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 Hafnium ApS

Email: {phei,henglein,mossin,hniss,rambo,tofte}@diku.dk

WWW: http://www.diku.dk, http://www.hafnium.com

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.[1] 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 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 A.D., not 2000 A.D..

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

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. 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. xxiii.

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, dispensible 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 8601). 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[2] 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

[2]AnnoDomini is a registered trademark of Hafnium 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 $\alpha^{(n)}$ (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 WWNNNN
 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 WWNNNN
 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 WWNNNN
 01  DATE-1 PIC XXXXXX.
*TS2K YYYYNNNN
 01  DATE-2 PIC XXXXXXXX.

     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 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 WWNNNNNN
     MOVE DATE-2 TO DATE-1
```

The ASSUME annotation tells the type checker that X should be treated as having type WWNNNNNN *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 WWNNNN BY D4TO2N4
     MOVE DATE-2 TO DATE-1.
```

The coercion D4TO2N4 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 fully 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 WWNNNN -> YYYYNNNN
 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`.[3] Apart from `YYYY` and `WW`, 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.

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.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{dec}\ \mathit{stmt} \\
\mathit{dec} & ::= & \epsilon \mid \mathit{dec}\ \ \mathit{dn}{:}\,\texttt{elem}\ \tau \mid \mathit{dec}\ \ \mathit{dn}{:}\,\texttt{group}(\mathit{dec}) \\
\mathit{stmt} & ::= & \epsilon \mid \mathit{stmt}\ \mathit{stmt} \\
 & \mid & \texttt{PERFORM}\ \mathit{stmt}\ \texttt{UNTIL}\ \mathit{condexp} \\
 & \mid & \texttt{MOVE}\ \mathit{sender}\ \texttt{TO}\ \mathit{receiver} \\
 & \mid & \texttt{IF}\ \mathit{condexp}\ \mathit{stmt}\ \texttt{ELSE}\ \mathit{stmt}\ \texttt{ENDIF} \\
\mathit{exp} & ::= & \mathit{lit} \mid \mathit{pqdn} \mid \texttt{assume}\ \mathit{exp}\ \texttt{is}\ \tau \\
 & \mid & \texttt{coerce}\ \mathit{exp}\ \texttt{by}\ c \mid \mathit{exp}\ \mathit{binop}\ \mathit{exp} \\
\mathit{sender} & ::= & \mathit{exp} \mid \texttt{padl}_{(n,m)}(\mathit{exp}) \mid \texttt{padr}_{(n,m)}(\mathit{exp}) \\
 & \mid & \texttt{takel}_{(n,m)}(\mathit{exp}) \mid \texttt{taker}_{(n,m)}(\mathit{exp}) \\
\mathit{receiver} & ::= & \mathit{pqdn} \mid \texttt{assume}\ \mathit{receiver}\ \texttt{is}\ \tau \\
\mathit{pqdn} & ::= & \mathit{dn} \mid \mathit{dn}\ \texttt{IN}\ \mathit{pqdn} \\
\mathit{binop} & ::= & +_{\mathrm{WN}} \mid +_{\mathrm{NW}} \mid +_{\mathrm{YN}} \mid +_{\mathrm{NY}} \mid +_{\mathrm{NN}} \\
\tau & ::= & \eta \mid \tau\,\tau \mid \epsilon \\
\eta & ::= & \texttt{WW} \mid \texttt{YYYY} \mid \texttt{N} \mid \alpha^{(n)} \\
\mathit{condexp} & ::= & \mathit{exp}\ \mathit{relop}\ \mathit{exp} \\
\mathit{relop} & ::= & <_{\mathrm{WW}} \mid <_{\mathrm{YY}} \mid <_{\mathrm{NN}}
\end{array}
$$

Figure 1: Grammar for COBOL$_{2000}$.

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 Deursen 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 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 COBOL$_{2000}$. While much simplifed, 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 Lit of *(decimal) numeric literals*, ranged over by *lit*; a set Dn of *data names*, ranged over *dn*;[4] for each natural number $n \geq 1$, a set $\mathrm{TyVar}^{(n)}$ of *type variables of size n*, ranged over by $\alpha^{(n)}$; and a set C of *coercion names*, ranged over by c. $\mathrm{TyVar} = \bigcup_{n=1}^{\infty} \mathrm{TyVar}^{(n)}$.

The grammar for COBOL$_{2000}$ 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 (ϵ), or a declaration followed by a declaration of an elementary item

[3]Year 2000 unsafe years in the range 1900-1999 are considered a special case of windowed years with pivot 00.

[4]The COBOL term "data name" corresponds to "variable" in other programming languages.

```
YR1:      elem WW
YR2:      elem WW
MYDATE:   elem WWNNNNN
LONGDATE: group(
            YEAR:  elem YYYY
            MONTH: elem NN
            DAY:   elem NN)
BUF:      elem N(80)

MOVE assume 94 is WW TO YR1
COMPUTE YR2 = YR1 +WN 10
PERFORM
  COMPUTE YR1 = YR1 +WN 1
UNTIL YR2 <WW YR1
MOVE assume 940203 is WWN(4) TO MYDATE
MOVE coerce MYDATE by D2TO4N4 TO LONGDATE
IF YEAR IN LONGDATE <YY assume 2000 is YYYY
    MOVE padr(8,72)(assume LONGDATE is N(8)) TO BUF
ELSE MOVE padl(1,1)(5) TO DAY IN LONGDATE
ENDIF
```

Figure 2: A COBOL$_{2000}$ program

or a group item.[5] In COBOL$_{2000}$, all elementary items are numeric.[6] A statement can be *empty* (ϵ) or the sequential composition of two statements. Next, `PERFORM` *stmt* `UNTIL` *condexp* corresponds roughly to `repeat` *stmt* `until` *condexp* 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 $\texttt{padl}_{(n,m)}(exp)$ can only be used when *exp* is an elementary item; it pads the value of *exp* with n leading zeros. The sender $\texttt{padr}_{(n,m)}(exp)$ can only be used when *exp* is a group item; it pads the value of *exp* with m trailing zeros.[7] The sender $\texttt{takel}_{(n,m)}(exp)$ can only be used when *exp* is a group item; it extracts the leftmost n digits of the value of *exp*. The sender $\texttt{taker}_{(n,m)}(exp)$ 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.[8]

A *perhaps qualified data name* (*pqdn*) is either an unqualified data name *dn* or a *qualified data name* (*dn* `IN` *pqdn*). In the latter case, *pqdn* 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: $\epsilon\,\tau = \tau\,\epsilon$ and $(\tau_1\,\tau_2)\,\tau_3 = \tau_1\,(\tau_2\,\tau_3)$. Thus every type can be written in the form $\eta_1 \ldots \eta_k\epsilon$, $k \geq 0$. We write $\tau(n)$ to mean $\tau \cdots \tau$, (n times).

