AnnoDomini in Practice: A Type-Theoretic Approach to the Year 2000 Problem

Peter Harry Eidorff Fritz Henglein Christian Mossin
Henning Niss Morten Heine B. Sørensen Mads Tofte

Dept. of Computer Science, Univ. of Copenhagen (DIKU) and Hafnium ApS
Email: {phei,henglein,mossin,hniss,rambo,tofte}@diku.dk
WWW: http://www.diku.dk, http://www.hafnium.com

Abstract. AnnoDomini is a commercially available source-to-source conversion tool for finding and fixing Year 2000 problems in COBOL programs. AnnoDomini uses type-based specification, analysis, and transformation to achieve its main design goals: flexibility, completeness, correctness, and a high degree of safe automation.

1 Introduction

The *Year 2000 (Y2K) problem* refers to the inability of software and hardware systems to process dates in the 21st century correctly.[1] The problem arises from representing calendars years by their last two digits and thus restricting the range of representable years to 1900-1999. Starting some 40 years ago, this convention was established as one of numerous techniques for conserving precious memory space.

The most widespread *Year-2000-unsafe* date representation consists of six characters. It has two characters each for the day of the month, the month of the year, and the calendar year, often in the order year-month-day (YYMMDD). The string "981106", for example, represents November 6th, 1998. The problem, of course, is that no provision is made for representing years in the 21st century: "00" represents 1900, not 2000.

Since the year 2000, mistakenly represented as "00", comes before e.g. "99", comparison of two-digit years may produce unexpected results in the 21st century, and this may incur problems in operating with e.g. expiry dates. Similarly, arithmetical operations involving two-digit years may produce unexpected results, affecting e.g. interest calculations.

The Y2K problem affects countless systems at all levels: embedded systems, operating systems, applications and data bases that process or contain dates. Both its size and consequences are staggering. Cost estimates vary widely, but according to Capers Jones, "the costs of fixing the year 2000 problem appear to constitute the most expensive single problem in human history" [2, p. xxiii].

Updating application programs to become Year 2000 compliant usually involves a combination of *expansion* and *masking*. Expansion refers to expanding

[1] We adopt the convention of viewing the year 2000 as belonging to the 21st century.

unsafe two-digit years to four-digit years in applications, data bases, files, etc. Expansion can be expensive, however: it requires that not only application programs be changed, but also data bases, files and all other programs communicating dates. Masking denotes a variety of methods for extending two-byte year representations into the 21st century; e.g. *windowing*, *compression* and *encapsulation*. These techniques aim at extending the lifetime of existing data in data bases and files as well as screen and print maps into the 21st century. In windowing, for example, a pivot year determines whether a two-digit year belongs to the 20th or the 21st century. For example, with pivot 70, 79 represents 1979, and 41 represents 2041.

AnnoDomini[2] is a tool (and accompanying method) for finding and fixing Year 2000 problems in COBOL programs. It accommodates expansion as well as masking by source-to-source transformation of COBOL programs. The converted programs do not require special compiler support, but compile and execute in their existing operating environment.

AnnoDomini consists of three components: an analysis and conversion engine (60,000 lines of Standard ML), a graphical user interface (10,000 lines of Visual Basic), and IBM's Live Parsing Editor (a syntax-sensitive program editor). The three components are tightly integrated, as will be explained in what follows. AnnoDomini runs on Windows NT 4.0 and Windows 9X, and is commercially available from Computer Generated Solutions, Inc. (an IBM business partner)—see http://www.cgsinc.com and http://www.hafnium.com.

AnnoDomini uses type-based specification, analysis, and transformation to achieve its main design goals: flexibility, completeness, correctness, and a high degree of safe automation. The type-theoretic foundations of AnnoDomini have been described elsewhere [1]. In the present brief account we aim to demonstrate how AnnoDomini actually works in practice—emphasizing the role of types—although we shall ignore many practical issues, e.g., key fields with years, alignment of key fields, aliasing, editing characters, padding/truncation, justification, and usage.

2 The AnnoDomini Approach

In COBOL programs, dates are represented using the data types and operations of the source language: strings of characters and digits, and flat records. Their *intensional* interpretation as representations of dates is not explicit. The AnnoDomini approach is based on *reverse engineering* the programmer-intended date interpretations as *abstract types*. This is done in three conceptual phases: seeding, type checking, and conversion.

2.1 An Example COBOL Program Fragment

To illustrate the AnnoDomini approach, we consider the following fragment of a COBOL program.

