types do not exist in the code, but must be inferred first. How do we present this abstraction in such a way that it provides an understandable, meaningful and useful view on a legacy system?

Implementation How do we implement tools to obtain this presentation?

Use What maintenance or program understanding questions can be answered using such a presentation, not only at the individual module level, but also at the architectural level?

We will explain how we dealt with these issues while constructing TYPEEXPLORER, a tool for exploring COBOL systems using types. In Section 2 we give an overview of related work. In Section 3 we discuss the theory of type inferencing for COBOL. We cover the design of the hypertext structure used by TYPEEXPLORER in Section 4, and the techniques that were used for implementation in Section 5. We then discuss the usefulness of TYPEEXPLORER for various program understanding tasks and describe its application in a 100,000 lines of code COBOL case study in Section 6. Finally, we summarize our contributions, and list possibilities for future work in Section 7 .

2. Related work

Our work on TYPEEXPLORER is related to three areas of research: (1) the exploration of legacy system using reverse engineering techniques; (2) the use of types for analyzing legacy systems; and (3) specific techniques used for analyzing Cobol legacy systems. We discuss related work in each of these areas.

2.1. Software exploration

Software evolution involves the process of keeping a software system in sync with the ever-changing needs of the system's users and environment. An unfortunate side-effect of evolution is that it often causes the knowledge about a system to degrade, which in turn impedes further evolution. *Software exploration* tools help software engineers navigating through and understanding evolving software systems. In [27], Moonen investigates techniques and tools that support the exploration of a software system and improve its legibility. He examines the analogy with urban exploration and presents techniques for the extraction, abstraction, and presentation of information needed for understanding software.

Exploration support can be seen as a form of (re)documentation. Chikofski and Cross define *redocumentation* as the creation of a semantically equivalent representation of a software system within the same level of abstraction. Common tools include pretty printers, diagram generators, and cross-reference listing generators [7]. Landis et al. discuss various documentation methodologies, such as Nassi Schneiderman charts, flow charts and Jackson diagrams [23]. Today, a growing body of literature on automated program documentation exists [3,9,12,14,19,32,34,35,38,42]. Of these, Brown discusses a tool that automatically creates links between program analysis data and hypertext documentation [3]. CHIME is a generator of tools that automatically insert certain links in source code elements [12]. PAS is a system that can be used to incrementally add *partitioned annotations of software* [35]. Documentu [32] follows a tag-based method inspired by literate programming also used in Javadoc.

Wong et al. emphasize *structural redocumentation*, which, as opposed to *documentation in-the-small*, deals with understanding architectural aspects of software [42]. They use Rigi for the extraction, querying, and presentation, using a *graph editor* for manipulating program representations. Several *views* of the legacy system can be browsed using the editor. Our approach also focuses on the structural aspects of documentation. Rather than using a dedicated graph editor, we use standard HTML browsers for viewing the documentation. Furthermore, we determined the required views in advance, via discussion with the team of maintenance programmers.

The software bookshelf [14] is an IBM initiative building upon the Rigi experience. In this metaphor, three roles are distinguished: the *builder* constructs (extraction) tools; the *librarian* populates repository with information using the building tools or other (manual) ways, and the *patron* is the end user of the bookshelf.

DOCGEN is a tool for generating hyperlinked visual and textual documentation from COBOL and batch job sources [9]. Distinguishing characteristics of DOCGEN include extraction based on *island grammars* [26,28] rather than full parsing, emphasis on industrial application,[1] and integration of various abstraction layers, ranging from source code up to system architecture. We will see later how the type information derived by TYPEEXPLORER can be integrated with documentation that is generated by DOCGEN.

2.2. *Type-based analysis of legacy systems*

Our own work on type inferencing started with [10], where we present the basic theory for COBOL type inferencing. In [11], we describe an implementation using Tarski relational algebra. Moreover, we carried out a detailed assessment of the benefits of using subtyping to deal with the problem of *pollution* (inferring too many type equivalences). The application of type inferencing (in combination with concept analysis) to the identification of objects in legacy systems is discussed by [22]. In this paper, we do not extend the theory of type inferencing: instead we explain how inferred types can be presented using hypertext, and used to understand COBOL systems at various levels of abstraction.

Closest in aims to the integration of type analysis and program understanding is Lackwit [31], a tool for analyzing C programs using type inferencing. Lackwit allows one to ask queries like “Which functions could directly access the representation of component X of variable Y?”. New in our work is not only the significantly different source language, but also the inference of subtyping for assignments, and the use of type inference to classify literals. Other work on type inference for C includes that of Siff and Reps [37], who use inferred types to generalize C functions to C++ function templates and that of Chandra and Reps who discuss “physical type checking of C”, which is a stronger form of type checking for type casts involving pointers to structures [5].

Wegman and Zadeck [40] describe a method for detecting 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. [25] 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 [16] 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” (i.e., close to each other in the program text) whereas we group constants based on their *usage* in the source code.

Other applications of type inferencing include the analysis of Fortran programs in order to find new type signatures for subroutines [41]. Palsberg presents a more detailed survey of the literature on type-based analysis and its applications [33].

2.3. COBOL *program analysis*

In the area of analyzing COBOL programs, our type inferencing approach is related to various tools for the analysis and correction of the year 2000 problem where date *seeds* are tracked through the statements in a program [17,29,20].

---

[1] Services using DOCGEN are available via the *Software Improvement Group*, http://www.software-improvers.com.

The approach of Kawabe et al. [20] 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. [6] describe a (semi-)automatic COBOL *variable classification* mechanism. They distinguish a fixed set of categories, such as input/output, constant, local variable, and each variable is placed into one or more of these classes. They provide a set of rules to infer this classification automatically, essentially using data flow analysis. Their technique is orthogonal to ours: types we infer can be used for both local or global variables, for variables that are used for databases access and for those that are not, etc.

