AnnoDomini: From Type Theory to Year 2000 Conversion Tool

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

Dept of Computer Science, Univ. of Copenhagen (DIKU) and Hafniun ApS
Email: {phei, henglein, mossin, hnniss, rambo, tofte}@diku.dk

Abstract

AnnoDomini is a source-to-source conversion tool for making COBOL programs Year 2000 compliant. It is technically and conceptually built upon type-theoretic techniques and methods: type-based specification, program analysis by type inference and type-directed transformation. These are combined into an integrated software reengineering tool and method for finding and fixing Year 2000 problems. AnnoDomini's primary goals have been flexibility (support for multiple year representations), completeness (identifying all potential Year 2000 problems), correctness (correct fixes for Year 2000 problems) and a high degree of safe automation in all phases (declarative specification of conversions, no second-guessing or dangerous heuristics).

In this paper we present the type-theoretic foundations of AnnoDomini: type system, type inference, unification theory, semantic soundness, and correctness of conversion. We also describe how these foundations have been applied and extended to a common COBOL mainframe dialect, and how AnnoDomini is packaged with graphical user interface and syntax-sensitive editor into a commercially available software tool.

1 Introduction

1.1 The Year 2000 Problem

The Year 2000 (Y2K) Problem refers to the inability of software and hardware systems to process dates in the 21st century correctly. The problem arises from representing calendars years by their last two digits and thus restricting range of representable years to 1900-1999 A.D. Starting some 40 years ago, this convention was established as one of numerous techniques for conserving precious memory space and has evolved into a de facto standard for representing and exchanging dates containing years.

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 month-year-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 A.D., not 2000 A.D.

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 most of them predict global direct and indirect costs on the order of US$100,000,000,000,000. Capers Jones writes that "the costs of fixing the year 2000 problem appear to constitute the most expensive single problem in human history" [Jon98], p. xxii.

1.2 Remediation

Depending on consequential and repair costs, systems can be put into one of three categories:

Remediate. Systems that can be and need to be fixed, or systems that are too expensive or difficult to replace by the Year 2000.

Replace. Systems that either cannot be fixed or where fixing would be too expensive.

Ignore. Systems that need not be fixed, either because they are Y2K-safe or because they are not used, useless, dispensable or have little impact.

Y2K repairs (“remediation”) usually involve a combination of (date field) expansion and masking. Expansion refers to expanding unsafe 2-digit years to 4-digit years in applications, data bases, files, etc., ideally as part of adopting a standardized date representation (e.g. ISO 8001). 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 2-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 50, 59 represents 1959 A.D. and 41 represents 2041 A.D..

1.3 The AnnoDomini Approach

AnnoDomini is a method and tool for making COBOL programs Year 2000 compliant by source-to-source transformation. It processes one program at a time and allows for incremental conversion and testing of large applications. The

---

1 We adopt the popular convention of viewing the year 2000 A.D. as belonging to the 21st century.

2 AnnoDomini is a registered trademark of Hafniun ApS.
converted programs do not require special compiler support, but compile and execute in their existing operating environment.

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

1.3.1 Seeding

In the first phase the user seeds the program with year (and possibly nonyear) information about some of its variables, inputs and outputs; that is, they are annotated with Type System 2000 (TS2K) types that specify where years occur in them, if at all.

TS2K types are concatenations of WW (two-digit, windowed year), YYYY (four-digit year), N (single nonyear character) and α(n) (type variable denoting an unknown TS2K type with n characters). Annotations are special COBOL comment lines starting with *TS2K in column 7. For example,

*TS2K WWWWNN
01 DATA-1 PIC XXXXXX.

is an annotation that says that the six-character variable DATA-1 contains a windowed year in its first two bytes. 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 6-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 exclusively 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.

1.3.2 Type checking

In the second phase AnnoDomini propagates the seeding information to other data by type inference. For example, if the program contains the MOVE statement

*TS2K WWWWNN
01 DATE-1 PIC XXXXXX.
01 DATE-2 PIC XXXXXX.

MOVE DATE-2 TO DATE-1

the type of DATE-1 is propagated to DATE-2, and AnnoDomini suggests that DATE-2 be given the same type.

During propagation AnnoDomini also checks that the seeded and propagated types are consistent with each other. For example, in

*TS2K WWWWNN
01 DATE-1 PIC XXXXXX.
*TS2K YYYYNN
01 DATE-2 PIC XXXXXX.

MOVE DATE-2 TO DATE-1

DATE-1 and DATE-2 have different TS2K types (DATE-2 has a four-digit year in its first four bytes), and AnnoDomini signals an error. In general, type errors may signal:

1. Seeding error. Seeding might be wrong; e.g., for DATE-1, DATE-2 above or even another variable whose type has been propagated to DATE-1 or DATE-2.
2. Not a Year 2000 problem. The type system does not allow both years and nonyears to occupy the same storage at different times, such as when printing both years and nonyears through the same print buffer.
3. Year 2000 Problem. The error might signal a Year 2000 Problem or other questionable computations on dates.

AnnoDomini does not attempt to guess what the real cause of a type error is and how to eliminate it. It suggests a number of plausible corrective actions, however. In the example above, for the first case it suggests changing either the type annotation for the declaration of DATE-1 or DATE-2. For the second case it suggests annotating the MOVE statement with an ASSUME annotation; e.g.,

*TS2K ASSUME DATE-2 IS WWWWNNNN
MOVE DATE-2 TO DATE-1

The ASSUME annotation tells the type checker that I should be treated as having type WWWWNNNN in this statement only. (This is dangerous, of course, and therefore requires an explicit annotation in the source code.) Finally, for the third case it suggests specifying a coercion; e.g.,

*TS2K COERCE DATE-2 TO WWWWNN BY D4TO2H4
MOVE DATE-2 TO DATE-1

The coercion D4TO2H4 converts an eight-digit value starting with an four-digit year to a value with the same year in windowed representation, followed by the four nonyear bytes. The coercion is specified as an annotation — the actual conversion code is inserted automatically in the conversion phase. AnnoDomini also provides point-and-click access to the MOVE statement causing the type error and
to the declarations of the variables DATE-1 and DATE-2 occurring in it for manual browsing and editing of the source code.

AnnoDomini issues warnings 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.

Phases 1 and 2 are repeated, possibly interchangeably, until all type errors are eliminated and the program is type correct.

1.3.3 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 W00000000 -> YYYYYYYYY 01 DATE-1 PIC XXXXXX.*

specifies that the variable DATE-1 should be expanded from a six-digit to an eight-digit date representation.

Each variable can have its own year representation. AnnoDomini has built-in support for four-digit years, denoted by YYYY, and fixed windowed two-digit years, WW. Apart from YYYY and WW, it allows abstract, user-defined two-digit years. These are denoted U$(t)$, 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 pair with the built-in year types.

Finally, 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 Year-2000-safe library routines.

1.4 Related Work

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 [OJ97]. Van Deusen and Moonen [VDM98] 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.