[2] AnnoDomini is a registered trademark of Hafnium ApS.

```
 77 CUR-DATE PIC 999999.
 77 LRD      PIC 999999.
 77 COLUMN   PIC 99.
    IF CUR-DATE > LRD PERFORM ISSUE-LAST-REMINDER.
    IF COLUMN < 80    PERFORM DISPLAY-STATUS.
```

The first three lines are declarations of three variables: CUR-DATE (containing six-digit data, signified by the six occurrences of 9), LRD (also containing six-digit data), and COLUMN (containing two-digit data).

The first statement invokes procedure ISSUE-LAST-REMINDER if CUR-DATE ("current date") is greater than LRD ("last reminder date").

The current date will most likely have form 000101 on January 1st, year 2000, so with a last reminder date of, say, December 31st, year 1999 (991231), no last reminder will ever be issued as a result of running the application in the year 2000 or later. This is not the desired behavior of the program.

The last statement invokes procedure DISPLAY-STATUS, provided the value contained in COLUMN is less than 80. This comparison has nothing to do with years and will continue to work in the 21st century.

2.2 Seeding

In the first phase of the AnnoDomini approach the user *seeds* the program with year (and possibly non-year) information. This is done by annotating variable declarations with *Type System 2000 (TS2K) types* that specify where years occur in them, if at all.

TS2K types are concatenations of the following different base types:

1. YYYY: four-digit year;
2. WW: two-digit, windowed year relative to a fixed pivot, by default 00;[3]
3. N: single non-year character;
4. -...- (n occurrences of -): n characters of unknown type.

For example, from the declarations alone in our example program we might guess that CUR-DATE is a six-digit date with a leading year, and that COLUMN is a column position at the terminal screen and hence unrelated to years. What LRD denotes, is not clear from the declaration alone. Thus, a seeded version of the example program might read:

```
*TS2K WWNNNN
 77 CUR-DATE PIC 999999.
*TS2K ------
 77 LRD      PIC 999999.
*TS2K NN
 77 COLUMN   PIC 99.
    IF CUR-DATE > LRD PERFORM ISSUE-LAST-REMINDER.
    IF COLUMN < 80    PERFORM DISPLAY-STATUS.
```

[3] With pivot 00, two-digit windowed years are the two-digit years of the 20th century.

Since COBOL comment lines start with * in column 7, the above TS2K type declarations are treated as comments by the COBOL compiler. However, to AnnoDomini they provide type information.

Seeding can be done *automatically* or *manually.* Automatic seeding works by scanning variable names in a program, including all the libraries it imports, and looking for matches according to both lexical and data description criteria. Informally, for each program variable the user asks: “Could this variable contain a calendar year, based on its name and its data description?” For example, a variable named `DEP-DAT` and occupying 6 bytes, might represent a six-digit date (“departure date”). Then again, it might not (“deposition data”). Automatic seeding is specified by a combination of lexical inclusion and exclusion criteria and a list of target date types. These specifications can be configured interactively, and they can be stored in separate files for future use. Automatic seeding is known to be quick, but also error-prone since it depends on nomenclature for variable names. AnnoDomini presents a list of all matches along with annotation suggestions, but does not automatically accept the results as bona-fide year annotations. Instead, it expects the user to explicitly accept or reject them, possibly after inspecting the variable declarations through a point-and-click interface.

Manual seeding works by systematically checking the *interfaces* of a program; e.g., data base, file, terminal and print map descriptions. In COBOL, these are typically localized in shared libraries that are copied into programs by `COPY` statements, COBOL’s macro expansion and source library access mechanism. Manual seeding is less error-prone since it reduces guesswork. Since data base, file, and map descriptions need to be annotated only once, but are typically used by multiple programs, manual seeding need not be done for each program and is thus often a quite efficient and safe seeding method.

2.3 Type checking

In the second phase AnnoDomini *propagates* the seeding information to other data by *type inference.* In particular, types are propagated through comparisons and assignments. For instance, since our example program contains the statement

```
IF CUR-DATE > LRD PERFORM ISSUE-LAST-REMINDER.
```

the type of `CUR-DATE` is propagated to `LRD`, and AnnoDomini *suggests* that `LRD` be given the same type. As with seeding suggestions, the user accepts and rejects such suggestions through a point-and-click interface.

During propagation AnnoDomini also *checks* that the seeded and propagated types are *consistent* with each other. For example, based on its cryptic name, we might mistakenly have assumed that `LRD` is entirely composed of non-year data and have assigned the type `NNNNNN` to it, in which case the types of `CUR-DATE` and `LRD` would be inconsistent. In this case AnnoDomini signals a *type error.*

In general, type errors may stem from the following sources:

1. *Seeding error.* Seeding might be wrong; for instance, we might have assumed the incorrect type `NNNNNN` for `LRD`.

