Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Exploring Legacy Systems Using Types

A. van Deursen, J.M.F. Moonen

Software Engineering (SEN)

SEN-R0031 November 30, 2000

Report SEN-R0031
ISSN 1386-369X

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

Exploring Legacy Systems Using Types*

Arie van Deursen Leon Moonen

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

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.

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, documentation generation, variable usage, hypertext.

Note: To appear in *Proceedings of the 7th Working Conference on Reverse Engineering. IEEE Computer Society, 2000*.

Note: Work carried out under projects SEN 1.1, *Software Renovation* and SEN 1.5, *Domain-Specific Languages*.

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 [19]. 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, 5, 15, 16, 18]. 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 [13, 16].

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

*We are in the process of seeking patent protection for the ideas described here.

The solution we propose is to *infer* types for COBOL automatically, based on an analysis of the *use* of variables [6]. This results in types for variables, program parameters, database records, literal values, and so on, which can be used to understand the relationships between programs, copybooks, databases, screens, and so on.

In earlier work, we presented an algorithm and toolset for determining types in COBOL systems [6, 7]. 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. Section 3 discusses the theory of type inferencing for COBOL. The design of the hypertext structure used by TYPEEXPLORER is covered in Section 4. The techniques that were used for implementation are described in Section 5. We 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

A growing body of literature on web-based program comprehension exists [3, 5, 15, 16, 18, 8]. 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 [8]. PAS is a system that can be used to incrementally add *partitioned annotations of software* [16]. *Documentu* derives documentation from COBOL sources based on special comment tags added by the programmer [15].

DocGen is a tool for generating hyperlinked visual and textual documentation from COBOL and batch job sources [5]. Distinguishing characteristics of DocGen include extraction based on *island grammars* 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 was generated by DocGen.

Many architecture extraction tools (such as Rigi [21], PBS [18], Dali [12], and also DocGen and TYPEEXPLORER) adopt the extract-query-view approach, extracting facts from sources, querying a database filled with facts, and presenting these facts in various ways, for example using hypertext. PBS, which has been applied mostly to analyze C systems such as Linux, uses Tarski relational algebra for querying, which is also used in the implementation of TYPEEXPLORER. Dali emphasizes the need for an open tool set, in which many different tools can be plugged in, when necessary. New in our work is the addition of type inferencing to the suite of analysis techniques used by such tools.

Closest in aims to the integration of type analysis and program understanding is Lackwit [14], 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?" Other work based on type inferencing includes "physical type checking of C", which is a stronger form of type checking for type casts involving pointers to structures [4], and the analysis of Fortran programs in order to find new type signatures for subroutines [20]. Type-based analysis of COBOL, for the purpose of year 2000 analysis, is presented by [9, 17]: both provide a type inference algorithm that splits aggregate structures into smaller units based on assignments between records that cross field boundaries.

Our own work on type inferencing started with [6], where we present the basic theory for COBOL type inferencing. In [7], we described 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). In our current paper, we do not extend the theory of type inferencing: instead we explain how inferred types can be presented using hyper-text, and used to understand COBOL systems at various levels of abstraction.

More references to related work can be found in [5] (documentation generation) and [6, 7] (type inference for COBOL).

[1] Documentation generation services using DocGen are available via the *Software Improvement Group*, `www.software-improvers.com`

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.

From the perspective of types, 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, it 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* to represent procedures. Neither sections nor paragraphs can have formal parameters, forcing the programmer to use global variables for parameter passing.

In [6], we propose a method to infer 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.

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.

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.

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.

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*.

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.

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*.

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 [6], 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.* [9] and Ramalingam *et al.* [17] have published an algorithm which splits aggregate structures in smaller "atoms", such that types can be propagated through record fields even if the records do not have the same structure.

[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.

Pollution We speak of *type pollution* when the types of two variables are inferred to be equivalent but would have been given different types in case 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 [7], we describe a range of experimental data showing the effectiveness of subtyping for dealing with pollution.

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 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. 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 not only want to see 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 would 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 is necessary to avoid pollution, whereas for others subtyping should actually have been type equivalence.