The unification theory described in Section 3.2 is a special and feasible case of associative unification (word unification, solving of word equations) [AP89] that, however, does not seem to have been treated in the literature before. Independently of us, Ramalingam, Field and Tip have developed

\[
\begin{align*}
\text{prog} & ::= \text{dec stmt} \\
\text{dec} & ::= \epsilon \mid \text{dec dn:elem } \tau \mid \text{dec dn:group(dec)} \\
\text{stmt} & ::= \epsilon \mid \text{stmt stmt} \\
& \mid \text{PERFORM stmt UNTIL condexp} \\
& \mid \text{MOVE sender TO receiver} \\
& \mid \text{IF condexp stmt ELSE stmt ENDSIF} \\
\text{exp} & ::= \text{lit } | \text{pqn } | \text{assume exp is } \tau \\
& \mid \text{coerce exp by } c \mid \text{exp binop exp} \\
\text{sender} & ::= \text{exp } | \text{padl}_{(n,m)}(\text{exp}) \mid \text{padr}_{(n,m)}(\text{exp}) \\
& \mid \text{takel}_{(n,m)}(\text{exp}) \mid \text{taker}_{(n,m)}(\text{exp}) \\
\text{receiver} & ::= \text{pqn } | \text{assume receiver is } \tau \\
\text{pqn} & ::= \text{dn } | \text{dn IN pqn} \\
\text{binop} & ::= \ast \text{WN } | \ast \text{NW } | \ast \text{YN } | \ast \text{NN } \\
\tau & ::= \eta \mid \tau \tau \mid \epsilon \\
\eta & ::= \ast \text{W } | \ast \text{Y } | \ast \text{N } \mid \alpha^{(n)} \\
\text{condexp} & ::= \text{exp relop exp} \\
\text{relop} & ::= \ast \text{W } \mid \ast \text{Y } \mid \ast \text{N } \\
\end{align*}
\]

Figure 1: Grammar for COBOL2000.

basically the same unification algorithm [RFT99] as ours in Section 3.2. They also demonstrate how their analysis is applicable to Year 2000 program analysis.

1.5 Outline

In the remainder of the paper we first present the core theory of AnnoDomini: a skeletal language (Section 2), type system and type inference (Section 3), dynamic semantics and soundness (Section 4), and safety of expansion (Section 5). Then we describe how salient features of a full COBOL dialect are treated (Section 6), and we briefly describe the system architecture of AnnoDomini (Section 7). Finally we discuss the potential of applying the general principles embedded in AnnoDomini as a general software reengineering methodology (Section 8).

2 A Skeletal Language

Our skeletal language is called COBOL2000. While much simplified compared to full COBOL, it contains some of the key ingredients of COBOL, including moves involving padding and truncation of data.

We assume a set L of (decimal) numeric literals, ranged over by $\text{lit}$; a set $\text{dn}$ of data names, ranged over by $\text{dn}$, for each natural number $n \geq 1$, a set $\text{TyVar}_n^{(k)}$ of type variables of size $n$, ranged over by $\alpha^{(n)}$, and a set $C$ of coercion names, ranged over by $\text{cTYVar} = \bigcup_{n=1}^{\infty} \text{TyVar}_n^{(n)}$.

The grammar for COBOL2000 appears in Figure 1; an example program appears in Figure 2.

A program (prog) consists of a declaration (dec) followed by a statement (stmt). A declaration can be empty ($\epsilon$), or a declaration followed by a declaration of an elementary item

$^3$The COBOL term “data name” corresponds to “variable” in other programming languages.

$^7$Year 2000 unsafe years in the range 1900-1999 are considered a special case of windowed years with pivot 00.
YR1: elem WW
YR2: elem WW
MYDATE: elem WNNNNN
LONDATE: group(
  YEAR: elem YYYY
  MONTH: elem NN
  DAY: elem NN)
BUF: elem N(80)

MOVE assume 94 in WW TO YR1
COMPUTE YR2 = YR1 +WW 10
PERFORM
  COMPUTE YR1 = YR1 +WW 1
UNTIL YR2 <WW YR1
MOVE assume 940203 is WNN(4) TO MYDATE
MOVE coerce MYDATE by D2TONM TO BUF
IF YEAR IN LONDATE <YY assume 2000 is YYYY
  MOVE padl(8,72) (assume LONDATE is N(8)) TO BUF
ELSE MOVE padl(1,1)(5) TO DAY IN LONDATE
ENDIF

Figure 2: A COBOL-2000 program

or a group item. In COBOL-2000, all elementary items are numeric. A statement can be empty (e) or the sequential composition of two or more statements. Next, COMPUTE stmt until condsnap corresponds roughly to repeat stmt until condsnap in other languages. MOVE sender TO receiver moves the value of sender to receiver. A sender may be an expression or a padded or truncated expression. The sender padl(n,m)(exp) can only be used when exp is an elementary item; it pads the value of exp with n leading zeros. The sender padr(n,m)(exp) can only be used when exp is a group item; it pads the value of exp with m trailing zeros. The sender takei(n,m)(exp) can only be used when exp is a group item; it extracts the leftmost n digits of the value of exp. The sender takeor(n,m)(exp) can only be used when exp is an elementary item; it extracts the rightmost m digits of the value of exp.

Padding and truncation are implicit in COBOL, but it simplifies the formal treatment to make them explicit. The front end of a COBOL compiler can insert padding and truncation operations.

Inspired by COBOL syntax, we write COMPUTE receiver = sender for MOVE sender TO receiver, when sender is an arithmetic expression.

A perhaps qualified data name (pgdn) is either an unqualified data name dn or a qualified data name (dn IN pgdn). In the latter case, pgdn must denote a group item and dn a data name declared in that group item.

In AnnoDomini, addition and subtraction are overloaded. Overloading is resolved using an overloading scheme similar to the one used in some Standard ML compilers. For the purpose of the present paper, however, we assume that overloading has already been resolved, so in COBOL-2000 we annotate the arithmetic operators (and we only have one) by type information. Similarly for relational operators.

The assume and coerce constructs represent AnnoDomini's *TS2K annotations ASSUME and COERCE, respectively.

We impose the monoid laws on types: \( \eta \tau = \tau \epsilon \) and \( (\tau \gamma) \eta = (\eta \gamma) \tau \). Thus every type can be written in the form \( \eta_1 \ldots \eta_k \tau, k \geq 0 \). We write \( \tau(n) \) to mean \( \tau \ldots \tau \), \( n \) times.

The size of a type is defined inductively as follows:

\[
\begin{align*}
\text{size}(\eta) &= \text{size}(\tau) = \\
\text{case } \eta &\text{ of case } \tau \\
\eta \epsilon &= 0 \\
| \tau_{\eta} &= 1 \\
| \eta^* &= \text{size}(\eta) + \text{size}(\tau) \\
\alpha^n &= n
\end{align*}
\]

3 Type System 2000

COBOL-2000 has two kinds of data items: numeric elementary items and group items

\[ k \in \text{Kind} = \{ e, g \} \]

We assume a function, TypeOf, from coercion names to function types, which must all be monotones. (A function type is a type of form \( \tau' \rightarrow \tau'' \), and a monotype is a type that contains no type variables.) We define environment types (\( \sigma \)) and type environments (TE) as follows:

\[
\begin{align*}
\sigma &::= \tau | TE \\
TE &::= e | TE, dn : \sigma
\end{align*}
\]

Intuitively, a type environment is a finite map from (unqualified) data names to environment types. It maps data names that denote elementary items to types and it maps data names that denote group items to type environments.

Whenever \( A \) is a type or an object constructed out of types (e.g., a type environment), we write tyvars(\( A \)) for the set of type variables that occur in \( A \).