Newcomb and Kotik [30] 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” [30, p. 240]. Our way of type inferencing may help to reduce this risk, as it provides a complementary way of grouping such 01 level records together based on *usage*.

An even more detailed solution to this problem is discussed by [36,13]. They propagate type information through the elements of aggregate data structures, such as arrays or records. For example, when two entire records are moved, types are propagated through the individual fields. Moreover, these moves may even cross field boundaries if the two records differ in record layout, or if records are aliased using COBOL’s redefine statement. The authors provide an algorithm that finds a minimal splitting of all aggregate structures such that types can be correctly propagated for the resulting “atoms”. This decomposition is based on the access patterns and COBOL picture clauses specific to the given program. In our earlier paper [10], we proposed a weaker method using an inference rule called *substructure completion*, which just ensures that type equivalences between structurally equivalent aggregates are propagated to the components. The aggregate analysis of [36,13] is orthogonal to our approach and can be combined with our type inferencing approach to further improve the accuracy.

3. 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.

Some typical variable declarations are shown in lines 1 to 20 of Fig. 1. In line 6, a variable `STREET` is declared. Its physical layout is described using the *picture* `X(18)`, which means “a sequence of 18 characters” (characters are indicated by picture code `X`). A numerical variable is defined in line 18, where the variable `N100` has picture `9(3)`, which is a sequence of three digits (picture code `9`).

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

From the perspective of types, such COBOL variable declarations suffer from a number of problems. First of all, it is not possible to separate type definitions from variable declarations. Consequently, when two variables for the same record structure are needed, the full record construction needs to be repeated.[2] This not only increases the chances of inconsistencies, but also makes it harder to understand the program, as the maintainer has to check and compare all record fields in order to decide that two records indeed have the same structure.

Furthermore, the absence of type definitions makes it difficult to group variables that are intended to represent the same kind of entities. Clearly, all such variables will share the same physical 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.

Besides these problems regarding 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*

---

[2] 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.

to represent procedures. Neither sections nor paragraphs can have formal parameters, forcing the programmer to use global variables for parameter passing.

In [10], we propose a method for inferring types for COBOL to remedy these problems. This method automatically infers types for COBOL variables by analyzing the *use* of these variables in the procedure division. The remainder of this section summarizes the essentials of COBOL type inferencing.

3.1. Primitive types

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

3.2. Type equivalence

From *expressions* occurring in statements, an *equivalence relation* between primitive types is inferred. We distinguish 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* to the same array such as $a[v]$ and $a[u]$ result in an equivalence between T_v and T_u.

We will generally speak of a *type*, meaning an *equivalence class of primitive types*. We will give names to types based on the names of the variables that are of that type. For example, the type of a variable with the name `L100-DESCRIPTION` will be called `DESCRIPTION`-type.

3.3. Subtyping

From *assignment statements* a *subtype relation* between primitive types is inferred. From the assignment $v := u$ we conclude that T_u is *subtype* of T_v, i.e., v can hold at least all the values u can hold.

3.4. Union types

From COBOL *redefine clauses*, a *union type* relation between primitive types is inferred. When an entry v in the data division redefines an entry u, we conclude that T_v and T_u are part of the same *union type*.

3.5. System-level analysis

The type relations described before are derived at the program level. We also derive a number of type relations at the system-wide level: (1) *program parameters:* the types of the actual parameters of a program call (listed in the COBOL `USING` clause) are *subtypes* of the formal parameters (listed in the COBOL `LINKAGE` section), (2) *file/table access:* variables read from or written to the same file or table have *equivalent* types, and (3) *copybooks:* a variable which is declared in a copybook gets the same type in all the programs that include this copybook.

3.6. Literals

Our type inference algorithm can easily be extended with analysis of literals in a COBOL program. Whenever a literal value l is assigned to, or compared with a variable v, we infer that l is a *permitted value* for the type of v. 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*.

```
DATA DIVISION.
/ variables containing business data.
 01 PERSON.
  03 INITIALS     PIC  X(05).
  03 NAME         PIC  X(27).
  03 STREET       PIC  X(18).
  ...
/ variables containing char array of length 40,
/ as well as several counters.
 01 TAB000.
  03 A00-NAME-PART.
   05 A00-POS     PIC  X(01)  OCCURS 40.
  03 A00-MAX      PIC S9(03)  COMP-3 VALUE 40.
  03 A00-FILLED   PIC S9(03)  COMP-3 VALUE 0.
  ...
/ other counters declared elsewhere.
 01 N000.
  03 N100         PIC S9(03)  COMP-3 VALUE 0.
  03 N200         PIC S9(03)  COMP-3 VALUE 0.

PROCEDURE DIVISION.
/ procedure dealing with initials.
 R210-VOORLT SECTION.
  MOVE INITIALS TO A00-NAME-PART.
  PERFORM R300-COMPOSE-NAME.

/ procedure dealing with last names.
 R230-NAME SECTION.
  MOVE NAME TO A00-NAME-PART.
  PERFORM R300-COMPOSE-NAME.

/ procedure for computing a result based
/ on the value of the A00-NAME-PART.
/ Uses A00-FILLED, A00-MAX, and N100
/ for array indexing.
 R300-COMPOSE-NAME SECTION.
  ...
  PERFORM UNTIL N100 > A00-MAX
   ...
   IF A00-FILLED = N100
    ...
```

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

3.7. *Aggregate structure identification*

Whenever the types of two records are related to each other, types for the individual fields should be propagated as well. In [10], we adopted a rule called *substructure completion*, which infers such type relations for record fields whenever the two record structures are identical (having the same number of fields, each of the same size). Since then, both Eidorff et al. [13] and Ramalingam et al. [36] have published an algorithm which splits aggregate structures into smaller "atoms", such that types can be propagated through record fields even if the records do not have the same structure.