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 sub 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	list of 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.

Figure 1. Information presented for a type τ.

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). In case the TYPEEXPLORER user is not interested in the subtyping details of a particular type, he 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 webbrowser). 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.

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, 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 behaviour based on the comment prologue [5]. 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 Figure 1.

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 most common case is that 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 will infer equivalences between these field types, If they are of different shape, *aggregate structure identification* [9, 17] 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.

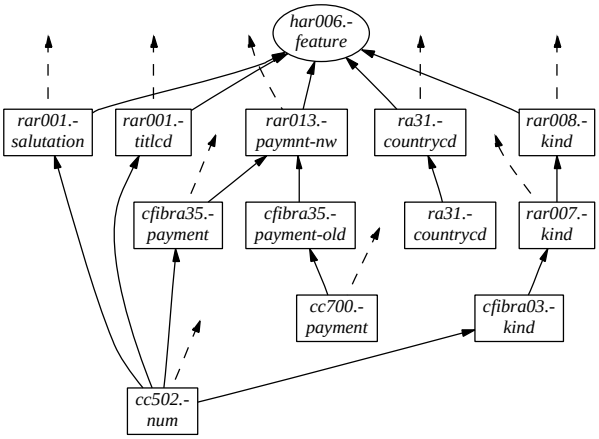

Figure 2. Example Type Graph

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 sub- and supertypes of τ in a *type graph*. An example type graph is shown in Figure 2. 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 a (red) ellipse. In Figure 2 it has name

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

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 Figure 2, 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*, etc. Such a type with several different subtypes is typically the *input* parameter of a procedure or program, where each incoming edge corresponds to the subtype of an actual parameter. If we would not infer 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 sub or supertypes of the type selected, *har006.feature*. An example is the left most *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, but 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 to keep them small: If we would expand all dashed arrows transitively, the type graph for *har006.feature* would become several hundreds nodes larger.

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 [5].

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 not only provides 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 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 to 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,

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

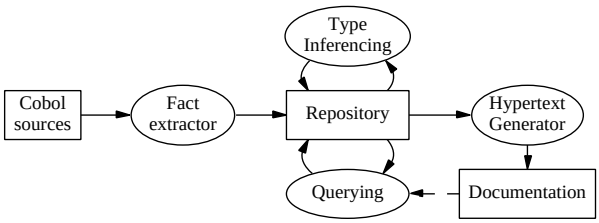

Figure 3. Overview of the TYPEEXPLORER tool set.

type cluster, and copybook level.

5. Implementation

The architecture of the TYPEEXPLORER tool set is shown in Figure 3. 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 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 [7].

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, etc.).

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* [11]. 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 sub- 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 JDBC interface[7] to MySQL. The layout and imagemaps for these images are generated using the `dot` graph drawing package [10].

6. Using Type Explorer

TYPEEXPLORER helps a software engineer to take a typeful look at his 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 spend most of our attention to 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 Figure 4. 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 (for example, expanding the *kind* variables in Figure 2 from two to three digits). Since COBOL has no facilities to encapsulate such a representation using explicitly declared types,

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

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

[7] Mark Matthews MySQL JDBC drivers. Available from: `http://www.worldserver.com/mm.mysql/`