We define a function flat from type environments to types as follows:

\[
\text{flat}(TE) = \text{case } TE \text{ of } \\
\epsilon \Rightarrow e \\
| TE, dn : \tau \Rightarrow \text{flat}(TE) \tau \\
| TE, dn : TE' \Rightarrow \text{flat}(TE) \text{ flat}(TE')
\]

The type inference rules appear below. The turnstile (\( \vdash \)) is overloaded on different syntactic categories. There is a small number of inference rules (usually one) for each production in the grammar of COBOL-2000. Rules in which the premise appears to be identical to the conclusion stem from productions where the right-hand side consists of a single non-terminal.

**Perhaps Qualified Data Names**

\[
\begin{align*}
TE \vdash \text{dn : } \sigma \quad \text{dn } \neq \text{dn}' \\
TE, \text{dn : } \sigma' \vdash \text{dn : } \sigma
\end{align*}
\]

(1)

\[
\begin{align*}
TE, \text{dn : } \sigma \vdash \text{dn : } \sigma
\end{align*}
\]

(2)
Expressions

\[
\frac{TE \vdash \text{exp} : \tau \cdot k}{TE \vdash \text{sender} : \tau}
\]

\[
\frac{\tau = \text{N(size(list))}}{TE \vdash \text{lit} : \tau \cdot e}
\]

\[
\frac{TE \vdash \text{pqn} : \tau}{TE \vdash \text{pqdn} : \tau \cdot e}
\]

\[
\frac{TE_1 \vdash \text{pqdn} : TE_2 \quad TE_1 \vdash \text{pqdn} : \text{flat}(TE_2) : g}{TE_1 \vdash \text{pqdn} : \text{flat}(TE_2) : g}
\]

\[
\frac{TE \vdash \text{exp} : \tau' \cdot k \quad \text{size}(\tau) = \text{size}(\tau')}{TE \vdash \text{assume exp is } \tau : \tau' \cdot k}
\]

\[
\frac{TE \vdash \text{exp} : \tau \cdot k \quad \text{TypeOf}(\text{c}) = \tau \rightarrow \tau'}{TE \vdash \text{coerce exp by } \text{c} : \tau' \cdot e}
\]

\[
\frac{TE \vdash \text{exp}_1 : \tau_1 : e \quad TE \vdash \text{exp}_2 : \tau_2 : e \quad \tau_1, \tau_2 \vdash \text{binop} : \tau_3}{TE \vdash \text{exp}_1 \text{ binop} \text{ exp}_2 : \tau_3 : e}
\]

Statements

\[
\frac{TE \vdash \text{sender} : \tau \quad TE \vdash \text{receiver} : \tau}{TE \vdash \text{MOVE sender TO receiver}}
\]

\[
\frac{TE \vdash \text{conexp} \quad TE \vdash \text{stmt}_1 \quad TE \vdash \text{stmt}_2}{TE \vdash \text{IF conexp stmt}_1 \text{ ELSE stmt}_2 \text{ ENDIF}}
\]

\[
\frac{TE \vdash \text{stmt} \quad TE \vdash \text{conexp} \quad \text{TE \ PERFORM stmt UNTIL conexp}}{TE \vdash \epsilon}
\]

\[
\frac{TE \vdash \text{stmt}_1 \quad TE \vdash \text{stmt}_2}{TE \vdash \text{stmt}_1 \text{ stmt}_2}
\]

Declarations

\[
\frac{\vdash \text{dec} : \text{TE}}{\vdash \epsilon : \epsilon}
\]

\[
\frac{\vdash \text{dec} : \text{TE}}{\vdash \text{dec} \ dn : \text{elem } \tau : TE, dn : \tau}
\]

\[
\frac{\vdash \text{dec}_1 : \text{TE}_1 \quad \vdash \text{dec}_2 : \text{TE}_2}{\vdash \text{dec}_1 \ dn : \text{group} (\text{dec}_2) : \text{TE}_1, dn : \text{TE}_2}
\]

Programs

\[
\frac{\vdash \text{dec} : \text{TE} \quad TE \vdash \text{stmt}}{\vdash \text{dec stmt}}
\]

Binary Operators

\[
\frac{\tau_1, \tau_2 \vdash \text{binop} : \tau_3}{\text{WW, N(i)} \vdash \text{+WN : WW}}
\]

\[
\frac{\text{YYY, N(i)} \vdash \text{YN : YYY}}{\text{N(i), N(j)} \vdash \text{+NN : N(1 + max(i, j))}}
\]

\[
\frac{\text{N(i), WW} \vdash \text{+NW : WW}}{\text{N(i), WW} \vdash \text{+NW : WW}}
\]
$$\overline{N(i), YYYY} \vdash +_{NY} : YYYY$$  (31)

**Conditional Expressions**

$$TE \vdash \text{condexp}$$

$$\begin{align*}
TE \vdash \text{exp}_1 : \tau_1 :: \epsilon & \quad & TE \vdash \text{exp}_2 : \tau_2 :: \epsilon & \quad & \tau_1, \tau_2 \vdash \text{relop} \\
TE \vdash \text{exp}_1, \text{relop} \text{exp}_2
\end{align*}$$  (32)

**Relational Operators**

$$\tau_1, \tau_2 \vdash \text{relop}$$

$$\overline{WW, WW} \vdash \leq_{WW}$$  (33)

$$\overline{YYYY, YYYY} \vdash \leq_{YY}$$  (34)

$$\overline{N(i), N(j)} \vdash \leq_{SN}$$  (35)

Rules 1–3 describe how perhaps qualified data names are looked up in the type environment. In rule 3, the first premise expresses that $p$ in $\text{dn}$ must be bound to a type environment ($TE'$), i.e., it must denote a group item, and the second premise expresses that $dn$ must be declared in this group item.

The rules for expressions (4–9) allow one to infer statements of the form $TE \vdash \text{exp} : \tau :: k$, read: in $TE$, $\text{exp}$ has type $\tau$ of kind $k$. The kind determines how the value of $\text{exp}$ may be padded, as explained later.

In rule 4, we use the notation $\text{size}(\text{lit})$, by which we mean the number of (decimal) digits in it, including any leading zeros.

Rule 5 gives kind $\epsilon$ to elementary items, while rule 6 gives kind $g$ to group items.

Rule 7 shows that applying $\text{assume}$ to an expression makes it possible to change the type of the expression to any other type with the same size; the kind is preserved.

Rule 8 shows that $\text{coerce}$ can be applied to an expression of either kind (elementary or group). Somewhat arbitrarily, we say that the result is always an elementary item.

Rule 9 concerns binary operators; the relation $\tau_1, \tau_2 \vdash \text{binop} : \tau$ holds if $\tau_1$ and $\tau_2$ are legal types of the left and right arguments of $\text{binop}$, respectively, and $\tau$ is the type of the result of applying $\text{binop}$; see rules 27–31.

The rules for senders (10–14) allow one to infer statements of the form $TE \vdash \text{sender} : \tau$, read: in $TE$, $\text{sender}$ has type $\tau$. Note that the sender does not have a kind; expressions have a kind, however, and this kind is used for determining how the expression can be padded in order to construct a sender. Rule 10 is the case where the expression is neither padded nor truncated; the kind is simply discarded.

Rule 11 concerns the case where a value is padded with $n$ zeros on the left; note that the expression must have kind $\epsilon$, but the zeros may be given any type of size $n$ (including types involving years).

Padding on the right (rule 12) requires group kind; truncation which discards rightmost digits requires group kind (rule 13); and truncation which discards leftmost digits requires kind $\epsilon$ (rule 14).