3.8. *Pollution*

We speak of *type pollution* when the types of two variables are inferred to be equivalent but would have been given different types if a typed language was used. Typical situations in which pollution occurs include the use of a single variable for different purposes in different program slices; the use of a global variable for parameter passing; and the use of a `PRINT-LINE` string variable for collecting values from various variables.

Inference of *subtypes* for assignments, rather than just type equivalences was introduced to avoid pollution. In [11], we describe a range of experimental data showing the effectiveness of subtyping for dealing with pollution.

3.9. *Example*

Fig. 1 contains a COBOL fragment illustrating various aspects of type inferencing. The first half contains the declarations of variables, the second half the actual 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 `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 acting as 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.

4. Presenting types in hypertext

This section describes how types can be presented in a hypertext to support program understanding. We cover the challenges that need to be addressed, as well as the solutions that we adopted in TYPEEXPLORER.

4.1. Challenges

4.1.1. Inventing a name for a type

Recall from Section 3 that a *type* is an equivalence class of *primitive types*, and that each primitive type directly corresponds to a variable declaration. For example, in Fig. 1, we inferred an equivalence between the three variables `A00-FILLED`, `N100`, and `A00-MAX`. In TYPEEXPLORER, we need to invent names for these equivalence classes. One way is to pick an arbitrary element, and make that the name of the type.

An alternative is to try to distill meaningful names from the variable names involved, by determining the *words* occurring in them. Such words can be found by splitting the variable names based on special characters ('-', '_', etc.) or lexical properties (e.g., caseChange). The actual splitting should be a parameter of the analysis since it is influenced by the coding style that is used in a system. Candidate names of a given type can then be based on the frequency of words that occur in names of variable of that type. Since we want these names to be as descriptive as possible, one also needs to consider all combinations of words that occur in variable names. As an example, for the `A00-NAME-PART` variable, we want to see not only the words `NAME` and `PART`, but also the word `NAME-PART`.

4.1.2. Duality of subtyping

Our type inferencing algorithm uses subtyping to avoid pollution. In some cases, though, there would be no pollution even if plain equivalences between types were to be used. One could even argue that using subtyping in those cases obscures understanding since it creates additional levels of indirection between types that would otherwise be considered equivalent. Thus, we are faced with the problem that for some types subtyping are necessary to avoid pollution, whereas for others subtyping should actually have been type equivalence.

Our solution is to include an additional abstraction layer, the *type cluster*. A cluster consists of all types that have an equivalence or subtype relation to each other (effectively regarding the subtyping relation as an equivalence relation). If the TYPEEXPLORER user is not interested in the subtyping details of a particular type, they can move up to the type cluster level.

4.1.3. Static/dynamic hypertext

We distinguish two versions of the hypertext. In the *off-line* (static) version all pages are generated in advance. The advantage of this version is portability; the complete documentation can be reproduced on a CD, taken anywhere, and browsed on almost any computer system (only requiring a standard web browser). Disadvantages are the static nature of the hypertext and the lack of dynamic querying.

In the *on-line* (dynamic) version the pages are generated on the fly based on queries on a database attached to the links clicked on. When the users makes updates, for example to improve the name of a type, such changes are propagated immediately. Advantages of this approach are the ability to generate hypertext based on queries by the user and the immediate response to changes. Disadvantages are the lack of portability and relatively high technical requirements on the computer system that is used for browsing.

Element	Available Information
annotation	hand-written description of this type
structure	the picture or record declaration(s) of variables of type τ
values	all literal values found for τ.
type graph	visualization of subtypes and supertypes of τ.
usage	links to source code lines where a variable or literal of τ is used.
parents	links to records with fields of type τ.
programs	links to programs that use τ.
copybooks	links to copybooks that use τ.
words	domain concepts extracted from names of variables of type τ (based on heuristics).
type name	suggestion for name of this type based on these domain concepts.

Fig. 2. Information presented for a type τ.

4.1.4. What are good starting points for browsing?

To be flexible and generic enough to handle the multitude of program understanding tasks, the resulting hypertext should support multiple starting points. Example starting points are persistent data stores (which are likely to contain data types that are closest to the business logic) program signatures, types matching a given name pattern (with an effect similar to seeding in year 2000 tools), or a specific variable directly in the source code. In the *off-line* version, the top-level index pages should easily lead to such starting points. In the *on-line* version, more flexibility is provided, as queries can be used to arrive at the desired HTML page.

4.1.5. Annotations

For programs, it is possible in some cases to derive a textual description explaining their behavior based on the comment prologue [9]. Since types are abstractions that are not directly present in one particular place in the source code, it is not possible to find meaningful texts explaining types automatically. Therefore, we give maintainers the ability to add (optional) annotations by hand. In practice, such a feature will be used mostly for types that play a significant role in the system. Furthermore, there can be a special annotation allowing a maintainer to improve the name given to a type. In the on-line version, annotations can be added on the fly, and have immediate effect; in the off-line mode, annotations are incorporated after regeneration.

4.2. Information available per type

The most important pages in TYPEEXPLORER are those that explain an inferred type, so we will first discuss the contents of these pages. An overview of the various page elements is shown in Fig. 2.

4.2.1. Pictures

The declared COBOL *pictures* of primitive types provide information about the bytes occupied and the intended use (number, character, . . .). In most cases, all primitive types in an equivalence class will have the same picture. If the pictures are different, this means that the COBOL code using variables of this type relies on coercions, which may indicate bad programming style or potential programming errors.