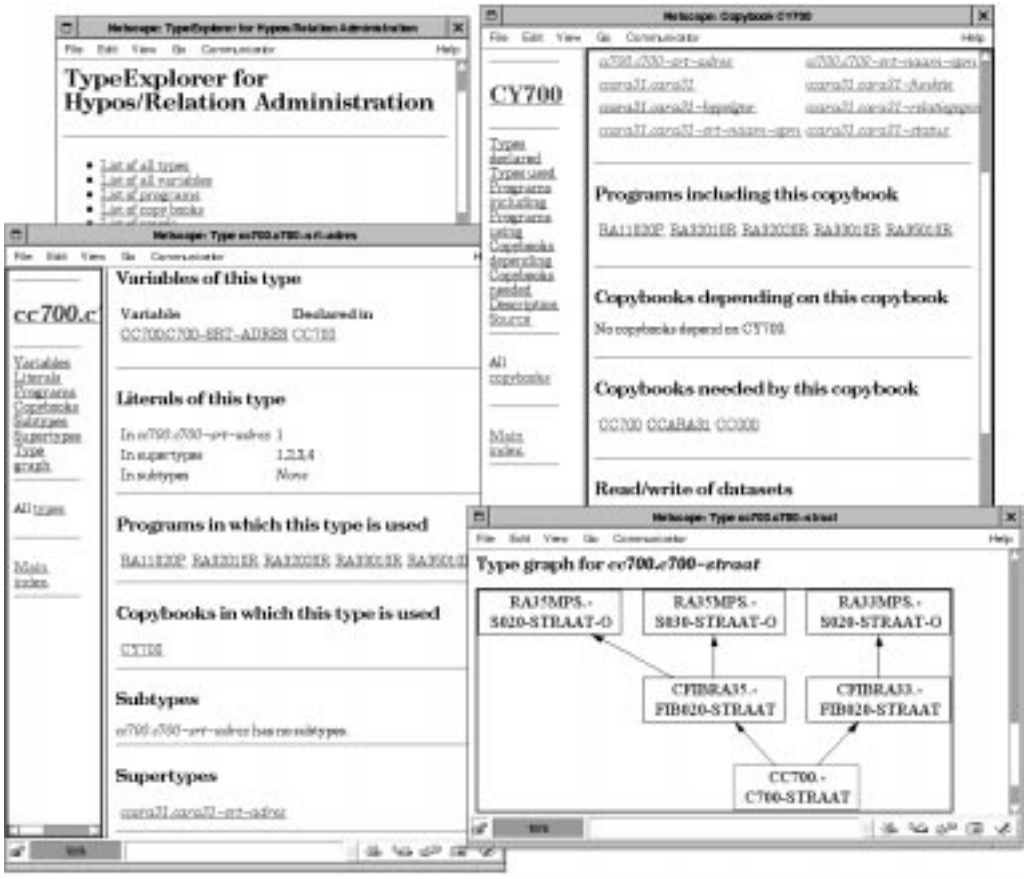

Figure 4. The TYPEEXPLORER in action.

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 the as-implemented software *architecture* of a system. Bass *et al.* [1] define this as the system's structure, which comprises software components, the externally visible properties of those components, and the relationships among them. 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. Example architectural structures manifest themselves at the level of modules, processes, data flow, control flow, and so on. 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.

Bass *et al.* [1] provide three reasons why software architecture is important: (1) it helps in *communication among stakeholders*; (2) it makes *design decisions* explicit; and (3) it provides a *transferable abstraction of a system*. TYPEEXPLORER helps to achieve these goals for type structures as well. 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 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. More-

over, 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 of each on line programs is the same, namely `DFHCOMMAREA`. However, they all have 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 5 to 10 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 10 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 be 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 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 applied TYPEEXPLORER to an actual system, and used it to identify type-dependencies between programs, understand design decisions, and to highlight requirements such as support for crash recovery.

Our next step will be to distribute TYPEEXPLORER to industrial users. Undoubtedly, this will raise additional requirements and questions, on which we will report in the near future. One possible extension is to propagate types via batch jobs (JCL) as well, thus arriving at better types for the datafiles processed.

Another 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 to take advantage of this new possibility.

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

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[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] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems J.*, 30(3):363–392, 1991.

[4] S. Chandra and T. Reps. Physical type checking for C. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE'99*, pages 66–75. ACM Press, September 1999. SIGSOFT Software Engineering Notes **24**(5).

[5] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[6] 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.

[7] 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.

[8] P. Devanbu, Y-F. Chen, E. Gansner, H. Müller, and J. Martin. CHIME: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *21st Int. Conf. on Software Engineering, ICSE-99*, pages 473–482. ACM, 1999.

[9] 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.

[10] 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.

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

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

[13] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

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

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

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

[17] 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.

[18] 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.

[19] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *20th Int. Conf. on Software Engineering; ICSE-97*, pages 361–370. ACM, 1998.

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

[21] 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.