The identifier which is assigned a value in the $\text{MOVE}$ statement is called a receiver. Receivers are typed much like perhaps qualified data names in expressions, except that no kind is needed (rules 15–17).

In rule 18, notice that the sender and the receiver must have exactly the same type. The sender must be padded or truncated, if necessary, to achieve this.

The rules for binary operators (27–31) impose a certain discipline on the use of arithmetic. Assume, for example, that $\text{MYYEAR}$ and $\text{MYDATE}$ have types $\overline{WW}$ and $\overline{WNNNN}$, respectively. Then $\text{COMPUTE MYYEAR} = \text{MYYEAR} + 1$ is well-typed, but $\text{COMPUTE MYDATE = MYDATE+10000}$ (a common COBOL idiom!) is not. Similarly,

\begin{verbatim}
LONGDATE : group(
    YEAR : YYYY
    MONTH : NN
    DAY : NN)
SHORTDATE : WNNNN
MOVE LONGDATE TO SHORTDATE
\end{verbatim}

is not well-typed, although COBOL permits it: the types $YYYYNNN$ and $WNNNN$ are not equal, not even after any padding or truncation (compare rule 18). Instead, Type System 2000 provides coercions for conversions between different date formats. In Figure 2, D2TO4H4 is a coercion of type $WNNNN \rightarrow YYYYNNN$.

Rules 33–35 show a rather limited set of comparisons. AnnoDomini supports a wider range, including, for example, comparison on types $WNNNN$.

An expression is well-typed in $TE$, if there exists a type $\tau$ and a kind $k$ such that $TE \vdash \text{exp} : \tau :: k$. Well-typedness of other phrase forms in a type environment is defined similarly.

### 3.1 Type Checking

By type checking we mean using an algorithm, called the type checker, which checks, given a type environment and a phrase, whether there exists a substitution instance of the type environment in which the phrase is well-typed.

As is the case with Standard ML, type checking is based on unification [Mil78, DM82]. However, because our types obey the monoid laws and because type variables have sizes associated with them, ordinary term unification is not a sufficient base for type checking in COBOL2000. We therefore develop a different unification algorithm and prove that there exists a form of most general unifier in this setting. The fact that types have known sizes is essential for this result, which appears to be new.\(^9\)

### 3.2 Unification

A substitution is a map $S$ from type variables to types satisfying

$$\forall \alpha^{(n)} \in TVar^{(n)}, \text{size}(S(\alpha^{(n)})) = n$$

The support of $S$, written $\text{Supp}(S)$, is the set

$$\text{Supp}(S) = \cup_{n \geq 1} \{ \alpha^{(n)} \mid S(\alpha^{(n)}) \neq \alpha^{(n)} \}$$

The yield of $S$, written $\text{Yield}(S)$, is defined by

$$\text{Yield}(S) = \bigcup \{ \text{tyvars}(S(\alpha^{(n)})) \mid \alpha^{(n)} \in \text{Supp}(S) \}$$

\(^9\) It turns out that Ramalingam, Field, and Tip independently have developed essentially the same algorithm, see elsewhere in these proceedings.
Further, the set of type variables involved in $S$, written $\text{Inv}(S)$ is defined by

$$\text{Inv}(S) = \text{Supp}(S) \cup \text{Yield}(S)$$

By natural extension, substitutions may be applied to types and to objects constructed out of types (for example programs). Substitution preserves equality: if two types $\tau_1$ and $\tau_2$ are equal under the monoid laws, then $S(\tau_1)$ and $S(\tau_2)$ are also equal under the monoid laws. Substitution preserves size.

**Lemma 3.1**
For all $\eta$ and $\tau$, $\text{size}(S(\eta)) = \text{size}(\eta)$ and $\text{size}(S(\tau)) = \text{size}(\tau)$.

Substitutions compose: $(S_2 \circ S_1)(\alpha^{(n)}) = S_2(S_1(\alpha^{(n)}))$. Note that, by Lemma 3.1, the composition is a substitution. The identity substitution is denoted $I$.

A unifier for types $\tau_1$ and $\tau_2$ is a substitution $S$ satisfying $S(\tau_1) = S(\tau_2)$. A substitution $S'$ is a most general unifier for $\tau_1$ and $\tau_2$ if $S'(\tau_1) = S'(\tau_2)$ and for every unifier $S$ for $\tau_1$ and $\tau_2$ there exists an $S'$ with $S' \circ S'$. Interestingly, not all pairs $(\tau_1, \tau_2)$ have a most general unifier. For example, let $\tau_1 = \alpha_1^{(4)} \alpha_2^{(2)}$ and $\tau_2 = \alpha_3^{(2)} \alpha_4^{(2)}$. In order to unify these two types, without committing the two middle digits of the six digits to a monotype, we introduce a fresh type variable, say $\alpha_5^{(2)}$. Consider the "natural" candidate for a most general unifier, namely

$$S^* = \{ \alpha_1^{(4)} \rightarrow \alpha_3^{(2)} \alpha_4^{(2)}, \alpha_4^{(4)} \rightarrow \alpha_5^{(2)} \alpha_2^{(2)} \}$$

We have $S^*(\tau_1) = \alpha_5^{(2)} \alpha_3^{(2)} \alpha_2^{(2)} = S^*(\tau_2)$, so $S^*$ unifies $\tau_1$ and $\tau_2$. However, $S^*$ is not a most general unifier. To see this, consider

$$S = \{ \alpha_1^{(4)} \rightarrow \alpha_3^{(2)} \alpha_4^{(4)}, \alpha_4^{(4)} \rightarrow \alpha_5^{(2)} \alpha_2^{(2)} \}$$

Note that $S$ is a unifier for $\tau_1$ and $\tau_2$; but there exists no $S'$ such that $S = S' \circ S^*$ (consider what $\alpha_5^{(2)}$ should be mapped to); the best we can achieve is $S \downarrow U = (S' \circ S^*) \downarrow U$, where $U$ is a set of type variables satisfying $\alpha_5^{(2)} \notin U$. (We write $S \downarrow U$ for the restriction of $S$ to $U$.)

To make this precise, we introduce a notion called $U$-unification. We use $U$ to range over finite subsets of $\text{TyVar}(U)$ stands for "used"). A substitution $S^*$ is a most general $U$-unifier for $\tau_1$ and $\tau_2$, if $\text{tyvars}(\tau_1) \cup \text{tyvars}(\tau_2) \subseteq U$; $S^*(\tau_1) = S^*(\tau_2)$; and for all substitutions $S$, if $S(\tau_1) = S(\tau_2)$ then there is a substitution $S'$ with $S \downarrow U = (S' \circ S^*) \downarrow U$.

We now present a unification algorithm and a theorem to the effect that the algorithm finds most general $U$-unifiers.