4.2.2. Records

If the primitive types of a type τ are all *records*, the commonest case is that where all variables in this type are declared with the same number of fields, each of the same length. In this case, our rule of substructure completion

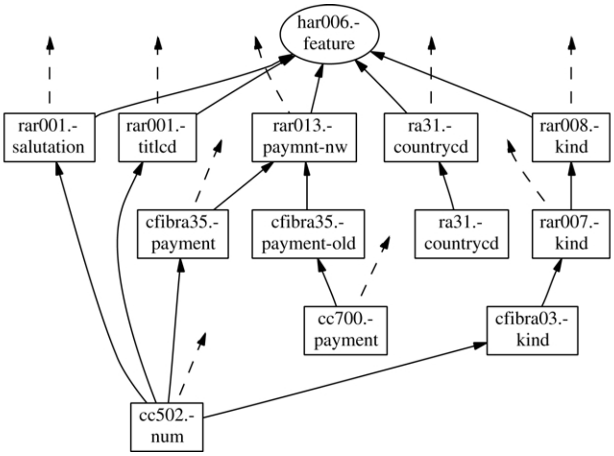


Fig. 3. Example type graph.

will infer equivalences between these field types; if they are of different shape, *aggregate structure identification* [13, 36] can be used to find subfields that are small enough to unify the various records in τ. Thus, although the primitive records in τ may be of different shape, we infer one record type with the smallest necessary fields for τ, and list the fields of τ in its page.

4.2.3. *Literals*

The inferred literals provide information about the sort of values that are permitted for this type. Moreover, they show which literal values are actually used in the system analyzed. Since a supertype τ can hold at least the values of all its subtypes, we also list the literals in all subtypes of τ.

4.2.4. *Usage*

In addition to structural information about a type τ, we can provide data on its *usage*. We include links to source code lines in which a variable of type τ is used, as well to those lines in which a literal of type τ is used. Moreover, we include links to the documentation of all programs and copybooks that use the type.

For types used as *fields* in other records, we include a link to each of the parent records.

4.2.5. *Type graphs*

An inferred type τ can be related to other types via subtype (or supertype) relationships. As part of the documentation generated for a type τ, we display all subtypes and supertypes of τ in a *type graph*. An example type graph is shown in Fig. 3. This figure comes from the actual type web derived for the case study described in Section 6.[3]

The nodes in the graph are types: the text in a node is the name chosen for a type. This name is obtained by picking one of its primitive types as representative. Clicking on the nodes brings up the page for the type clicked on. The type τ itself is shown in an ellipse. In Fig. 3 it has name *har006.feature*. An arrow from τ_1 to τ_2 means that τ_1 is a subtype of τ_2.

A number of observations can be made from this graph. First of all, the subtype relationship on types closely corresponds to the assignment relationship between variables. Thus, one can read an arrow $\tau_1 \rightarrow \tau_2$ also as: "variables of type τ_1 are assigned to variables of type τ_2."

Second, within the graph, one can recognize groups of related types: in Fig. 3, examples are the three *kind* types on the right, or the four *payment* types in the middle.

Third, the type selected, *har006.feature*, happens to be a supertype of several other types. Thus, *har006.feature* can accept values of several different subtypes, dealing with various sorts of numbers, such as *country codes*, *title codes*. Such a type with several different subtypes is typically the *input* parameter of a procedure or program, where each

---

[3] For presentation purposes, we have translated the variable names from Dutch into English in the figure.

incoming edge corresponds to the subtype of an actual parameter. If we were to infer not subtypes, but equivalences instead, all these types would become the same (via *har006.feature*).

Fourth, some types have dashed outgoing (or incoming) edges. This means that these types have other supertypes (subtypes), which are, however, not subtypes or supertypes of the type selected, *har006.feature*. An example is the leftmost *salutation* type. Its outgoing edge to *har006.feature* means that *salutations* are moved to *features*; its dashed outgoing edge means that *salutations* are moved elsewhere as well.

Fifth, the type *c502.num* only has outgoing edges. This typically means that *c502.num* is the output parameter of procedure or section. Furthermore, the fact that *c502.num* has no incoming edges means that there are no assignments from other types into *c502.num*. This can mean one of three things for variables of type *c502.num*:

(1) They never get a value within the programs analyzed, only in external libraries.
(2) They do get a value, but only from variables also of type *c502.num*.
(3) They do get a value, yet not as a scalar value, but viewed as an aggregate. This, is in fact the case for *c502.num*, which is filled as an array, digit by digit.

In short, type graphs can be used to show a number of interesting properties regarding types and variables. For the case studies conducted, most of the type graphs are reasonably small and understandable. The dashed arrows are an important tool for keeping them small: if we were to expand all dashed arrows transitively, the type graph for *har006.feature* would become several hundreds of nodes larger.

4.2.6. *Type metrics*

Types can furthermore be characterized by metrics. Example metrics for an inferred type τ include the number of literal values occurring in τ, the number of variables in τ, or the number of record types that τ is involved in as a field. Concerning the subtype relationship, the number of subtypes or supertypes of τ, and the size of the type cluster that τ belongs to are of interest. Relatively large values for these metrics point to heavily used types that will be difficult to modify. In [11] we have analyzed these metrics for a given system in order to evaluate the effectiveness of subtyping for dealing with pollution.

4.3. *Types in programs and copybooks*

To present types in the context of programs and copybooks, we integrate them with system documentation that is automatically derived from legacy sources using DocGen. This hypertext describes the system at various levels of detail. At the program level we find copybooks that are included, flatfiles read or written, database tables that are updated or selected, screens that are presented to the user, etc. Zooming in from the program level, we arrive at the level of the individual sections, copybooks, and ultimately the full source. Zooming out, we arrive at the subsystem level that groups collections of batch (JCL) jobs, programs, copybooks, etc. corresponding to subsystem decompositions as used by the maintenance team (usually visible in naming conventions or directory structure) or as found by automatic clustering techniques. A more detailed account can be found in [9].