2. *Not a Year 2000 problem.* The type system does not allow both years and and non-years to occupy the same storage at different times, such as when printing both years and non-years through the same print buffer, as in the following program fragment.[4]

   ```
   *TS2K WW
    77 CUR-YEAR  PIC 99.
   *TS2K NN
    01 NON-YEAR  PIC 99.
   *TS2K NN
    77 PRINT-BUF PIC 99.
       MOVE CUR-YEAR TO PRINT-BUF.
       MOVE NON-YEAR TO PRINT-BUF.
   ```

3. *Year 2000 problem.* The error might signal a Year 2000 problem or other questionable computations on dates, e.g. a hardwired conversion between four-digit years to two-digit years, as in the following program fragment in which the two last digits of a four-digit year are moved into a two-digit variable utilizing COBOL's alignment and truncation rules. A similar coercion in the other direction is adding 1900 to a two-digit year. Such hardwired coercions do not generally work in the 21st century.

   ```
   *TS2K WW
    77 YEAR2 PIC 99.
   *TS2K YYYY
    77 YEAR4 PIC 9999.
       MOVE YEAR4 TO YEAR2.
   ```

AnnoDomini does not attempt to guess what the cause of a type error is and how to eliminate it. It suggests a number of plausible *corrective actions*, however. These include changing the declarations of the variables involved in the type incorrect statement—the relevant option in case of a seeding error.

Two other forms of suggestions are to insert `ASSUME` and `COERCE` annotations. For instance, for the print buffer example, AnnoDomini suggests annotating the `MOVE` statements with an `ASSUME` annotation; e.g.,

```
*TS2K WW
 77 CUR-YEAR  PIC 99.
*TS2K NN
 77 NON-YEAR  PIC 99.
*TS2K NN
 77 PRINT-BUF PIC 99.
*TS2K ASSUME CUR-YEAR IS NN
    MOVE CUR-YEAR TO PRINT-BUF.
    MOVE NON-YEAR TO PRINT-BUF.
```

[4] The `MOVE` statement is COBOL's assignment statement.

The ASSUME annotation tells the type checker that CUR-YEAR should be treated as having type NN *in this statement only.* (This is dangerous, of course, and therefore requires an *explicit* annotation in the source code.)

The COERCE annotation is used to convert between different year formats. For instance, AnnoDomini suggest annotating the type incorrect statement MOVE YEAR4 TO YEAR2 with a COERCE statement, e.g.,

```
*TS2K COERCE YEAR4 TO WW BY D4TO2NO
     MOVE YEAR4 TO YEAR2.
```

The coercion D4TO2NO converts a four-digit year to a value with the same year in windowed representation. The COERCE annotation is similar to the ASSUME annotation: the former instructs in the above example AnnoDomini to regard YEAR4 as having type WW. The difference is that, in the conversion phase COERCE annotations are replaced by code performing the coercions, whereas ASSUME annotations have no run-time significance.

AnnoDomini also provides point-and-click access to the statements causing type errors and to the declarations of the variables occurring in the type incorrect statement for manual browsing and editing of the source code.

AnnoDomini issues *warnings* for all relational and arithmetic operations on two-digit years as well as for all relational and arithmetic operations for which there is insufficient type information to determine whether their operands contain years or not. This is a case where seeding is incomplete, with potentially dangerous consequences. The user is expected to check the warnings to determine whether they cover over any potential Year 2000 problems. They can also be eliminated by strengthening the seeding to resolve the operand types.

Seeding and type checking are repeated, possibly interchangeably, until *all* type errors are eliminated and the program is *type correct.*

2.4 Conversion

The third and final phase consists of *virtual conversion* and *actual conversion.* During virtual conversion the user specifies Year 2000-safe types for each variable. For example

```
*TS2K WWNNNN->YYYYNNNN
  77 CUR-DATE PIC 999999.
*TS2K WWNNNN->YYYYNNNN
 77 LRD      PIC 999999.
*TS2K NN
 77 COLUMN   PIC 99.
    IF CUR-DATE > LRD PERFORM ISSUE-LAST-REMINDER.
    IF COLUMN < 80    PERFORM DISPLAY-STATUS.
```

is a virtual conversion which specifies that CUR-DATE and LRD should be *expanded* from a six-digit to an eight-digit date representation. The actual conversion is then fully automatic, yielding the following program fragment:

```
*TS2K YYYYNNNN
 77 CUR-DATE PIC 99999999.
*TS2K YYYYNNNN
 77 DUE-DATE PIC 99999999.
*TS2K NN
 77 COLUMN PIC 99.
    IF CUR-DATE > DUE-DATE  PERFORM ISSUE-LAST-REMINDER.
    IF COLUMN < 80          PERFORM DISPLAY-STATUS.
```