The algorithm relies on the fact that every type can be written in the form $\eta_1 \ldots \eta_k$, $(k \geq 0)$. The algorithm is:

```
f n unif y(\eta_1, \eta_2, U) = case \eta_1 \ldots \eta_k of
| (\epsilon, \epsilon) \Rightarrow (I, \emptyset)
| (\eta_1, \emptyset) \Rightarrow \text{case } \eta_2 \text{ of }
| \text{if } \text{size}(\eta_1) > \text{size}(\eta_2) \text{ then }
| \text{else if } \text{size}(\eta_1) = \text{size}(\eta_2) \text{ then }
| \text{let } S_1 = \text{unify}_y(\eta_1, \eta_2)
| \left( (S_1, U_1) = \text{unify}(S_1(\tau_1), S_1(\tau_2), U) \text{ in } (S_2 \circ S_1, U_2) \text{ end }
| \text{else if } \text{size}(\eta_1) > \text{size}(\eta_2) \text{ then }
| \text{let } S_1 = \text{split}(\eta_1, \text{size}(\eta_2), U)
| \text{let } S_2 = \text{unify}(S_1(\tau_1), S_1(\tau_2), U \cup U_1)
| \left( (S_2, U_3) = \text{unify}(S_1(\tau_1), S_1(\tau_2), U \cup U_1) \text{ in } (S_2 \circ S_1, U_2) \text{ end }
| \left( \right) \Rightarrow \text{raise } \text{FAIL}
```

**Theorem 3.1**
Assume $\text{tyvars}(\tau_1) \cup \text{tyvars}(\tau_2) \subseteq U$. If there exists a unifier for $\tau_1$ and $\tau_2$ then $S = \text{unify}_y(\eta_1, \eta_2)$ succeeds and $S$ is a most general unifier for $\eta_1$ and $\eta_2$. If there exists no unifier for $\eta_1$ and $\eta_2$, then $\text{unify}_y(\tau_1, \tau_2)$ raises $\text{FAIL}$.

**3.3 A Type Checker**
We now present a type checker which is sound and complete for the type inference rules in Section 3. The soundness relies on the following result:

**Lemma 3.3**
$A \vdash B$ implies $S(A) \vdash S(B)$.

For brevity, we show only the part of the type checker that is necessary for type checking expressions and phrases that are part of expressions. (The remaining rules are treated analogously.) Concerning rules 1–3, let $\text{TE}$ be a type environment and let $y$ be a perhaps qualified data name. Note that there exists at most one $\sigma$ such that $\text{TE} + y : \sigma$. Define lookup($\text{TE}, y$) to be this $\sigma$, if it exists, and let lookup($\text{TE}, y$) terminate with $\text{FAIL}$ otherwise.

Next, rules 27–31 are checked by the following function:

```
\text{BinOp}(\tau_1, \tau_2, \text{binop}, U) =
\text{case } \text{binop of }
| \text{*} \Rightarrow \text{let } (S_1, U_1) = \text{unify}(\tau_1, \text{W}, U)
| \left( (S_2, U_3) = \text{unify}(S_1(\tau_2), \text{N(size(\tau_2)), U \cup U_1}) \text{ in } (S_2 \circ S_1, \text{W}, U_1 \cup U_2) \text{ end }
| \left( \right) \Rightarrow \text{raise } \text{FAIL}
```

For example:

```

```
\[
S_{0}(TE) \vdash S_{0}(\exp) : \tau_{0} \vdash k_{0}
\]

then
\[
(S', \tau', k', U') = \text{Exp}(TE, \exp, U)
\]
succeeds and for all \(S_{0}, \tau_{0}\) and \(k_{0}\) satisfying (39) there exists an \(S''\) such that \(S_{0} \downarrow U = (S'' \circ S') \downarrow U, S''(\tau') = \tau_{0}\) and \(k_{0} = k'\). Moreover, if there is no \(S_{0}, \tau_{0}\) and \(k_{0}\) satisfying (39), then (40) terminates with \text{FAIL}.

The proofs are by induction on the structure of \(\exp\), using Lemmas 3.4, 3.5.

4 Dynamic Semantics and Soundness

In this section we give a dynamic semantics for COBOL_2000 and prove that the static semantics is sound with respect to the dynamic semantics.

We use \(d\) to range over decimal digits. We distinguish between numerals (\(\text{a}\)) and numbers (\(a\)). A value, \(v\), is a numeral, i.e., a sequence of digits. The size of a value \(a\), written \(\text{size}(a)\), is the number of digits in \(a\). \(0(n)\) denotes the sequence of \(n\) zeros. For every value \(v\) and \(n \geq 0\) satisfying \(\text{size}(r) \leq n\), we define \(\text{fill}(n, v)\) to be \(0(n)\).

Evaluation can fail. The semantics distinguishes between the following failures: \text{Y2K} (Year 2000 failure), \text{Arith} (arithmetic failure), and \text{Wrong} (other failure).

A store (of size \(n\)) is a map from \(\{0, \ldots, n-1\}\) to digits. We use \(s\) to range over stores. We define \(\text{extract}(s, l, u)\) to be the value \(s(l) \cdots s(u-1)\), if \(0 \leq l \leq u \leq \text{size}(s)\), and the failure \text{Wrong} otherwise. Further, we define \(\text{update}(s, l, u, v)\) to be the store \(s'\) defined by
\[
s'(i) = \begin{cases} s(i) & \text{if } 0 \leq i < l \text{ or } u \leq i \leq \text{size}(s) \\ d_{l-1} & \text{if } i < u \end{cases}
\]

provided \(0 \leq l \leq u \leq \text{size}(s)\) and \(v = d_{0} \cdots d_{u-1}\); otherwise, we define \(\text{update}(s, l, u, v)\) to be the failure \text{Wrong}.

We define (dynamic) environments (\(E\)) and denotable values (\(\text{dval}\)) as follows:
\[
E ::= e \mid E, dn = \text{dval}
\]
\[
\text{dval} ::= (E, l, u)
\]

where \(l\) and \(u\) range over non-negative integers. Intuitively, given store \(s\), \(E(dn) = (E', l, u)\) means that \(dn\) represents the value \(s(l) \cdots s(u-1)\). Further, \(E'\) is empty, if \(dn\) is an elementary item.

The evaluation relation takes the form \(A \vdash \text{phrase} \Rightarrow B\), where \(A\) is an environment or a pair of an environment and a store and \(B\) is a result, whose kind depends on the kind of phrase. An environment result \(e\) is either a denotable value of \text{Wrong}. An expression result \(\tau\) is either a value, \text{Y2K}, \text{Wrong}, or \text{Arith}. A statement result \(t\) is either a store, \text{Y2K}, \text{Wrong}, or \text{Arith}. In the evaluation of relational expressions, we use booleans (\text{true}, \text{false}); a condition result \(b\) is either \text{true}, \text{false}, \text{Wrong}, \text{Arith} or \text{Y2K}.

We assume a function \(\text{Apply}\) with the following property: for all \(c\), letting \(\tau \rightarrow \tau' = \text{TypeOf}(c)\), for all \(v\), if \(\text{size}(\tau) = \text{size}(v)\) then \(\text{Apply}(c, v)\) either results in \text{Y2K} or in a value \(v'\) of size \(\tau'\).

Finally, \(\text{initstore}(z)\) denotes a store of size \(z\) consisting of zeros only.

The dynamic semantics is defined as a “big step” semantics, with a small number of rules for each production in the grammar. The rules appear below. Many of the rules have associated with them one, two or three failure propagation...
rules, which describe that once the evaluation of a phrase fails, the evaluation of the entire program fails. For example, rule 48 has the following two associated failure rules:

\[
E \vdash pqdn \Rightarrow f \\
s, E \vdash pqdn \Rightarrow f
\]

\[
E \vdash pqdn \Rightarrow (\_ , l , u) \quad \text{Wrong = extract}(s , l , u) \\
s, E \vdash pqdn \Rightarrow \text{Wrong}
\]

where \( f \) ranges over \{\text{Wrong, Y2K, Arith}\}. For brevity, we omit the failure propagation rules (although they are used in the proof of Theorem 4.1.)