One obvious (and straightforward) method of integration is to provide links from variables and literals occurring in the source code to their inferred type pages.

Moreover, we derive *signatures* for modules that are called or can be called by others. Such a signature documents the intended use of a module. It gives the types of the *formal* parameters, which are derived from the variables declared in the COBOL linkage section. This does not only provide information about the formal parameters: the type graph of each of the formal parameters also contains subtypes for all actual parameters used in the system analyzed.

Second, we obtain types for the records that are written to or read from persistent data stores such as files or database tables. In particular in COBOL systems, such records are likely to hold business-related data. The types of these records indicate how such business data is used within individual programs, or across the entire software system analyzed.

Third, we can find *type dependencies* between programs and copybooks. Clearly, if a program uses a variable declared in a copybook, the program depends on that copybook. A second possibility, which we encountered in our case study, is that a copybook C_p containing a section (to be included in the procedure division) uses variables declared in a separate copybook C_d (to be included in the data division).[4] This leads to an inferred type dependency

---

[4] Since COBOL sections cannot have parameters, global variables are the only way to pass data to sections.

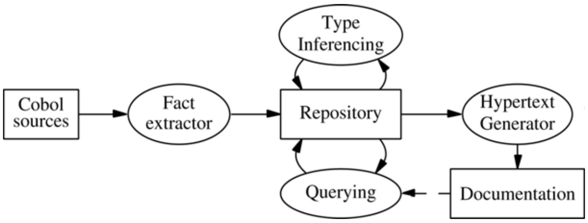


Fig. 4. Overview of the TYPEEXPLORER toolset.

between the using copybook C_p and the declaring copybook C_d. In our case study, the programmers had tried to document such dependencies in comments in both copybooks — however, our analysis found additional dependencies not documented at all.

Last but not least, we provide index files for types and programs, listing all words found in types, type names, types used in signatures, types used in persistent data stores, and so on. Moreover, we augment existing index files listing all programs, tables, and so on with additional type information, such as the type signature which concisely reveals the intended purpose of a program. These index files are included at the top level, but also at the subsystem, program, type cluster, and copybook level.

Index files can be simply sorted by name or by a particular metric of interest, such as the number of variables of that type. We color items with a metric value higher than the average plus one standard deviation red, in order to indicate that these may need special attention during maintenance. We have also used various graphical representations in order to show, for example, correlations between program size and the maximum value for the number of subtype metric, but these have not been included in TYPEEXPLORER yet.

5. Implementation

The architecture of the TYPEEXPLORER toolset is shown in Fig. 4. The dashed line between documentation and querying indicates the dynamic queries available in the on-line TYPEEXPLORER.

The toolset follows an extract–query–view approach, separating source code analysis, inferencing and presentation. This approach, also adopted in such tools as Rigi [42], PBS [38], Dali [21], and DOCGEN [9], makes it easier to adapt to different source languages or to other ways of presenting the types found. The TYPEEXPLORER toolset incorporates the COBOL type inferencing tools presented in [11].

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

In the second phase, the derived facts are combined and abstracted to infer a number of conclusions regarding type relations. One of the tools we use for inferring type relations is `grok`, a calculator for *Tarski relational algebra* [18]. Relational algebra provides operators for relational composition, for computing the transitive closure of a relation, for computing the difference between two relations, and so on. We use it, for example, to turn the derived type facts into the required equivalence relation. Finally we store the derived and inferred facts in the MYSQL relational database.[5]

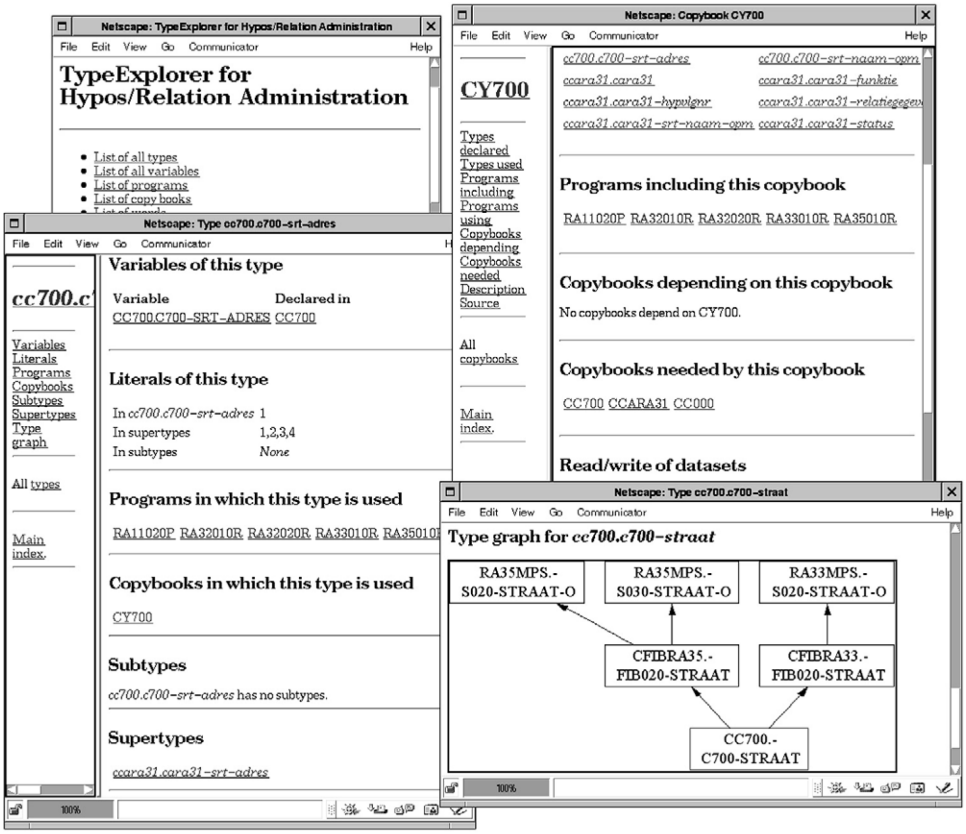
In the final phase, we query the database and generate hypertext documentation. We use PHP[6] to generate HTML code based on queries on the database. PHP is an HTML-embedded scripting language that was developed to allow web developers to write dynamically generated pages quickly. It contains support for a wide range of databases, including MYSQL. The on-line version of TYPEEXPLORER utilizes PHP as a server-side scripting engine to generate HTML code dynamically. For the off-line TYPEEXPLORER, PHP is used at “compile time” to generate static HTML pages.