The *size* of a type is defined inductively as follows:

$$\begin{array}{ll}
\text{size}(\eta) = & \text{size}(\tau) = \\
\quad \text{case } \eta \text{ of} & \quad \text{case } \tau \text{ of} \\
\quad\; \texttt{WW} \Rightarrow 2 & \quad\; \epsilon \Rightarrow 0 \\
\quad | \; \texttt{YYYY} \Rightarrow 4 & \quad | \; \eta\tau' \Rightarrow \\
\quad | \; \texttt{N} \Rightarrow 1 & \qquad \text{size}(\eta) + \text{size}(\tau') \\
\quad | \; \alpha^{(n)} \Rightarrow n &
\end{array}$$

3 Type System 2000

COBOL$_{2000}$ has two kinds of data items: *numeric elementary items* and *group items*:

$$k \in \text{Kind} = \{\mathsf{e}, \mathsf{g}\}$$

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

$$\begin{array}{rcl}
\sigma & ::= & \tau \mid \mathit{TE} \\
\mathit{TE} & ::= & \epsilon \mid \mathit{TE}, \mathit{dn} : \sigma
\end{array}$$

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:

$$\begin{array}{l}
\text{flat}(\mathit{TE}) = \text{case } \mathit{TE} \text{ of} \\
\quad \epsilon \Rightarrow \epsilon \\
\;| \; \mathit{TE}, \mathit{dn} : \tau \Rightarrow \text{flat}(\mathit{TE})\; \tau \\
\;| \; \mathit{TE}, \mathit{dn} : \mathit{TE}' \Rightarrow \text{flat}(\mathit{TE})\; \text{flat}(\mathit{TE}')
\end{array}$$

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 $\boxed{\mathit{TE} \vdash \mathit{pqdn} : \sigma}$

$$\frac{\mathit{TE} \vdash \mathit{dn} : \sigma \quad \mathit{dn} \neq \mathit{dn}'}{\mathit{TE}, \mathit{dn}' : \sigma' \vdash \mathit{dn} : \sigma} \tag{1}$$

$$\frac{}{\mathit{TE}, \mathit{dn} : \sigma \vdash \mathit{dn} : \sigma} \tag{2}$$

[5]The COBOL term "group item" corresponds to the term "record" in other programming languages.

[6]COBOL allows several other kinds of elementary items.

[7]In COBOL group items are regarded as alphanumeric data and padding is with spaces, but since COBOL$_{2000}$ has only numeric elementary items, we pad with zeros.

[8]In COBOL, the two statements are not generally equivalent, not even when both are syntactically correct.

$$\frac{TE \vdash pqdn' : TE' \quad TE' \vdash dn : \sigma}{TE \vdash dn \ \mathtt{IN}\ pqdn' : \sigma} \tag{3}$$

Expressions $\boxed{TE \vdash exp : \tau :: k}$

$$\frac{\tau = \mathtt{N}(\mathrm{size}(lit))}{TE \vdash lit : \tau :: \mathtt{e}} \tag{4}$$

$$\frac{TE \vdash pqdn : \tau}{TE \vdash pqdn : \tau :: \mathtt{e}} \tag{5}$$

$$\frac{TE_1 \vdash pqdn : TE_2}{TE_1 \vdash pqdn : \mathrm{flat}(TE_2) :: \mathtt{g}} \tag{6}$$

$$\frac{TE \vdash exp : \tau' :: k \quad \mathrm{size}(\tau) = \mathrm{size}(\tau')}{TE \vdash \mathtt{assume}\ exp\ \mathtt{is}\ \tau : \tau :: k} \tag{7}$$

$$\frac{TE \vdash exp : \tau :: k \quad \mathrm{TypeOf}(c) = \tau \to \tau'}{TE \vdash \mathtt{coerce}\ exp\ \mathtt{by}\ c : \tau' :: \mathtt{e}} \tag{8}$$

$$\frac{TE \vdash exp_1 : \tau_1 :: \mathtt{e} \quad TE \vdash exp_2 : \tau_2 :: \mathtt{e} \quad \tau_1, \tau_2 \vdash binop : \tau_3}{TE \vdash exp_1\ binop\ exp_2 : \tau_3 :: \mathtt{e}} \tag{9}$$

Sending item $\boxed{TE \vdash sender : \tau}$

$$\frac{TE \vdash exp : \tau :: k}{TE \vdash exp : \tau} \tag{10}$$

$$\frac{TE \vdash exp : \tau :: \mathtt{e} \quad \mathrm{size}(\tau') = n \quad \mathrm{size}(\tau) = m}{TE \vdash \mathtt{padl}_{(n,m)}(exp) : \tau'\tau} \tag{11}$$

$$\frac{TE \vdash exp : \tau :: \mathtt{g} \quad \mathrm{size}(\tau) = n \quad \mathrm{size}(\tau') = m}{TE \vdash \mathtt{padr}_{(n,m)}(exp) : \tau\tau'} \tag{12}$$

$$\frac{TE \vdash exp : \tau :: \mathtt{g} \quad \tau = \tau_l\tau_r \quad \mathrm{size}(\tau_l) = n \quad \mathrm{size}(\tau_r) = m}{TE \vdash \mathtt{takel}_{(n,m)}(exp) : \tau_l} \tag{13}$$

$$\frac{TE \vdash exp : \tau :: \mathtt{e} \quad \tau = \tau_l\tau_r \quad \mathrm{size}(\tau_l) = n \quad \mathrm{size}(\tau_r) = m}{TE \vdash \mathtt{taker}_{(n,m)}(exp) : \tau_r} \tag{14}$$

Receivers $\boxed{TE \vdash receiver : \tau}$

$$\frac{TE \vdash pqdn : \tau}{TE \vdash pqdn : \tau} \tag{15}$$

$$\frac{TE \vdash pqdn : TE}{TE \vdash pqdn : \mathrm{flat}(TE)} \tag{16}$$

$$\frac{TE \vdash receiver : \tau' \quad \mathrm{size}(\tau) = \mathrm{size}(\tau')}{TE \vdash \mathtt{assume}\ receiver\ \mathtt{is}\ \tau : \tau} \tag{17}$$

Statements $\boxed{TE \vdash stmt}$

$$\frac{TE \vdash sender : \tau \quad TE \vdash receiver : \tau}{TE \vdash \mathtt{MOVE}\ sender\ \mathtt{TO}\ receiver} \tag{18}$$

$$\frac{TE \vdash condexp \quad TE \vdash stmt_1 \quad TE \vdash stmt_2}{TE \vdash \mathtt{IF}\ condexp\ stmt_1\ \mathtt{ELSE}\ stmt_2\ \mathtt{ENDIF}} \tag{19}$$

$$\frac{TE \vdash stmt \quad TE \vdash condexp}{TE \vdash \mathtt{PERFORM}\ stmt\ \mathtt{UNTIL}\ condexp} \tag{20}$$

$$\frac{}{TE \vdash \epsilon} \tag{21}$$

$$\frac{TE \vdash stmt_1 \quad TE \vdash stmt_2}{TE \vdash stmt_1\ stmt_2} \tag{22}$$

Declarations $\boxed{\vdash dec : TE}$

$$\frac{}{\vdash \epsilon : \epsilon} \tag{23}$$

$$\frac{\vdash dec : TE}{\vdash dec \quad dn\!: \mathtt{elem}\ \tau \ : \ TE, dn : \tau} \tag{24}$$

$$\frac{\vdash dec_1 : TE_1 \quad \vdash dec_2 : TE_2}{\vdash dec_1 \ dn\!: \mathtt{group}(dec_2) \ : \ TE_1, dn : TE_2} \tag{25}$$