**Perhaps Qualified Data Names**

\[ E \vdash pqdn \Rightarrow e \]

\[
e \vdash pqdn \Rightarrow \text{Wrong}
\]

**Expressions**

\[ s, E \vdash list \Rightarrow list \]

\[ E \vdash pqdn \Rightarrow (\_ , l , u) \quad v = \text{extract}(s , l , u) \\
s, E \vdash pqdn \Rightarrow v \]

\[ s, E \vdash \text{assume } \text{exp} \Rightarrow \tau : v \]

\[ s, E \vdash \text{exp} \Rightarrow v \\
\text{Apply}(c , v) = v' \\
s, E \vdash \text{coerce } \text{exp} \text{ by } c : v' \]

\[ s, E \vdash \text{exp} \Rightarrow v_1 \\
s, E \vdash \text{exp} \Rightarrow v_2 \\
v_1 , v_2 \vdash \text{binop} \Rightarrow v \\
s, E \vdash \text{binop } \text{exp} \_1 \Rightarrow \text{exp} \_2 \Rightarrow v \]

\[ s, E \vdash \text{sender} \Rightarrow r \]

**Receivers**

\[ E \vdash \text{receiver} \Rightarrow (l , u) / \text{Wrong} \]

\[
s, E \vdash \text{receiver} \Rightarrow (l , u) \\
E \vdash \text{receiver} \Rightarrow (l , u)
\]

**Statements**

\[ s, E \vdash \text{stmt} \Rightarrow t \]

\[
s, E \vdash \text{sender} \Rightarrow v \\
s, E \vdash \text{receiver} \Rightarrow (l , u) \\
s, E \vdash \text{MOVE } \text{sender TO } \text{receiver} \Rightarrow \text{update}(s , l , u , v)
\]

\[ s, E \vdash \text{conexp} \Rightarrow \text{true} \\
s, E \vdash \text{stmt} \_1 \Rightarrow s' \\
\]

**Declarations**

\[ l \vdash \text{dec} \Rightarrow E , z \]

\[ l \vdash \text{epsilon} \Rightarrow \epsilon , 0 \]
\[ l \vdash \text{dec}_1 \Rightarrow E_1, z_1 \]
\[ l \vdash \text{dec}_1 \text{ dn: elem } \tau \Rightarrow E_1, \text{dn} = (\epsilon, l + z_1, l + z_1 + \text{size}(\tau)), \]
\[ z_1 + \text{size}(\tau) \]

(67)

\[ l \vdash \text{dec}_1 \Rightarrow E_1, z_1 \]
\[ l \vdash \text{dec}_2 \Rightarrow E_2, z_2 \]
\[ l \vdash \text{dec}_1 \text{ group(dec2)} \Rightarrow E_1, \text{dn} = (E_2, l + z_1, l + z_1 + z_2), \]
\[ z_1 + z_2 \]

(68)

Programs

\[ \vdash \text{prog} \Rightarrow s \]

(69)

\[ 0 \vdash \text{dec} \Rightarrow E, z \]
\[ \text{initstore}(z), E \vdash \text{stmt} \Rightarrow s \]
\[ \vdash \text{dec stmt} \Rightarrow s \]

Binary Operators

\[ v, v \vdash \text{binop} \Rightarrow r \]

(70)

\[ a + b < 99 \]
\[ a \vdash b \Rightarrow \text{fill}(2, a + b) \]

(71)

\[ a + b > 999 \]
\[ a \vdash b \Rightarrow \text{Y2K} \]

(72)

\[ a + b < 9999 \]
\[ a \vdash b \Rightarrow \text{fill}(4, a + b) \]

(73)

\[ a + b > 9999 \]
\[ a \vdash b \Rightarrow \text{Arith} \]

(74)

Conditional Expressions

\[ s, E \vdash \text{condexp} \Rightarrow b \]

(75)

Relational Operators

\[ v, v \vdash \text{rel} \Rightarrow b \]

(76)

\[ a \leq 99 \]
\[ b \leq 99 \]
\[ a < b \]
\[ \vdash b \Rightarrow \text{true} \]

\[ a \leq 9999 \]
\[ b \leq 9999 \]
\[ a < b \]
\[ \vdash b \Rightarrow \text{true} \]

\[ \vdash b \Rightarrow \text{false} \]

(77)

\[ a < b \]
\[ \vdash b \Rightarrow \text{true} \]

(78)

\[ a < b \]
\[ \vdash b \Rightarrow \text{false} \]

(79)

4.1 Soundness

A store typing, \( ST \), is a sequence of triples

\[(l_1, u_1, \tau_1) \ldots (l_n, u_n, \tau_n), \quad n \geq 0\]

satisfying

1. \( \text{size}(\tau_i) = u_i - l_i \), for all \( i = 0 \ldots n \); and
2. \( l_i = u_{i-1} \), for all \( i = 1 \ldots n \).

The address space of \( ST \), written \( \text{addr}(ST) \), is the interval \( \{i \mid l_i \leq i < u_n \} \). Given \( l \) and \( u \), we define the store type of \((ST, l, u)\), written \( \text{sttyp}(ST, l, u) \), to be \( \tau_1 \ldots \tau_{n-i} \).

We say that an environment \( E \) matches type environment \( TE \) in \( ST \), if \( ST \models E : TE \), where \( ST \models E : TE \) is defined thus:

Store Typing

\[ ST \models E : TE \]

(1) \[ ST \models \epsilon : \epsilon \]
\[ ST \models E, \text{dn} = (\epsilon, l, u) : TE, \text{dn} : \tau \]

The following soundness theorem states that if an expression is well-typed and is evaluated in an environment and a store that match the type environment, then the result of the evaluation is not \text{Wrong}, although it may be a Year 2000 failure. Furthermore, if the result is not a failure, it is a value of the size predicted by the type system.

Theorem 4.1 If \( E \vdash \text{exp} : \tau :: k \) and \( ST \models E : TE \) and \( \text{Dom}(s) = \text{addr}(ST) \) and \( s, E \vdash \text{exp} \Rightarrow r \) then \( r \) is not \text{Wrong} (although it may be \text{Y2K}) and if \( r \) is a value \( v \) then \( \text{size}(v) = \text{size}(\tau) \).

5 Expansion

In this section we formalize transformations that fix Year 2000 problems using expansion and show that, informally speaking, all computations that succeed using the original program will still succeed (with a related result) after expansion. A similar result is difficult to obtain for windowing, for as one moves a window, some Year 2000 problems are fixed while others may be introduced (because the new window does not represent all the years that could be represented with the old window).

To prove our result, we shall interpret \( \text{WWW} \) as representing the years 1900–1999 by their last two digits. Further, we consider only coercions which expand two-digit years to four digit years (by adding 1900).

Formally, we introduce a subtyping relation on types and type environments, written \( \eta \leq \eta' \), \( \tau \leq \tau' \), and \( TE \leq TE' \), respectively, as the smallest reflexive and transitive relation satisfying:

1. \( \text{WWW} \leq \text{YYYY} \).
Informally, the theorem states that a translated expression returns the year expanded value of the original expression, if the translated expression is evaluated in a year expanded version of the store and environment that the original expression is evaluated in.