The pages documenting types contain pictures of type graphs showing the subtypes and supertypes of a type. These type graphs are coupled to imagemaps that connect URLs to nodes in the picture allowing the user to navigate through the documentation by clicking in the graph. These graphs are extracted from the database in a Java program using the

---

[5] http://www.mysql.org/.

[6] PHP: PHP Hypertext Preprocessor. http://www.php.net/.



TypeExplorer for Hypos/Relation Administration

- List of all types
- List of all variables
- List of programs
- List of copy books

CY700

Types declared
Types used
Programs including
Programs using
Copybooks depending
Copybooks needed
Description
Source

All copybooks

Main index.

cc700.c700-srt-adres *cc700.c700-srt-naam-opm*
ccara31.cara31 *ccara31.cara31-funktie*
ccara31.cara31-hypvlgnr *ccara31.cara31-relatiegegev*
ccara31.cara31-srt-naam-opm *ccara31.cara31-status*

Programs including this copybook

RA11020P RA32010R RA32020R RA33010R RA35010R

Copybooks depending on this copybook

No copybooks depend on CY700.

Copybooks needed by this copybook

CC700 CCARA31 CC000

Read/write of datasets

cc700.c

Variables
Literals
Programs
Copybooks
Subtypes
Supertypes
Type graph

All types

Main index.

Variables of this type

Variable	Declared in
CC700.C700-SRT-ADRES	CC700

Literals of this type

In *cc700.c700-srt-adres*	1
In supertypes	1,2,3,4
In subtypes	*None*

Programs in which this type is used

RA11020P RA32010R RA32020R RA33010R RA35010

Copybooks in which this type is used

CY700

Subtypes

cc700.c700-srt-adres has no subtypes.

Supertypes

ccara31.cara31-srt-adres

Type graph for *cc700.c700-straat*

Fig. 5. The TypeExplorer in action.

JDBC interface[7] to MySQL. The layout and imagemaps for these images are generated using the `dot` graph drawing package [15].

6. Using type explorer

TypeExplorer helps a software engineer to take a *typeful* [4] look at their legacy system. In this section, we will discuss what sort of questions can be fruitfully answered by navigating through a legacy system using TypeExplorer. Clearly, TypeExplorer reveals so much information that many different questions can be answered using it. We will focus on two extremes: first, we will see that types are the natural way to reveal structure at the detailed level of individual *variables*; next we will cover how TypeExplorer helps to get a high level overview of the overall system *architecture*. Since the latter is, in our opinion, the most surprising application, we will focus most of our attention on architectural understanding using types.

Our running example will be a real life Cobol/cics system called Mortgage of approximately 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. An example screen shot from a session using TypeExplorer is shown in Fig. 5. It shows the main index, the page derived for copybook `CY700`, the page for type *cc700.c700-srt-adres*, as well as the type graph for one of the other types used in `CY700`.

6.1. Supporting maintenance tasks

One possible way of using TypeExplorer for Mortgage is to support maintenance tasks related to specific domain concepts or variables. A (fairly common) example is to modify the representation of a group of variables

---

[7] MySQL JDBC drivers. http://www.worldserver.com/mm.mysql/.

(for example, expanding the *kind* variables in Fig. 3 from two to three digits). Since COBOL has no facilities for encapsulating such a representation using explicitly declared types, this usually involves a painful search for all other variables affected by this modification, including those via chains of assignments. TYPEEXPLORER helps the maintainer to operate at the higher type level, which immediately provides all related variables.

6.2. *Architectural structures*

TYPEEXPLORER can be used to analyze the as-implemented software *architecture* of a system. The SEI school of architecture defines software architecture as “the structure or structures of the system, which consist of elements, their externally visible properties, and the relationships among them” [1,8]. Bass et al. emphasize that there generally are multiple structures (called *architectural structures*), and that no one structure holds the irrefutable claim to being *the* architecture [1]. Example architectural structures manifest themselves at the level of modules, processes, data flow, control flow, and so on. Following [8], “each structure provides a *view* that imparts a particular kind of understanding of the architecture”. We argue that the *type* structure of a system is an additional architectural structure, which is important not only for systems constructed using strongly typed languages, but also for legacy systems built using untyped languages such as Cobol. TYPEEXPLORER helps to inspect this type structure. To illustrate this, we will navigate through the MORTGAGE case study, and discuss some architectural issues of interest.

6.3. *Exploring* MORTGAGE*’s architecture*

When exploring MORTGAGE, a natural starting point is the index listing all programs together with their inferred signature. When doing this, one observation can be immediately made: The type of the first formal parameter of all batch programs is the same — the *program-fields* type. This raises the question of why this is so, and what sort of type this *program-fields* type is. Inspection shows us that it is a record type, storing the name of the program, the current status, the name of the files currently processed, etc. Moreover, it holds data which is not necessary for the proper execution of the program. Instead, the data is used to quickly find the program responsible for the problems if one of the batch runs crashes.