Alternatively, a virtual conversion can be specified by changing the default pivot for windowing from 00 to, say, 70 (this does not require any change to the program). Actual conversion then yields, fully automatically:

```
*TS2K WWNNNN
 77 CUR-DATE PIC 999999.
*TS2K WWNNNN
 77 DUE-DATE PIC 999999.
*TS2K NN
 77 COLUMN PIC 99.
    MOVE CUR-DATE TO ARG-1 OF ARGUMENT OF LT70N4-PARAMS.
    MOVE DUE-DATE TO ARG-2 OF ARGUMENT OF LT70N4-PARAMS.
    CALL "LT70N4" USING LT70N4-PARAMS.
    IF RESULT OF LT70N4-PARAMS = ’1’ PERFORM ISSUE-LAST-REMINDER.

    IF COLUMN < 80  PERFORM DISPLAY-STATUS.
```

The first four program statements call the AnnoDomini library routine `LT70N4` which compares dates with leading two-digit windowed years relative to pivot 70 (COBOL's built-in operator `>` does not work since it does not take the pivot into account). The year-unrelated comparison `COLUMN < 80` is left as is.

Each variable can have its own year representation. AnnoDomini has built-in support for four-digit years and windowed two-digit years, Apart from these, it allows abstract, user-defined two-digit years. These are denoted `AA(`t`)`, where t is the name of a user-defined library, which must contain the required arithmetic and relational operations. These can be type-checked on a par with the built-in year types.

Actual conversion is fully automatic: at the push of a button, data declarations are expanded as desired, calls to the specified coercions are inserted, and arithmetic and relational operations involving two-digit years are replaced by calls to AnnoDomini's Year 2000-safe library routines.

3 Conclusion and Related Work

The decision to base AnnoDomini on types has had a number of advantages.

First, types are good for explicating the intention of data. For instance, types are good for distinguishing between years and non-years, e.g. between 80 as a two-digit year and 80 as a column position. Similarly, types are good for distinguishing between different types of years, e.g. 80 as the two-digit windowed year 1980 and 80 as the year 80 A.D.

Second, types are good for discovering Year 2000 problems; for instance, the comparison `CUR-YEAR < 80` is problematic if `CUR-YEAR` is a two-digit year, whereas `COLUMN < 80` is unproblematic if `COLUMN` denotes non-year information.

Third, types are good for guiding transformations. In particular, year-unrelated code can be left as is.

Fourth, many design choices are made simply and elegantly by casting our analysis as a type inference. For instance, how to report inconsistent usage of years in the program? This obviously becomes a type error.

Finally, we have been able to benefit greatly from the design and implementation of ML, and its underlying theory, in developing AnnoDomini. For instance, some of the main results concerning type inference in [1] were adopted from Hindley-Milner type inference, with some modifications.

There is a vast literature on type theory and type-based program analysis. There are also numerous Year 2000 tools. Very few of those are semantics-based, however, and of those only AnnoDomini appears to be type-based with integrated automatic analysis and conversion.

The value of working with type notions in software understanding and reengineering has been observed previously by O'Callahan and Jackson [3]. Van Deursen and Moonen [5] describe type inference rules for COBOL for classifying data into sets of data representations. Subtyping is interpreted as subsumption of value sets. Their system specifies type equivalences, and it allows subtyping steps at assignments. Intuitively, this specifies a flow-insensitive data flow analysis, refined by data flow sensitivity at assignments.

Independently of us, Ramalingam, Field and Tip have developed basically the same unification algorithm as used by AnnoDomini's type inference algorithm [4]. They also demonstrate how their algorithm is applicable to Year 2000 program analysis.

References

1. P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H.B Sø rensen, and M. Tofte. Annodomini: From type theory to year 2000 conversion tool. In *Principles of Programming Languages*, January 1999.
2. C. Jones. *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences.* Addison-Wesley, ACM Press, 1998. ISBN 0-201-30964-5.
3. R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proc. 1997 International Conference on Software Engineering (ICSE '97), Boston, Massachusetts*, pages 338–348, May 1997.
4. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Principles of Programming Languages*, January 1999.
5. A. van Deursen and L. Moonen. Type inference for COBOL systems. To appear in Proc. 5th IEEE Working Conference on Reverse Engineering, Honolulu, Hawaii, October 1998.