**Data Names**

\[ TE \vdash \mathit{pgdn} : \sigma \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \sigma' \]

\[ \frac{\varepsilon \neq \varepsilon'}{\mathit{TE} \vdash \varepsilon} \quad \frac{\mathit{TE} \vdash \varepsilon : \sigma \leadsto \mathit{TE}' \vdash \varepsilon : \sigma' \quad \sigma_1 \leq \sigma_1'}{\mathit{TE} \vdash \varepsilon : \sigma \leadsto \mathit{TE}' \vdash \varepsilon : \sigma' \leadsto \mathit{TE}' \vdash \varepsilon : \sigma'} \] (82)

\[ \frac{\mathit{TE} \vdash \mathit{pgdn} : \sigma \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \sigma'}{\mathit{TE} \vdash \mathit{pgdn} : \sigma \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \sigma'} \] (83)

\[ \mathit{TE} \vdash \mathit{pgdn} : \sigma \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \sigma' \]

\[ \frac{\mathit{TE} \vdash n : \mathit{size}(\mathit{lit}) \quad \mathit{TE} \vdash \mathit{lit} : \tau \leadsto \mathit{TE}' \vdash \mathit{lit} : \tau \leadsto \varepsilon}{\mathit{TE} \vdash \mathit{pgdn} : \tau \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \tau'} \] (86)

\[ \mathit{TE} \vdash \mathit{pgdn} : \tau \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \tau' \]

\[ \mathit{TE} \vdash \mathit{pgdn} : \sigma \leadsto \mathit{TE}' \vdash \mathit{pgdn} : \sigma' \]

\[ \frac{\mathit{TE} \vdash n \cdot \mathit{size} (\mathit{TE} _ 2) : \mathit{g} \leadsto \mathit{TE} \vdash \mathit{pgdn} : \mathit{flat} (\mathit{TE} _ 2) : \mathit{g}}{\mathit{TE} \vdash \mathit{pgdn} : \mathit{flat} (\mathit{TE} _ 2) : \mathit{g} \leadsto \mathit{TE} \vdash \mathit{pgdn} : \mathit{flat} (\mathit{TE} _ 2) : \mathit{g}} \] (87)

\[ \mathit{TE} \vdash \mathit{exp} : \tau : \mathit{k} \leadsto \mathit{TE}' \vdash \mathit{exp} : \tau' : \mathit{k}' \]

\[ \mathit{TE} \vdash \mathit{assume} \mathit{exp} _ 1 : \tau : \mathit{k} \leadsto \mathit{TE}' \vdash \mathit{assume} \mathit{exp} _ 1 : \tau : \mathit{k}' \]

\[ \mathit{TE} \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau : \mathit{e} \leadsto \mathit{TE}' \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau' : \mathit{e} \]

\[ \mathit{TE} \vdash \mathit{exp} : \tau : \mathit{k} \leadsto \mathit{TE}' \vdash \mathit{exp} : \tau' : \mathit{e} \]

\[ \mathit{TE} \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau' : \mathit{e} \]

\[ \mathit{TE} \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau : \mathit{e} \leadsto \mathit{TE}' \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau : \mathit{e} \]

\[ \mathit{TE} \vdash \mathit{exp} : \tau : \mathit{k} \leadsto \mathit{TE}' \vdash \mathit{exp} : \tau' : \mathit{k}' \]

\[ \mathit{TE} \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau' : \mathit{e} \] (90)

\[ \mathit{TE} \vdash \mathit{exp} : \tau : \mathit{k} \leadsto \mathit{TE}' \vdash \mathit{exp} : \tau' : \mathit{k}' \]

\[ \mathit{TE} \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau : \mathit{e} \leadsto \mathit{TE}' \vdash \mathit{coerce} \mathit{exp} : \mathit{c} : \tau : \mathit{e} \] (91)
The only interesting rules are the ones involving 
associative and commutative. In rule 88, note that the type of exp₂ must be the same before and after transformation and that the type of the whole expression must remain unchanged too. This is used in an essential way in the proof of Theorem 5.1. In rule 89, note that we can only keep the coercion, if the type of exp' equals the type of exp. Rule 90 says that a coercion in the original program may be cancelled by an expansion in the resulting program. Such an expansion can come about for example because exp is a data name dn with TE(dn) = WW but TE'(dn) = WYY. Rule 91 says that one can add a coercion in the resulting program.

One can prove Theorem 5.1 using the following lemmas.

Lemma 5.1 If \( TE \downarrow \exp \cdot \tau \vdash k \sim TE' \downarrow \exp' \cdot \tau' \vdash \tau' \cdot k' \) then \( TE \downarrow \exp \cdot \tau \vdash \tau \cdot k \) and \( TE' \downarrow \exp' \cdot \tau' \vdash \tau' \cdot k' \).

Lemma 5.2 If \( TE \downarrow \exp \cdot \tau \vdash k \sim TE' \downarrow \exp' \cdot \tau' \vdash \tau' \cdot k' \) then \( TE \leq TE' \), \( \tau \leq \tau' \), and \( k = k' \).

Lemma 5.3 If \( \tau_1 \leq \tau_2 \leq \tau_3 \) and \( \text{size}(\tau_1) = \text{size}(\tau) \) then \( \text{expand}(\tau_2, \tau_3, \text{expand}(\tau_1, \tau_2, \tau)) = \text{expand}(\tau_1, \tau_3, \tau) \).

6 Expansion to Full COBOL

The conceptual and technical core of AnnoDomini's type system has been described in Sections 3, 4, and 5. In this section we describe how AnnoDomini addresses a number of COBOL features not found in COBOL 2000, but of relevance to Year 2000 remediation.

6.1 Storage model

Data representation, alignment, synchronization, truncation, padding, and editing rules for COBOL are very complex. Here we shall give only a glimpse of how they are modeled in AnnoDomini. For a more thorough description we refer to the OS/VS COBOL Application Programmer's Reference [IBM86], AnnoDomini User's Guide [Ha89b] and AnnoDomini Reference Manual [Ha89a].

6.1.1 Internal representation

In the standard representation of data in COBOL each character position in an elementary data item requires one byte of storage. In this representation, even numeric data are stored in the form of contiguous decimal digits, e.g., the numeral '98' is stored in two contiguous bytes, the first of which contains the EBCDIC representation for the character '9', and the second contains the corresponding representation for '8'.

Numeric data, however, can also be stored in binary or packed form. In COBOL, there is a difference between moving a binary or packed item by directly referring to it or by moving it as part of a group item that contains it. In the first case the item is first transformed into its standard representation before its value is moved. In the latter case the item is moved in its binary or packed form as part of the group item, without prior transformation.

AnnoDomini models binary and packed representations by adding\[
\eta := \mathbb{B} \circ \tau \ | \ P \circ \tau > \]
to the language of TS2K type expressions and reflecting their implicit transformations in the type inference rules.

6.1.2 Editing

COBOL specifies the format of elementary data items by a number of clauses. The picture-string clause determines the class of allowable characters for each character position of a data item in standard representation. In particular, it determines the information-carrying character positions and the editing character positions in the data item. Data from another data item is written only into the information-carrying character positions. The editing positions are filled independently. For example, consider\[+TS2K \quad \text{WNNNNN}
\]
\[01 \quad \text{DATE-1  PI\C XXXXX VALUE IS '981106' .}
\[01 \quad \text{DATE-2 PI\C XX/XX/XX.}
\]

MOVE DATE-1 TO DATE-2

The two occurrences of '/' in the declaration for DATE-2 are editing characters. They are, in a sense, hardwired into 2 of the 8 bytes at the positions indicated. During the MOVE the six digits '981106' are moved into the remaining character positions. As a result, DATE-2 contains the string '98/11/06'.