Programs $\boxed{\vdash prog}$

$$\frac{\vdash dec : TE \quad TE \vdash stmt}{\vdash dec\ stmt} \tag{26}$$

Binary Operators $\boxed{\tau_1, \tau_2 \vdash binop : \tau_3}$

$$\frac{}{\mathtt{WW}, \mathtt{N}(i) \vdash +_{\mathrm{WN}} : \mathtt{WW}} \tag{27}$$

$$\frac{}{\mathtt{YYYY}, \mathtt{N}(i) \vdash +_{\mathrm{YN}} : \mathtt{YYYY}} \tag{28}$$

$$\frac{}{\mathtt{N}(i), \mathtt{N}(j) \vdash +_{\mathrm{NN}} : \mathtt{N}(1 + \max(i,j))} \tag{29}$$

$$\frac{}{\mathtt{N}(i), \mathtt{WW} \vdash +_{\mathrm{NW}} : \mathtt{WW}} \tag{30}$$

$$\frac{}{\texttt{N}(i), \texttt{YYYY} \vdash +_{\text{NY}} : \texttt{YYYY}} \tag{31}$$

Conditional Expressions $\boxed{TE \vdash condexp}$

$$\frac{TE \vdash exp_1 : \tau_1 :: \texttt{e} \quad TE \vdash exp_2 : \tau_2 :: \texttt{e} \quad \tau_1, \tau_2 \vdash relop}{TE \vdash exp_1\ relop\ exp_2} \tag{32}$$

Relational Operators $\boxed{\tau_1, \tau_2 \vdash relop}$

$$\frac{}{\texttt{WW}, \texttt{WW} \vdash <_{\text{WW}}} \tag{33}$$

$$\frac{}{\texttt{YYYY}, \texttt{YYYY} \vdash <_{\text{YY}}} \tag{34}$$

$$\frac{}{\texttt{N}(i), \texttt{N}(j) \vdash <_{\text{NN}}} \tag{35}$$

Rules 1–3 describe how perhaps qualified data names are looked up in the type environment. In rule 3, the first premise expresses that $pqdn'$ 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 $TE \vdash exp : \tau :: k$, read: in TE, exp has type τ of kind k. The kind determines how the value of exp may be padded, as explained later.

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

Rule 5 gives kind e to elementary items, while rule 6 gives kind g to group items.

Rule 7 shows that applying `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 `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 $\tau_1, \tau_2 \vdash binop : \tau$ holds if τ_1 and τ_2 are legal types of the left and right arguments of $binop$, respectively, and τ is the type of the result of applying $binop$, see rules 27–31.

The rules for senders (10–14) allow one to infer statements of the form $TE \vdash sender : \tau$, read: in TE, $sender$ has type τ. 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 e, 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 e (rule 14).

The identifier which is assigned a value in the `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 `MYYEAR` and `MYDATE` have types `WW` and `WWNNNN`, respectively. Then `COMPUTE MYYEAR = MYYEAR + 1` is well-typed, but `COMPUTE MYDATE = MYDATE+10000` (a common COBOL idiom!) is not. Similarly,

```
LONGDATE : group(
    YEAR: YYYY
    MONTH: NN
    DAY  : NN)
SHORTDATE: WWNNNN
MOVE LONGDATE TO SHORTDATE
```

is not well-typed, although COBOL permits it: the types `YYYYNNNN` and `WWNNNN` 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, `D2TO4N4` is a coercion of type `WWNNNN` $\to$ `YYYYNNNN`.

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

An expression is *well-typed in TE*, if there exists a type τ and a kind k such that $TE \vdash exp : \tau :: k$. 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 COBOL$_{2000}$. 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 are 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

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

The *support of* S, written 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 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 $\mathrm{Inv}(S)$ is defined by

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

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 τ_1 and τ_2 are equal under the monoid laws, then $S(\tau_1)$ and $S(\tau_2)$ are also equal under the monoid laws. Substitution preverves size:

Lemma 3.1
For all η and τ, $size(S(\eta)) = size(\eta)$ and $size(S(\tau)) = size(\tau)$.

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 τ_1 and τ_2 is a substitution S satisfying $S(\tau_1) = S(\tau_2)$. A substitution S^* is a *most general unifier for τ_1 and τ_2* if $S^*(\tau_1) = S^*(\tau_2)$ and for every unifier S for τ_1 and τ_2 there exists an S' with $S = S' \circ S^*$.

Interestingly, not all pairs (τ_1, τ_2) have a most general unifier. For example, let $\tau_1 = \alpha_1^{(4)}\alpha_2^{(2)}$ and $\tau_2 = \alpha_3^{(2)}\alpha_4^{(4)}$. 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

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

We have $S^*(\tau_1) = \alpha_3^{(2)}\alpha_5^{(2)}\alpha_2^{(2)} = S^*(\tau_2)$, so S^* unifies τ_1 and τ_2. However, S^* is not a most general unifier. To see this, consider

$$S = \{\alpha_1^{(4)} \mapsto \alpha_3^{(2)}\mathtt{NN},\ \alpha_4^{(4)} \mapsto \mathtt{NN}\alpha_2^{(2)},\ \alpha_5^{(2)} \mapsto \mathtt{WW}\}$$

Note that S is a unifier for τ_1 and τ_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 TyVar. (U stands for "used".) A substitution S^* is a *most general U-unifier for τ_1 and τ_2*, if $\mathrm{tyvars}(\tau_1) \cup \mathrm{tyvars}(\tau_2) \subseteq U$; $S^*(\tau_1) = S^*(\tau_2)$; 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 either raises an exception, FAIL, or returns a substitution. It uses two auxiliary functions:

```
fun unify_η($\eta_1, \eta_2$) = (* size($\eta_1$) = size($\eta_2$) *)
  case ($\eta_1, \eta_2$) of
    ($\alpha_1^{(n)}$, _) ⇒ {$\alpha_1^{(n)} \mapsto \eta_2$}
  | (_, $\alpha_2^{(n)}$) ⇒ {$\alpha_2^{(n)} \mapsto \eta_1$}
  | (_, _) ⇒ if $\eta_1 = \eta_2$ then $I$ else raise FAIL.

fun splitl($\eta, i, U$) = (* $0 < i <$ size($\eta$) *)
  case $\eta$ of
    $\alpha^{(n)}$ ⇒ let $\alpha_1^{(i)}, \alpha_2^{(n-i)}$ be distinct type
                              variables not in $U$
            in ({$\alpha^{(n)} \mapsto \alpha_1^{(i)}\alpha_2^{(n-i)}$}, {$\alpha_1^{(i)}, \alpha_2^{(n-i)}$})
            end
  | _ ⇒ raise FAIL
```

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

```
fun unify($\tau_1, \tau_2, U$) = case ($\tau_1, \tau_2$) of
  ($\epsilon, \epsilon$) ⇒ ($I, \emptyset$)
| ($\eta_1\tau_{11}, \eta_2\tau_{22}$) ⇒
    if size($\eta_1$) = size($\eta_2$) then
      let $S_1$ = unify_η($\eta_1, \eta_2$)
          ($S_2, U_2$) = unify($S_1(\tau_{11}), S_1(\tau_{22}), U$)
      in ($S_2 \circ S_1, U_2$) end
    else if size($\eta_1$) > size($\eta_2$) then
      let ($S_1, U_1$) = splitl($\eta_1$, size($\eta_2$), $U$)
          ($S_2, U_2$) = unify($S_1(\tau_1), S_1(\tau_2), U \cup U_1$)
      in ($S_2 \circ S_1, U_1 \cup U_2$) end
    else unify($\tau_2, \tau_1$)
| _ ⇒ raise FAIL
```