This shared first parameter shown by TYPEEXPLORER thus immediately leads to an architectural requirement, namely that the system should support fast repairs and restarts at the proper position whenever one of the batch runs crashes in the middle of the night.

TYPEEXPLORER also shows us that this convention is actually used. The *program-fields* record contains one field (the *subroutine* field) holding the name of the program currently being run. TYPEEXPLORER lists all literal values that are used for (i.e., assigned to variables of) the type *subroutine*, This list exactly corresponds to the list of all batch programs, which is the result of the fact that each program correctly starts by setting the *subroutine* field to the program’s name.

It is interesting to observe that MORTGAGE also clearly shows that just looking at the *names* of formal parameters is not sufficient. To see why this is so, we take a look at the *on-line* part of MORTGAGE (the part invoked from screens via CICS). The first parameter in each on-line program is the same, namely `DFHCOMMAREA`. However, each one has a different type! All `DFHCOMMAREA` variables are strings of different lengths. The specific name `DFHCOMMAREA` is required by CICS. The first thing each program does is to assign that variable to a more structured record variable. It is the type of that structured record variable that TYPEEXPLORER recognizes as the appropriate type for the first parameter of the linkage sections, which it displays in the inferred signature.

TYPEEXPLORER also helps us to understand the meaning of the program parameters. For example, many programs in MORTGAGE have integer-valued numbers as parameters (having picture string `S(9) COMP-3`). Often, these are in fact enumeration types, in which case TYPEEXPLORER recognizes them as such. Several programs turn out to have a parameter named *function*, with five to ten permitted values. Based on this function value, the program performs one of several functions. This leads us to two design decisions: different (but related) functions are grouped into programs, and the mechanism used is a switch on an enumerated value, instead of the Cobol feature in which one program can have multiple entry points.

Last but not least, TYPEEXPLORER shows how such *function* enumeration parameters are passed from one program to another. As an example, one of the MORTGAGE programs contains a parameter for determining how a person’s name is formatted (full first names, one initial only, with title, and so on), and another to format street names (capitalized, street abbreviated, and so on). One of the top level programs has ten different parameters, corresponding to these formatting codes. The types inferred exactly show how each of the codes (which are all integer numbers) correspond to the parameters of the various formatting programs.

In short, TYPEEXPLORER can be used to discuss whether requirements such as crash recovery are properly supported, how functionality is grouped in modules, and how modules are dependent via types. Other architectural issues can be identified using TYPEEXPLORER by studying the type relationships between copybooks, the use of database record types across programs, and so on.

7. Concluding remarks

In this paper, we have shown how hypertext-based program understanding tools can achieve higher levels of abstraction by using inferred type information for cases where the underlying software system is written in a weakly typed language. We proposed TYPEEXPLORER, a tool for browsing COBOL legacy systems based on these types. The main contributions of the paper are in the following areas:

Presentation Although types are an invented abstraction, not directly present in the code, we showed how they can be made tangible by displaying a name for them, associated domain concepts, literal values, and variable use in the source code. Moreover, type graphs help one to see types in context, and view their relationships to other types. Last but not least, type information can be integrated with pages documenting programs, databases and copybooks, extended them with type links for program signatures, copybook dependencies, and record types for persistent data stores.

Implementation We have described an implementation based on the extract–query–view paradigm, using Tarski relational algebra, SQL, and PHP to realize both an on-line and off-line version of TYPEEXPLORER.

Use We have shown how navigating through a legacy system using TYPEEXPLORER provides useful information both at the detailed level of individual programs and at the higher level of the overall architecture. We have used TYPEEXPLORER to document an actual system, and used the resulting hypertext to identify type dependencies between programs, to understand design decisions, and to highlight requirements such as support for crash recovery.

Future work consists of applying TYPEEXPLORER to other COBOL systems as well. Furthermore, distributing TYPEEXPLORER to industrial users will raise additional requirements and questions, and offer opportunities to compare TYPEEXPLORER with the tools that they are already using.

Another interesting area of future work is to use TYPEEXPLORER to support the migration of COBOL to the new COBOL standard, which is an object-oriented extension of COBOL-85. This new version of COBOL does support types, and offers the possibility of using type definitions. Our tools provide the technology for taking advantage of this new possibility.

Acknowledgement

We would like to thank Jan Heering (CWI) for commenting on a draft of this paper.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, 2003.
[2] M.G.J. van den Brand, A. Sellink, C. Verhoef, Generation of components for software renovation factories from context-free grammars, in: 4th Working Conference on Reverse Engineering, WCRE'97, IEEE, 1997, pp. 144–155.
[3] P. Brown, Integrated hypertext and program understanding tools, IBM Systems Journal 30 (3) (1991) 363–392.
[4] L. Cardelli, Typeful programming, in: E.J. Neuhold, M. Paul (Eds.), Formal Description of Programming Concepts, Springer-Verlag, Berlin, 1991, pp. 431–507.
[5] S. Chandra, T. Reps, Physical type checking for C, in: Workshop on Program Analysis for Software Tools and Engineering, PASTE'99, September, ACM Press, 1999, pp. 66–75; SIGSOFT Software Engineering Notes 24(5).
[6] X.P. Chen, W.T. Tsai, J.K. Joiner, H. Gandamaneni, J. Sun, Automatic variable classification for COBOL programs, in: 18th Ann. Int. Computer Software and Applications Conference, COMPSAC'94, IEEE, 1994, pp. 432–437.
[7] E.J. Chikofsky, J.H. Cross, Reverse engineering and design recovery: A taxonomy, IEEE Software 7 (1) (1990) 13–17.
[8] P. Clements et al., Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2003.

[9] A. van Deursen, T. Kuipers, Building documentation generators, in: International Conference on Software Maintenance, ICSM'99, IEEE Computer Society, 1999, pp. 40–49.