AnnoDomini keeps track of storage at the byte-level and captures faithfully the effect of editing. In particular, it infers the type 'WNNNNNN' for DATE-2, where the 1st and the 4th N are derived from the editing for DATE-2. The other N's stem from the type for DATE-1.

6.1.3 Data alignment

Numeric data are aligned on an assumed decimal point before they are moved. We have generalized the notion of assumed decimal point to a reference point, which also applies to nonnumeric data. The reference point indicates where the data should be logically aligned and, if necessary, where padding and truncation is necessary during a MOVE.

AnnoDomini models each data item by \((k, \tau_1, \tau_2, \tau)\), where \(k, \tau\) correspond to \(\tau \vdash k\) in Section 5, and \(\tau_1\) and \(\tau_2\) describe the type information of the information-carrying character positions to the left and to the right of the data item's reference point, respectively. The 4-tuple contains sufficient information to determine how the bytes of a data item are processed both as a sender and as receiver in a MOVE statement.

6.2 Aliasing

COBOL provides a number of facilities for accessing the same storage using multiple data item definitions.

Explicit redefinition. A REDEFINES clause in the working storage section of a COBOL program expresses that the storage area associated with a data item definition can be accessed using another data item definition.
Implicit redefinition. Multiple 01-level record definitions for a single file are implicit redefinitions for the file buffer area shared by all records from the file.

Renamings. Consecutive storage inside a single 01-level storage can be can be (re)named by a RENAMES clause and accessed through it.

These forms of aliasing are usually used for two distinct purposes:

1. to provide different views of the same data;
2. to reuse storage area for efficiency purposes or to process different records contained in the same file.

In the first case the program interleaves reads from and writes to the shared storage area using different views.

For example, in

```
01 DATE-1 PIC X(6).
01 DATE-2 REDEFINES DATE-1.
  02 YY PIC 99.
  02 MM PIC 99.
  02 DD PIC 99 VALUE IS '31'.
```

MOVE '981106' TO DATE-1.
DISPLAY DD OF DATE-2.

the program writes ‘981106’ into the shared storage area using the view of DATE-1, and consequently reads it from the same storage area using the view of DATE-2. (Thus the program displays ‘06’, not ‘31’.)

In the second case the program uses the shared data storage primarily to conserve storage space: the shared storage area is always read using the same view as when last written to. In this case the storage areas can be treated as conceptually separate (unshared).

By default, all views of shared storage must have the same TS2K type in AnnoDomini. If, in the above example, YY is annotated with WW and both MM and DD with WW, then DATE-1 is inferred to have TS2K type WWWWW, since DATE-2 has type WWWWW, and DATE-1 and DATE-2 are aliased.

Requiring that all views of shared storage have the same TS2K type leads to propagation of type information from one view to the others also in cases where the different views are used conceptually separately. AnnoDomini allows annotating each both implicit and explicit redefinitions with

```
*TS2K ASSUME SEPARATE
```

which expresses that the redefinition should be treated as a separate, unshared definition. Thus type information from one view is not propagated to the other. AnnoDomini takes ASSUME SEPARATE annotations at face value. Since it is control-flow insensitive it does not check whether read-accesses to shared storage are always performed using the same view as the (dynamically) most recent write access.

6.4 Program Constants

Since moving literals into dates is a potential Year 2000 problem, literals both in the data division and procedure division sections are typed to be nonzero in AnnoDomini. Thus the MOVE statement in

```
*TS2K WWWWWW
  01 DATE-1 PIC XXXXXX.
MOVE 9990909 TO DATE-1
```

causes a type error as it is an apparent Year 2000 Problem (9990909 is often used as a special value, not a regular date). If, however, the literal is a correct date literal it can be given the desired type by an explicit ASSUME annotation, which eliminates the type error:

```
*TS2K ASSUME 9990909 IS WWWWWW
MOVE 9990909 TO DATE-1
```

6.5 Arrays

Arrays, called tables in COBOL, have program-static length in COBOL. AnnoDomini treats them as group items with repeated element types. For example an array of length 4 with element type WWWWWW has type WWWWWWWWWWWWW.

6.6 Call statements

COBOL programs can call other COBOL programs in one of two ways: by static calls or by dynamic calls. Static calls are linked at compile time, whereas dynamic calls are resolved at run-time. In the latter case it is statically evident which program is called. AnnoDomini issues warnings if arguments of calls have year types, but does not itself propagate the types across call boundaries.

6.7 Nonuniform language features

There are a number of language features in COBOL that make correct conversion complex: e.g., impossibility of calling programs in PERFORM statements; nondistributivity of size error handling in ADD/SUBTRACT CORRESPONDING statements; COPY statements with a REPLACING phrase; variable-sized arrays. To a large degree these can be attributed to a "nonuniform" language design. AnnoDomini is designed to preserve the semantics of the underlying program as much as possible. Where this is impractical or impossible AnnoDomini warns of and describes potential problems.

7 AnnoDomini System Components

AnnoDomini 1.0 for OS/VS COBOL runs on Windows NT 4.0 and Windows 9X. It is commercially available from Computer Generated Solutions, Inc.; see http://www.cgsinc.com or http://www.hafnium.com. AnnoDomini consists of 3 components:

1. the analysis and conversion engine (ACE),
2. the graphical user interface (GUI), and
3. IBM's Live Parsing Editor (LPEX).
ACE is the heart of AnnoDomini. It is written in Standard ML and consists of about 60,000 lines of code, of which about 25,000 are generated from lexing and parsing specifications using ML-Lex and ML-Yacc. LPEX is a syntax-sensitive program editor developed by IBM and delivered with its VisualAge series of software development tools. The GUI is written in Visual Basic and consists of about 10,000 lines of code. It interacts with the user and provides access to both ACE and LPEX. Communication with ACE is through files that pass parameters from the GUI to ACE and return results to the GUI for presentation to the user. Communication with LPEX is through REXX macros, which position the cursor in LPEX at a particular source code line. They are also used to insert TS2K annotations and to generate the code during actual conversion.

Every time the GUI invokes ACE, ACE processes the whole program under consideration from scratch. This batch architecture makes AnnoDomini fault-tolerant and allows for a simplified software architecture. It also means that the state of a conversion project is completely represented — and documented — by the annotated programs themselves. The cost is that a complete pass with lexing, parsing and type checking is required each time a type check is initiated. This appears to be acceptable in practice since AnnoDomini operates on one program at a time and Year 2000 problems discovered are eliminated in groups. An untuned version of AnnoDomini takes about 30 seconds to process a 10,000 line program on a 133MHz PC with 64 MBytes of RAM.

8 Conclusion

We have presented the type-theoretic foundations and associated practical aspects of AnnoDomini, an integrated find-and-fix tool for making COBOL programs Year 2000 compliant.

The underlying software reengineering method consists of identifying and isolating potentially problematic data and their associated operations according to their intended use (here as calendar years), encapsulating them as abstract types, and finally replacing their implementation by safe, improved code with the same interface. This method appears to be eminently applicable to other problems than Year 2000 remediation, such as reengineering financial systems for the introduction of the Euro or for the Dow Jones Index passing the 10,000 mark.

References