Lemma 3.2 *Assume $size(\eta_1) = size(\eta_2)$. If there exists a unifier for η_1 and η_2 then $S = unify_\eta(\eta_1, \eta_2)$ succeeds and S is a most general unifier for η_1 and η_2. If there exists no unifier for η_1 and η_2, then $unify_\eta(\tau_1, \tau_2)$ raises FAIL.*

Theorem 3.1 *Assume $tyvars(\tau_1) \cup tyvars(\tau_2) \subseteq U$. If there exists a unifier for τ_1 and τ_2 then*

$$(S^*, U') = unify(\tau_1, \tau_2, U) \qquad (36)$$

succeeds and $S^*(\tau_1) = S^*(\tau_2)$, for any unifier S_0 of τ_1 and τ_2 there is an S' with $S_0 \downarrow U = (S' \circ S^*) \downarrow U$, $Inv(S^*) \subseteq tyvars(\tau_1) \cup tyvars(\tau_2) \cup U'$, and $U \cap U' = \emptyset$.

If there is no unifier for τ_1 and τ_2, then (36) raises 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
$A \vdash phrase : B$ implies $S(A) \vdash S(phrase) : S(B)$.

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 TE be a type environment and let y be a perhaps qualified data name. Note that there exists at most one σ such that $TE \vdash y : \sigma$. Define lookup(TE, y) to be this σ, if it exists, and let lookup(TE, y) terminate with FAIL otherwise.

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

```
BinOp($\tau_1, \tau_2, binop, U$) =
  case $binop$ of
    $^{+}$WN ⇒ let ($S_1, U_1$) = unify($\tau_1$, WW, $U$)
                  ($S_2, U_2$) = unify($S_1(\tau_2)$, N(size($\tau_2$)), $U \cup U_1$)
              in ($S_2 \circ S_1$, WW, $U_1 \cup U_2$) end
  | $^{+}$YN ⇒ let ($S_1, U_1$) = unify($\tau_1$, YYYY, $U$)
                  ($S_2, U_2$) = unify($S_1(\tau_2)$, N(size($\tau_2$)), $U \cup U_1$)
              in ($S_2 \circ S_1$, YYYY, $U_1 \cup U_2$) end
  | $^{+}$NN ⇒ let ($S_1, U_1$) = unify($\tau_1$, N(size($\tau_1$)), $U$)
                  ($S_2, U_2$) = unify($S_1(\tau_2)$, N(size($\tau_2$)), $U \cup U_1$)
              in ($S_2 \circ S_1$, N(1 + max(size($\tau_1$), size($\tau_2$)), $U_1 \cup U_2$)
              end
  | $^{+}$NW ⇒ let ($S_1, U_1$) = unify($\tau_1$, N(size($\tau_1$)), $U$)
                  ($S_2, U_2$) = unify($S_1(\tau_2)$, WW, $U \cup U_1$)
              in ($S_2 \circ S_1$, WW, $U_1 \cup U_2$) end
```

```
| $^{+}\mathrm{NY} \Rightarrow$ let $(S_1, U_1) = \mathrm{unify}(\tau_1, \mathtt{N}(\mathrm{size}(\tau_1)), U)$
              $(S_2, U_2) = \mathrm{unify}(S_1(\tau_2), \mathtt{YYYY}, U \cup U_1)$
          in $(S_2 \circ S_1, \mathtt{YYYY}, U_1 \cup U_2)$ end
```

This algorithm satisfies the following soundness and completeness properties.

Lemma 3.4 *Assume that $tyvars(\tau_1, \tau_2) \subseteq U$ and $(S, \tau, U') = BinOp(\tau_1, \tau_2, binop, U)$ succeeds. Then $S(\tau_1), S(\tau_2) \vdash binop : \tau$, $Inv(S) \subseteq tyvars(\tau_1, \tau_2) \cup U'$ and $U \cap U' = \emptyset$.*

Lemma 3.5
Assume $tyvars(\tau_1, \tau_2) \subseteq U$. If there exist S_0 and and τ_0 such that

$$S_0\tau_1, S_0\tau_2 \vdash binop : \tau_0 \tag{37}$$

then

$$(S, \tau, U') = BinOp(\tau_1, \tau_2, binop, U) \tag{38}$$

succeeds and for every S_0 and τ_0 satisfying (37) there exists an S' such that $S_0 \downarrow U = (S' \circ S) \downarrow U$ and $\tau_0 = \tau$. Moreover, if there is no S_0 and τ_0 satisfying (37), then (38) terminates with FAIL.

The proof is by cases on *binop* using Theorem 3.1.
The following functions check rules 4–9. (In the last case, we assume that the function raises `FAIL`, if the pattern matching on `e` fails.)

```
Exp($TE$, $exp$, $U$) = case $exp$ of
  $lit \Rightarrow (I, \mathtt{N}(\mathrm{size}(lit)), \mathtt{e}, \emptyset)$
 | $pqdn \Rightarrow$ (case lookup($TE$, $pqdn$) of
                     $\tau \Rightarrow (I, \tau, \mathtt{e}, \emptyset)$
                   | $TE' \Rightarrow (I, \mathrm{flat}(TE'), \mathtt{g}, \emptyset)$)
 | assume $exp_1$ is $\tau \Rightarrow$
  let $(S_1, \tau_1, k_1, U_1) = \mathrm{Exp}(TE, exp_1, U)$
  in if $\mathrm{size}(\tau_1) = \mathrm{size}(\tau)$
     then $(S_1, S_1(\tau), k_1, U_1)$
     else raise FAIL
  end
 | coerce $exp_1$ by $c \Rightarrow$
  let $(S_1, \tau_1, k_1, U_1) = \mathrm{Exp}(TE, exp_1, U)$
  in let $\mathrm{TypeOf}(c) = \tau_2 \to \tau_2'$ in
        let $(S_2, U_2) = \mathrm{unify}(\tau_1, \tau_2, U \cup U_1)$
        in $(S_2 \circ S_1, S_2(\tau_2'), \mathtt{e}, U_1 \cup U_2)$
        end
     end
  end
 | $exp_1$ $binop$ $exp_2 \Rightarrow$
  let $(S_1, \tau_1, \mathtt{e}, U_1) = \mathrm{Exp}(TE, exp_1, U)$
      $(S_2, \tau_2, \mathtt{e}, U_2) = \mathrm{Exp}(S_1(TE), S_1(exp_2), U \cup U_1)$
      $(S_3, \tau_3, U_3) = \mathrm{BinOp}(S_2(\tau_1), \tau_2, binop, U \cup U_1 \cup U_2)$
  in $(S_3 \circ S_2 \circ S_1, \tau_3, \mathtt{e}, U_1 \cup U_2 \cup U_3)$
  end
```

The algorithm Exp satisfies the following soundness and completeness properties:

Theorem 3.2 *Assume $tyvars(TE) \cup tyvars(exp) \subseteq U$. If $(S, \tau, k, U') = Exp(TE, exp, U)$ succeeds then $S(TE) \vdash S(exp) : \tau :: k$, $Inv(S) \cup tyvars(\tau) \subseteq tyvars(TE) \cup tyvars(exp) \cup U'$, and $U \cap U' = \emptyset$.*

Theorem 3.3 *Assume $tyvars(TE) \cup tyvars(exp) \subseteq U$. If there exist S_0, τ_0 and k_0 satisfying*

$$S_0(TE) \vdash S_0(exp) : \tau_0 :: k_0 \tag{39}$$

then

$$(S', \tau', k', U') = Exp(TE, exp, U) \tag{40}$$

succeeds and for all S_0, τ_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, τ_0 and k_0 satisfying (39), then (40) terminates with 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 ($\underline{a}$) and numbers (a). A *value*, v, is a numeral, i.e., a sequence of digits. The *size* of a value $\underline{a}$, written size($\underline{a}$), is the number of digits in $\underline{a}$. $\mathtt{0}(n)$ denotes the sequence of n zeros. For every value v and $n \geq 0$ satisfying $\mathrm{size}(v) \leq n$, we define $\mathrm{fill}(n, v)$ to be $\mathtt{0}(n - \mathrm{size}(v))v$.