[10] A. van Deursen, L. Moonen, Type inference for COBOL systems, in: 5th Working Conference on Reverse Engineering, WCRE'98, IEEE Computer Society, 1998, pp. 220–230.

[11] A. van Deursen, L. Moonen, An empirical study into cobol type inferencing, Science of Computer Programming 40 (2–3) (2001) 189–211. July.

[12] P. Devanbu, Y.-F. Chen, E. Gansner, H. Müller, J. Martin, CHIME: Customizable hyperlink insertion and maintenance engine for software engineering environments, in: 21st International Conference on Software Engineering, ICSE-99, ACM, 1999, pp. 473–482.

[13] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sorensen, M. Tofte, Anno Domini: From type theory to Year 2000 conversion tool, in: 26th Symp. on Principles of Progr. Languages, POPL'99, ACM, 1999, pp. 1–14.

[14] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Müller, J. Mylopoulos, S.G. Perelgut, The software bookshelf, IBM Systems Journal 36 (4) (1997) 564–593.

[15] E.R. Gansner, E. Koutsofios, S. North, K.-P. Vo, A technique for drawing directed graphs, IEEE Transactions on Software Engineering 19 (3) (1993) 214–230.

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

[17] J. Hart, A. Pizzarello, A scaleable, automated process for year 2000 system correction, in: 18th International Conference on Software Engineering, ICSE-18, IEEE, 1996, pp. 475–484.

[18] R. Holt, Structural manipulations of software architecture using Tarski relational algebra, in: 5th Working Conference on Reverse Engineering, WCRE'98, IEEE Computer Society, 1998, pp. 210–219.

[19] G.E. Kaiser, S.E. Dossick, W. Jiang, J.J. Yang, An architecture for WWW-based hypercode environments, in: 19th International Conference on Software Engineering, ICSE-97, ACM, 1997, pp. 3–13.

[20] K. Kawabe, A. Matsuo, S. Uehara, A. Ogawa, Variable classification technique for software maintenance and application to the year 2000 problem, in: P. Nesi, F. Lehner (Eds.), Conference on Software Maintenance and Reengineering, IEEE Computer Society, 1998, pp. 44–50.

[21] R. Kazman, J. Carrière, Playing detective: Reconstructing software architecture from available evidence, Automated Software Engineering 6 (1999) 107–138.

[22] T. Kuipers, L. Moonen, Types and concept analysis for legacy systems, in: 8th International Workshop on Program Comprehension, IEEE Computer Society Press, 2000, June.

[23] L.D. Landis, P.M. Hyland, A.L. Gilbert, A.J. Fine, Documentation in a software maintenance environment, in: Conference on Software Maintenance, IEEE Computer Society, 1988, pp. 66–73.

[24] A. von Mayrhauser, A.M. Vans, Program comprehension during software maintenance and evolution, IEEE Computer (1995) 44–55, August.

[25] E. Merlo, J.F. Girard, L. Hendren, R. De Mori, Multi-valued constant propagation analysis for user interface reengineering, International Journal of Software Engineering and Knowledge Engineering 5 (1) (1995) 5–23, March.

[26] L. Moonen, Generating robust parsers using island grammars, in: 8th Working Conference on Reverse Engineering, IEEE Computer Society Press, 2001, pp. 13–22, October.

[27] L. Moonen, Exploring Software Systems, Ph.D. Thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, December 2002.

[28] L. Moonen, Lightweight impact analysis using island grammars, in: 10th International Workshop on Program Comprehension, IWPC 2002, June, IEEE Computer Society Press, 2002.

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

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

[31] R. O'Callahan, D. Jackson, Lackwit: A program understanding tool based on type inference, in: 19th International Conference on Software Engineering, ICSE-97, ACM, 1997, pp. 338–348.

[32] Ch. de Oliveira Braga, A. von Staa, J.C.S. do Prado Leite, Documentu: A flexible architecture for documentation production based on a reverse-engineering strategy, Journal of Software Maintenance 10 (1998) 279–303.

[33] J. Palsberg, Type-based analysis and applications. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE, 2001, pp. 20–27.

[34] V. Rajlich, Incremental redocumentation with hypertext, in: 1st Euromicro Working Conference on Software Maintenance and Reengineering, CSMR 97, IEEE Computer Society Press, 1997, pp. 68–72.

[35] V. Rajlich, S. Varadarajan, Using the web for software annotations, International Journal of Software Engineering and Knowledge Engineering 9 (1) (1999) 55–72.

[36] G. Ramalingam, J. Field, F. Tip, Aggregate structure identification and its application to program analysis, in: 26th Symp. on Principles of Progr. Languages, POPL'99, ACM, 1999, pp. 119–132.

[37] M. Siff, T. Reps, Program generalization for software reuse, in: ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE'96, 1996, pp. 135–146; Software Engineering Notes 21 (6).

[38] S.E. Sim, C.L.A. Clarke, R.C. Holt, A.M. Cox, Browsing and searching software architectures, in: International Conference on Software Maintenance, ICSM'99, IEEE Computer Society, 1999, pp. 381–390.

[39] S.E. Sim, R.C. Holt, The ramp-up problem in software projects: A case study of how software immigrants naturalize, in: 20th International Conference on Software Engineering, ICSE-97, ACM, 1998, pp. 361–370.

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

[41] N. Williams-Preston, New type signatures for legacy Fortran subroutines, in: Workshop on Program Analysis for Software Tools and Engineering, PASTE'99, September, ACM Press, 1999, pp. 76–85; SIGSOFT Software Engineering Notes 24 (5).

[42] K. Wong, S.R. Tilley, H.A. Müller, M.-A.D. Storey, Structural redocumentation: A case study, IEEE Software 12 (1) (1995) 46–54.