Evaluation can fail. The semantics distinguishes between the following failures: `Y2K` (Year 2000 failure), `Arith` (arithmetic failure), and `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 $\mathrm{extract}(s, l, u)$ to be the value $s(l) \cdots s(u-1)$, if $0 \leq l \leq u \leq \mathrm{size}(s)$, and the failure `Wrong` otherwise. Further, we define $\mathrm{update}(s, l, u, v)$ to be the store s' defined by

$$s'(i) = \begin{cases} s(i) & \text{if } 0 \leq i < l \text{ or } u \leq i < \mathrm{size}(s) \\ d_{i-l} & \text{if } l \leq i < u \end{cases}$$

provided $0 \leq l \leq u \leq \mathrm{size}(s)$ and $v = d_0 \cdots d_{u-l-1}$; otherwise, we define $\mathrm{update}(s, l, u, v)$ to be the failure `Wrong`.

We define *(dynamic) environments* (E) and *denotable values* (dval) as follows:

$$\begin{array}{rcl} E & ::= & \epsilon \mid E, dn = \mathrm{dval} \\ \mathrm{dval} & ::= & (E, l, u) \end{array}$$

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, iff dn is an elementary item.

The evaluation relation takes the form $A \vdash phrase \Rightarrow B$, 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 `Wrong`. An *expression result* (r) is either a value, `Y2K`, `Wrong`, or `Arith`. A *statement result* (t) is either a store, `Y2K`, `Wrong`, or `Arith`. In the evaluation of relational expressions, we use booleans (`true`, `false`); a *condition result* (b) is either `true`, `false`, `Wrong`, `Arith` or `Y2K`.

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

Finally, 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:

$$\frac{E \vdash \mathit{pqdn} \Rightarrow f}{s, E \vdash \mathit{pqdn} \Rightarrow f} \tag{41}$$

$$\frac{E \vdash \mathit{pqdn} \Rightarrow (\_, l, u) \quad \texttt{Wrong} = \text{extract}(s, l, u)}{s, E \vdash \mathit{pqdn} \Rightarrow \texttt{Wrong}} \tag{42}$$

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

Perhaps Qualified Data Names $\boxed{E \vdash \mathit{pqdn} \Rightarrow e}$

$$\frac{}{\epsilon \vdash \mathit{pqdn} \Rightarrow \texttt{Wrong}} \tag{43}$$

$$\frac{E \vdash \mathit{dn} \Rightarrow \text{dval} \quad \mathit{dn} \neq \mathit{dn}'}{E, \mathit{dn}' = \text{dval}' \vdash \mathit{dn} \Rightarrow \text{dval}} \tag{44}$$

$$\frac{}{E, \mathit{dn} = \text{dval} \vdash \mathit{dn} \Rightarrow \text{dval}} \tag{45}$$

$$\frac{E \vdash \mathit{pqdn}' \Rightarrow (E', \_, \_) \qquad E' \vdash \mathit{dn} \Rightarrow \text{dval}}{E \vdash \mathit{dn}\ \texttt{IN}\ \mathit{pqdn}' \Rightarrow \text{dval}} \tag{46}$$

Expressions $\boxed{s, E \vdash \mathit{exp} \Rightarrow r}$

$$\frac{}{s, E \vdash \mathit{lit} \Rightarrow \mathit{lit}} \tag{47}$$

$$\frac{E \vdash \mathit{pqdn} \Rightarrow (\_, l, u) \quad v = \text{extract}(s, l, u)}{s, E \vdash \mathit{pqdn} \Rightarrow v} \tag{48}$$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow v}{s, E \vdash \texttt{assume}\ \mathit{exp}\ \texttt{is}\ \tau : v} \tag{49}$$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow v \quad \text{Apply}(c, v) = v'}{s, E \vdash \texttt{coerce}\ \mathit{exp}\ \texttt{by}\ c : v'} \tag{50}$$

$$\frac{s, E \vdash \mathit{exp}_1 \Rightarrow v_1 \quad s, E \vdash \mathit{exp}_2 \Rightarrow v_2 \quad v_1, v_2 \vdash \mathit{binop} \Rightarrow v}{s, E \vdash \mathit{exp}_1\ \mathit{binop}\ \mathit{exp}_2 \Rightarrow v} \tag{51}$$

Sending item $\boxed{s, E \vdash \mathit{sender} \Rightarrow r}$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow v}{s, E \vdash \mathit{exp} \Rightarrow v} \tag{52}$$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow v'}{s, E \vdash \texttt{padl}_{(n,m)}(\mathit{exp}) \Rightarrow \texttt{0}(n)v'} \tag{53}$$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow v'}{s, E \vdash \texttt{padr}_{(n,m)}(\mathit{exp}) \Rightarrow v'\texttt{0}(m)} \tag{54}$$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow d_1 \ldots d_n d'_1 \ldots d'_n}{s, E \vdash \texttt{takel}_{(n,m)}(\mathit{exp}) : d_1 \ldots d_n} \tag{55}$$

$$\frac{s, E \vdash \mathit{exp} \Rightarrow d_1 \ldots d_n d'_1 \ldots d'_m}{s \vdash \texttt{taker}_{(n,m)}(\mathit{exp}) : d'_1 \ldots d'_m} \tag{56}$$

Receivers $\boxed{E \vdash \mathit{receiver} \Rightarrow (l, u)/\texttt{Wrong}}$

$$\frac{E \vdash \mathit{pqdn} \Rightarrow (E', l, u)}{E \vdash \mathit{pqdn} \Rightarrow (l, u)} \tag{57}$$

$$\frac{E \vdash \mathit{receiver} \Rightarrow (l, u)}{E \vdash \texttt{assume}\ \mathit{receiver}\ \texttt{is}\ \tau \Rightarrow (l, u)} \tag{58}$$

Statements $\boxed{s, E \vdash \mathit{stmt} \Rightarrow t}$

$$\frac{s, E \vdash \mathit{sender} \Rightarrow v \quad s, E \vdash \mathit{receiver} \Rightarrow (l, u)}{s, E \vdash \texttt{MOVE}\ \mathit{sender}\ \texttt{TO}\ \mathit{receiver} \Rightarrow \text{update}(s, l, u, v)} \tag{59}$$

$$\frac{s, E \vdash \mathit{condexp} \Rightarrow \texttt{true} \quad s, E \vdash \mathit{stmt}_1 \Rightarrow s'}{s, E \vdash \texttt{IF}\ \mathit{condexp}\ \mathit{stmt}_1\ \texttt{ELSE}\ \mathit{stmt}_2\ \texttt{ENDIF} \Rightarrow s'} \tag{60}$$

$$\frac{s, E \vdash \mathit{condexp} \Rightarrow \texttt{false} \quad s, E \vdash \mathit{stmt}_2 \Rightarrow s'}{s, E \vdash \texttt{IF}\ \mathit{condexp}\ \mathit{stmt}_1\ \texttt{ELSE}\ \mathit{stmt}_2\ \texttt{ENDIF} \Rightarrow s'} \tag{61}$$

$$\frac{\begin{array}{c} s, E \vdash \mathit{condexp} \Rightarrow \texttt{true} \quad s, E \vdash \mathit{stmt} \Rightarrow s' \\ s', E \vdash \texttt{PERFORM}\ \mathit{stmt}\ \texttt{UNTIL}\ \mathit{condexp} \Rightarrow s'' \end{array}}{s, E \vdash \texttt{PERFORM}\ \mathit{stmt}\ \texttt{UNTIL}\ \mathit{condexp} \Rightarrow s''} \tag{62}$$

$$\frac{s, E \vdash \mathit{condexp} \Rightarrow \texttt{false}}{s, E \vdash \texttt{PERFORM}\ \mathit{stmt}\ \texttt{UNTIL}\ \mathit{condexp} \Rightarrow s} \tag{63}$$

$$\frac{}{s, E \vdash \epsilon \Rightarrow s} \tag{64}$$

$$\frac{s, E \vdash \mathit{stmt}_1 \Rightarrow s_1 \quad s_1, E \vdash \mathit{stmt}_2 \Rightarrow s_2}{s, E \vdash \mathit{stmt}_1\ \mathit{stmt}_2, \Rightarrow s_2} \tag{65}$$

Declarations $\boxed{l \vdash \mathit{dec} \Rightarrow E, z}$

$$\frac{}{l \vdash \epsilon \Rightarrow \epsilon, 0} \tag{66}$$

$$\frac{l \vdash dec_1 \Rightarrow E_1, z_1}{\begin{array}{l} l \vdash dec_1\ dn\texttt{: elem}\ \tau \Rightarrow\ E_1, dn = (\epsilon, l + z_1, l + z_1 + \text{size}(\tau)), \\ \qquad\qquad z_1 + \text{size}(\tau)\end{array}} \tag{67}$$

$$\frac{l \vdash dec_1 \Rightarrow E_1, z_1 \quad l + z_1 \vdash dec_2 \Rightarrow E_2, z_2}{\begin{array}{l} l \vdash dec_1\ dn\texttt{: group}(dec_2) \Rightarrow\ E_1, dn = (E_2, l + z_1, l + z_1 + z_2), \\ \qquad\qquad z_1 + z_2\end{array}} \tag{68}$$

Programs $\boxed{\vdash prog \Rightarrow s}$

$$\frac{0 \vdash dec \Rightarrow E, z \quad \text{initstore}(z), E \vdash stmt \Rightarrow s}{\vdash dec\ stmt \Rightarrow s} \tag{69}$$

Binary Operators $\boxed{v, v \vdash binop \Rightarrow r}$

$$\frac{a + b \leq 99}{\underline{a}, \underline{b} \vdash +_{\text{WN}} \Rightarrow \text{fill}(2, \underline{a+b})} \qquad \frac{a + b > 99}{\underline{a}, \underline{b} \vdash +_{\text{WN}} \Rightarrow \texttt{Y2K}} \tag{70}$$

$$\frac{a + b \leq 9999}{\underline{a}, \underline{b} \vdash +_{\text{YN}} \Rightarrow \text{fill}(4, \underline{a+b})} \qquad \frac{a + b > 9999}{\underline{a}, \underline{b} \vdash +_{\text{YN}} \Rightarrow \texttt{Arith}} \tag{71}$$

$$\frac{i = 1 + \max(\text{size}(\underline{a}, \underline{b}))}{\underline{a}, \underline{b} \vdash +_{\text{NN}} \Rightarrow \text{fill}(i, \underline{a+b})} \tag{72}$$

$$\frac{a + b \leq 99}{\underline{a}, \underline{b} \vdash +_{\text{NW}} \Rightarrow \text{fill}(2, \underline{a+b})} \qquad \frac{a + b > 99}{\underline{a}, \underline{b} \vdash +_{\text{NW}} \Rightarrow \texttt{Y2K}} \tag{73}$$

$$\frac{a + b \leq 9999}{\underline{a}, \underline{b} \vdash +_{\text{NY}} \Rightarrow \text{fill}(4, \underline{a+b})} \qquad \frac{a + b > 9999}{\underline{a}, \underline{b} \vdash +_{\text{NY}} \Rightarrow \texttt{Arith}} \tag{74}$$

Conditional Expressions $\boxed{s, E \vdash condexp \Rightarrow b}$

$$\frac{s, E \vdash exp_1 \Rightarrow v_1 \quad s, E \vdash exp_2 \Rightarrow v_2 \quad v_1, v_2 \vdash relop \Rightarrow b}{s, E \vdash exp_1\ relop\ exp_2 \Rightarrow b} \tag{75}$$

Relational Operators $\boxed{v, v \vdash relop \Rightarrow b}$

$$\frac{a \leq 99\ b \leq 99\ a < b}{\underline{a}, \underline{b} \vdash <_{\text{WW}} \Rightarrow \texttt{true}} \qquad \frac{a \leq 99\ b \leq 99\ a \geq b}{\underline{a}, \underline{b} \vdash <_{\text{WW}} \Rightarrow \texttt{false}} \tag{76}$$

$$\frac{a \leq 9999\ b \leq 9999\ a < b}{\underline{a}, \underline{b} \vdash <_{\text{YY}} \Rightarrow \texttt{true}} \qquad \frac{a \leq 9999\ b \leq 9999\ a \geq b}{\underline{a}, \underline{b} \vdash <_{\text{YY}} \Rightarrow \texttt{false}} \tag{77}$$

$$\frac{a > 99 \vee b > 99}{\underline{a}, \underline{b} \vdash <_{\text{WW}} \Rightarrow \texttt{Y2K}} \qquad \frac{a > 9999 \vee b > 9999}{\underline{a}, \underline{b} \vdash <_{\text{YY}} \Rightarrow \texttt{Wrong}} \tag{78}$$

$$\frac{a < b}{\underline{a}, \underline{b} \vdash <_{\text{NN}} \Rightarrow \texttt{true}} \qquad \frac{a \geq b}{\underline{a}, \underline{b} \vdash <_{\text{NN}} \Rightarrow \texttt{false}} \tag{79}$$

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 $\{i \mid l_1 \leq i < u_n\}$. Given l and u, we define the restriction of ST to (l, u) to be the subsequence

$$(l_i, u_i, \tau_i) \ldots (l_{i+j}, u_{i+j}, \tau_{i+j})$$

which satisfies $l_i = l$ and $u_{i+j} = u$, if one exists; further we define the *store type* of (ST, l, u), written $\text{sttyp}(ST, l, u)$, to be $\tau_i \cdots \tau_{i+j}$.

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 $\boxed{ST \models E : TE}$

$$\frac{}{ST \models \epsilon : \epsilon} \qquad \frac{ST \models E : TE \quad \text{sttyp}(ST, l, u) = \tau}{ST \models E, dn = (\epsilon, l, u) : TE, dn : \tau}$$

$$\frac{ST \models E_1 : TE_1 \quad ST \models E_2 : TE_2 \quad \text{sttyp}(ST, l, u) = \text{flat}(TE_2)}{ST \models E_1, dn = (E_2, l, u) : TE_1, dn : TE_2}$$

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 `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 $TE \vdash exp : \tau :: k$ and $ST \models E : TE$ and $Dom(s) = addr(ST)$ and $s, E \vdash exp \Rightarrow r$ then r is not `Wrong` (although it may be `Y2K`) and if r is a value v then $size(v) = size(\tau)$.*

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 `WW` 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. $\texttt{WW} \leq \texttt{YYYY}$,

2. $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$ implies $\tau_1\tau_2 \leq \tau_1'\tau_2'$;

3. $TE \leq TE'$ and $\tau \leq \tau'$ implies $TE, dn : \tau \leq TE', dn : \tau'$;

4. $TE_1 \leq TE_1'$ and $TE_2 \leq TE_2'$ implies $TE_1, dn : TE_2 \leq TE_1', dn : TE_2'$;

A *store expander* is a finite map from store indices to store indices. (A *store index* is a non-negative integer.) Let ST and ST' be store typings and let φ be a store expander. We say that ST' *expands* ST *via* φ, if $\varphi \vdash ST \leq ST'$, where $\_ \vdash \_ \leq \_$ is defined by

$$\frac{}{\varphi \vdash \epsilon \leq \epsilon} \quad (80)$$

$$\frac{\varphi \vdash ST \leq ST' \quad \tau \leq \tau'}{\varphi \vdash ST(l,u,\tau) \leq ST'(\varphi(l),\varphi(u),\tau')} \quad (81)$$

For example, one has $\varphi \vdash (0,5,\mathtt{N}(5))(5,7,\mathtt{WW})(7,9,\mathtt{WW}) \leq (0,5,\mathtt{N}(5))(5,9,\mathtt{YYYY})(9,11,\mathtt{WW})$, where $\varphi = \{0 \mapsto 0, 5 \mapsto 5, 7 \mapsto 9, 9 \mapsto 11\}$. Note that expansion allows some of the two-digit years to be expanded to four digits while leaving other years in two-digit form.

The *application* of a store expander φ to an environment E, written $\varphi \bullet E$, is defined inductively: $\varphi \bullet \epsilon = \epsilon$; and $\varphi \bullet (E, dn = (E', l, u)) = (\varphi \bullet E), dn = (\varphi \bullet E', \varphi(l), \varphi(u))$.

Given two types τ and τ' with $\tau \leq \tau'$ and a value v with the same size as τ we define $\mathrm{expand}(\tau,\tau',v)$ to be the following value:

```
fun $\mathrm{expand}(\tau,\tau',v) = $ case $(\tau,\tau',v)$ of
  $(\epsilon,\epsilon,\epsilon) \Rightarrow \epsilon$
  | $(\mathtt{WW}\tau_1, \mathtt{YYYY}\tau_1', d_1d_2v') \Rightarrow 19d_1d_2\,\mathrm{expand}(\tau_1,\tau_1',v')$
  | $(\eta\tau_1', \eta\tau_2', d_1\ldots d_nv') \Rightarrow d_1\ldots d_n\,\mathrm{expand}(\tau_1',\tau_2',v')$
      where $n = \mathrm{size}(\eta)$
```

Note that $\tau \leq \tau'$ and $\mathrm{size}(v) = \mathrm{size}(\tau)$ implies that $v' = \mathrm{expand}(\tau,\tau',v)$ succeeds and $\mathrm{size}(v') = \mathrm{size}(\tau')$.

Assume $\varphi \vdash ST \leq ST'$ and $\mathrm{Dom}(s) = \mathrm{addr}(ST)$; we say that store s' *expands* s, written $\varphi \vdash s : ST \leq s' : ST'$, if $\mathrm{Dom}(s') = \mathrm{addr}(ST')$ and for every (l,u,τ) in ST and corresponding $(\varphi(l),\varphi(u),\tau')$ in ST' we have $\mathrm{extract}(s',\varphi(l),\varphi(u)) = \mathrm{expand}(\tau,\tau',\mathrm{extract}(s,l,u))$.

We say that a coercion c is *expansive* if, letting $\tau \to \tau' = \mathrm{TypeOf}(c)$, we have $\tau \leq \tau'$ and, for all values v of size $\mathrm{size}(\tau)$, $\mathrm{Apply}(c,v) = \mathrm{expand}(\tau,\tau',v)$. Henceforth, we require that all coercions be expansive.

We now define how programs may be transformed in order to fix Year 2000 problems using expansion. The transformation is not a function but a binary relation ($\leadsto$) on judgements of the form $A \vdash phrase : B$. For example,

$$TE \vdash exp : \tau :: k \leadsto TE' \vdash exp' : \tau' :: k'$$

is read: the judgement $TE \vdash exp : \tau :: k$ is *transformed into* $TE' \vdash exp' : \tau' :: k'$.

We first state the expansion theorem and then define and explain the transformation. For brevity, we show only the theorem for expressions.

Theorem 5.1 *If $TE \vdash exp : \tau :: k \leadsto TE' \vdash exp' : \tau' :: k'$ and $\varphi \models s : ST \leq s' : ST'$ and $ST \models E : TE$ and $ST' \models \varphi \bullet E : TE'$ and $s, E \vdash exp \Rightarrow v$ then $s', \varphi \bullet E \vdash exp' \Rightarrow expand(\tau,\tau',v)$.*

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 $\boxed{TE \vdash pqdn : \sigma \leadsto TE' \vdash pqdn : \sigma'}$

$$\frac{dn \neq dn' \quad TE \vdash dn : \sigma \leadsto TE' \vdash dn : \sigma' \quad \sigma_1 \leq \sigma_1'}{TE, dn' : \sigma_1 \vdash dn : \sigma \leadsto TE', dn' : \sigma_1' \vdash dn : \sigma'} \quad (82)$$

$$\frac{TE \leq TE' \quad \sigma \leq \sigma'}{TE, dn : \sigma \vdash dn : \sigma \leadsto TE', dn : \sigma' \vdash dn : \sigma'} \quad (83)$$

$$\frac{\begin{array}{c}TE \vdash pqdn' : TE_1 \leadsto TE' \vdash pqdn' : TE_1' \\ TE_1 \vdash dn : \sigma \leadsto TE_1' \vdash dn : \sigma'\end{array}}{TE \vdash dn \;\mathtt{IN}\; pqdn : \sigma \leadsto TE' \vdash dn \;\mathtt{IN}\; pqdn : \sigma'} \quad (84)$$

Expressions $\boxed{TE \vdash exp : \tau :: k \leadsto TE' \vdash exp' : \tau' :: k'}$

$$\frac{\tau = \mathtt{N}(\mathrm{size}(lit)) \quad TE \leq TE'}{TE \vdash lit : \tau :: \mathsf{e} \leadsto TE' \vdash lit : \tau : \mathsf{e}} \quad (85)$$

$$\frac{TE \vdash pqdn : \tau \leadsto TE' \vdash pqdn : \tau'}{TE \vdash pqdn : \tau :: \mathsf{e} \leadsto TE' \vdash pqdn : \tau' : \mathsf{e}} \quad (86)$$

$$\frac{TE_1 \vdash pqdn : TE_2 \leadsto TE_1' \vdash pqdn : TE_2'}{TE_1 \vdash pqdn : \mathrm{flat}(TE_2) :: \mathsf{g} \leadsto TE_1' \vdash pqdn : \mathrm{flat}(TE_2') :: \mathsf{g}} \quad (87)$$

$$\frac{\mathrm{size}(\tau_1) = \mathrm{size}(\tau) \quad TE \vdash exp_1 : \tau_1 :: k \leadsto TE' \vdash exp_1' : \tau_1 :: k'}{\begin{array}{c}TE \vdash \mathtt{assume}\; exp_1 \;\mathtt{is}\; \tau : \tau :: k \leadsto \\ TE' \vdash \mathtt{assume}\; exp_1' \;\mathtt{is}\; \tau : \tau :: k'\end{array}} \quad (88)$$

$$\frac{\begin{array}{l}\mathrm{TypeOf}(c) = \tau \to \tau' \\ TE \vdash exp : \tau :: k \leadsto TE' \vdash exp' : \tau :: k\end{array}}{\begin{array}{l}TE \vdash \mathtt{coerce}\; exp \;\mathtt{by}\; c : \tau' :: \mathsf{e} \leadsto \\ TE' \vdash \mathtt{coerce}\; exp' \;\mathtt{by}\; c : \tau' :: \mathsf{e}\end{array}} \quad (89)$$

$$\frac{\begin{array}{l}\mathrm{TypeOf}(c) = \tau \to \tau' \\ TE \vdash exp : \tau : k \leadsto TE' \vdash exp' : \tau' :: \mathsf{e}\end{array}}{\begin{array}{l}TE \vdash \mathtt{coerce}\; exp \;\mathtt{by}\; c : \tau' :: \mathsf{e} \leadsto \\ TE' \vdash exp' : \tau' :: \mathsf{e}\end{array}} \quad (90)$$

$$\frac{\begin{array}{l}\mathrm{TypeOf}(c) = \tau' \to \tau'' \\ TE \vdash exp : \tau :: \mathsf{e} \leadsto TE' \vdash exp' : \tau' : k'\end{array}}{\begin{array}{l}TE \vdash exp : \tau :: \mathsf{e} \leadsto \\ TE' \vdash \mathtt{coerce}\; exp' \;\mathtt{by}\; c : \tau'' : \mathsf{e}\end{array}} \quad (91)$$

$$\frac{\begin{array}{l} TE \vdash exp_1 : \tau_1 :: \mathsf{e} \leadsto TE' \vdash exp'_1 : \tau'_1 :: \mathsf{e} \\ TE \vdash exp_2 : \tau_2 :: \mathsf{e} \leadsto TE' \vdash exp'_2 : \tau'_2 :: \mathsf{e} \\ \tau_1, \tau_2 \vdash binop : \tau \qquad \tau'_1, \tau'_2 \vdash binop' : \tau' \end{array}}{\begin{array}{l} TE \vdash exp_1\, binop\, exp_2 : \tau :: \mathsf{e} \leadsto \\ TE' \vdash exp'_1\, binop'\, exp'_2 : \tau' : \mathsf{e} \end{array}} \tag{92}$$

The only interesting rules are the ones involving `assume` and `coerce`. In rule 88, note that the type of exp_1 must be the same before and after transformation and that the type of the whole assume 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 $TE(dn) = \texttt{WW}$ but $TE'(dn) = \texttt{YYYY}$. 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* $TE \vdash exp : \tau :: k \leadsto TE' \vdash exp' : \tau' : k'$ *then* $TE \vdash exp : \tau :: k$ *and* $TE' \vdash exp' : \tau' :: k'$

Lemma 5.2 *If* $TE \vdash exp : \tau : k \leadsto TE' \vdash exp' : \tau' : k'$ *then* $TE \leq TE'$, $\tau \leq \tau'$, *and* $k = k'$.

Lemma 5.3 *If* $\tau_1 \leq \tau_2 \leq \tau_3$ *and* $size(\tau_1) = size(v)$ *then* $expand(\tau_2, \tau_3, expand(\tau_1, \tau_2, v)) = expand(\tau_1, \tau_3, v)$.

6 Extension 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 [Haf98b] and AnnoDomini Reference Manual [Haf98a].

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'.[10] Numeric data, however, can also be stored in *binary* or *packed* form.[11]

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

$$\eta ::= \texttt{B} < \tau > \mid \texttt{P} < \tau >$$

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 WWNNNN
 01  DATE-1 PIC XXXXXX VALUE IS '981106'.
 01  DATE-2 PIC 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 `WWNNNNNN` 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 $(k, \tau, \tau_l, \tau_r)$, where k, τ correspond to $\tau :: k$ in Section 3, and τ_l and τ_r 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.

[10] For simplicity, we assume that EBCDIC is the underlying character set here.

[11] Other possibilities, not covered here, are short and long internal floating point representations. AnnoDomini does not allow year data in those, though.

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 `NN`, then `DATE-1` is inferred to have TS2K type `WWNNNN`, since `DATE-2` has type `WWNNNN`, 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.

6.3 Key fields

AnnoDomini identifies and warns of two-digit windowed and user-defined years in key fields of index-sequential files. This is because, for windowed years, dates in the 21st Century appear *before* dates in the 20th Century in the standard (EBCDIC or ASCII) sorting orders.

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 nonyears in AnnoDomini. Thus the `MOVE` statement in

```
*TS2K WWNNNN
 01 DATE-1 PIC XXXXXX.

     MOVE 990909 TO DATE-1
```

causes a type error as it is an apparent Year 2000 Problem (990909 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 990909 IS WWNNNN
     MOVE 990909 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 `WWNNNN` has type `WWNNNNWWNNNNWWNNNNWWNNNN`.

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

[AP89] Habib Abdulrab and Jean-Pierre Pécuchet. Solving word equations. *Journal of Symbolic Computation*, 8(5):499–521, November 1989.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, January 1982.

[Haf98a] Hafnium ApS. *AnnoDomini Version 1.0 for IBM OS/VS COBOL — Reference Manual*, 1st edition, 1998. See http://www.hafnium.com.

[Haf98b] Hafnium ApS. *AnnoDomini Version 1.0 for IBM OS/VS COBOL — User's Guide*, 1st edition, 1998. See http://www.hafnium.com.

[IBM86] IBM. *IBM VS COBOL for OS/VS, Release 2.4*, 5th edition, September 1986. Doc. no. GC26-3857-4.

[Jon98] Capers Jones. *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences.* Addison-Wesley, ACM Press, 1998. ISBN 0-201-30964-5.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[OJ97] 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.

[RFT99] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *These proceedings*, January 1999.

[vDM98] Arie van Deursen and Leon Moonen. Type inference for COBOL systems. To appear in Proc. 5th IEEE Working Conference on Reverse Engineering, Honolulu, Hawaii, October 1998.