TYPE-FLOW ANALYSIS FOR LEGACY COBOL CODE

Alvise Spanò, Michele Bugliesi and Agostino Cortesi
Dipartimento di Scienze Ambientali, Informatica e Statistica, Università Ca' Foscari Venezia, Venice, Italy

Keywords: Static analysis, Analyzer, Type system, Type flow, Flow Type, Type rule, Storage, Picture, Record, Cobol, Label, Variable, Branch, Termination, Status, Convergence, Abstract interpretation, Coercion, Coerce, Environment, Judgement, Substitution, Grammar, Island grammar, Parser, Island parsing, Lexer, Parsing, LALR, Yacc, Lex, F#, .NET, IBM, z/OS, COBOL, COBOL85.

Abstract: Many business applications today still rely on COBOL programs written decades ago that are difficult to maintain and upgrade due to technological limitations and lack of experts in the language. Several companies have been trying to migrate their software base to modern platforms, but code translation is problematic because most business processes implemented are often no longer documented or even known. Applying existing Program Understanding techniques to COBOL could be a way for aiding IT specialists in charge of a porting - but useful raw information must be extracted from the source code in order to get these techniques yield to meaningful results. We believe that the types of variables used in programs are an important part of such raw information and we present an approach based on static analysis of types rather than data. Our system is capable of reconstructing the type-flow of a COBOL program throughout branches, jumps and loops in finite time and to track type information on reused variables occurring in the code. It also detects a number of error-prone situations, type mismatches or misuses and notifies that by means of messages annotated in the code along with types inferred for each variable occurrence.

1 INTRODUCTION

Analyzing COBOL code using type inference techniques has been proposed many times in the last decade and before. From the system first described in (van Deursen and Moonen, 1998) to its later refinement in (Moonen, 2003), giving informative types to COBOL variables seems to be a good way for automatically generating a basic tier of documentation of legacy software (van Deursen and Moonen, 2006) and is also a reliable starting point for further Program Understanding approaches (Kuipers and Moonen, 2000). These systems are quite sophisticate and rely on a number of complex side models and tools aimed to extract properties and information from COBOL programs at a high level of abstraction, thus inevitably omitting several details at a lower language and type level - e.g. how to deal exactly with the many picture formats supported by COBOL and with control constructs that alter the program flow.

In this paper, we propose a light-weight system for typing COBOL with rich yet simple types that pursue a number goals:

1. model the COBOL picture system without approximating storage format information such as computational fields or the amount of digits in a numeric, in order to reconstruct the exact in-memory representation of datatypes and perform precise comparisons among the many formats COBOL supports;
2. deal with what in (van Deursen and Moonen, 2001) is called *pollution* in such a way that no complex relational property system among types is needed, by tracking type alterations that variables are subject to in the following scenarios:
   (a) when data is reused for different purposes in a program: many COBOL programmers are used to this practice in order to save memory and the result is often poor maintainability and error-proneness;
   (b) when the language performs an implicit datatype cast, reformatting values to fit target variables, either at compile-time or at run-time.
3. deal with branches in the program flow that are not statically decidable (i.e. conditional statements) by embedding into the type itself multiple types a variable may possibly assume during the execution.

Spanò A., Bugliesi M. and Cortesi A..
TYPE-FLOW ANALYSIS FOR LEGACY COBOL CODE.
DOI: 10.5220/0003506700640075
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 64-75
ISBN: 978-989-8425-77-5

We introduce a kind of storage type for variables declared as pictures in COBOL and a special *flow-type* for collecting storage types resulting from conditional branches in the program.

On top of that, having to do with a language where `GOTO` and other low-level commands altering the control-flow are frequently used in programs, our system cannot behave like an ordinary type-checker or type-inference algorithm: it is a *type analyzer* able to follow jumps and branches in the code, detect cycles and avoid loops by checking for a convergence in the status of the typing function - in pretty much the same way as many basic techniques of Abstract Interpretation for Static Analysis do (F. Nielson, 1999). The status here consists in a special *topological* environment mapping variable occurrences to their flow-type at that point in the program. Overall, this approach resembles a sort of data-flow analysis where data are actually types rather than values.

A prototype of the system described in this article has been developed in F# for the .NET 4.0 platform and implements a Lex & Yacc tweak for reproducing the behavior of Island Grammars (Moonen, 2001) with the benefit of efficient LALR parsing.

It is able to parse large COBOL source programs (up to many thousands of lines) and to type them generating as output the flow-types annotated at every variable occurrence (i.e. the topological environment mentioned above). Additionally, it produces useful information about type usage in form of error messages, warnings and hints. Again as opposed to a compiler, here errors do not imply a failure: the system adopts a *keep-going* approach and is tolerant to most recoverable error scenarios. All type mismatches or misuses are notified and other hints over possible error-prone situations are signaled; an undefined variable, though, would make the system fail. Thus, we assume to process production code that compiled successfully and does work.

1.1 Overview

Our system do not manipulate COBOL code directly: as other remarkable systems do (van Deursen and Moonen, 1999), we translate COBOL into a more comfortable *intermediate language* (from now on referred to as IL) resembling modern imperative languages without altering COBOL semantics and principles. Notably, what in COBOL speak is referred to as a program (i.e. a compilation unit), here is translated into a procedure, with its own static variable declarations. A COBOL application made up of many units becomes a single large IL program, where the main code shows up as the bottom unnamed block.

Before performing the type analysis, the system must also label all variables occurring in the program with an unique identifier - simply a fresh integer tag. The type analyzer eventually explores the code, statement by statement and recursively descending into expressions, basically performing two operations that affect either the topological or type environment:

1. keeping track of the *current* type(s) of variables by updating flow-types in the type environment;
2. annotating variable occurrences with their flow-type at that point in the program, i.e. creating new bindings in the topological environment.

Assignments and call-by-reference argument applications are two scenarios where variables could be subject to an implicit cast, hence the flow-type of a variable appearing for example at the left-hand side of an assignment must be updated. Conditional constructs, instead, lead to branches in the code exploration, thus the analyzer would produce two parallel results for the two sub-blocks of an `if-then-else` statement: the resulting environments must therefore be merged somehow to reflect that the same variables may possibly have different types after the `if` block and these multiple *choices* are collected in the flow-type itself.

Look at the following example code directly written in IL:

```
{
  x := x + 1;
  if x > 0 then
  {
     x := "foo";
  }
  x := x + 23;
}
where x : num[2] := 11
```

What we want to achieve is reconstructing the types of the program and producing annotations for each occurrence of variable `x` with its type in that point of the code, as well as outputting error and warning messages. For doing that the system has to follow all branches in the control flow and keep updated the type of `x`: by the end of the conditional block we want to to show somehow that `x` might have become a string. And where there is an ambiguous operation, we want the system to recover to a default decision and add a comment about it.

```
{
  (x : num[2]) := (x : num[2]) + 1;
  // [WARNING] possible truncation
  //           detected in assignment:
  //           num[3] :> num[2]

  if (x : num[2]) > 0 then
  {
     (x : alpha[2]) := "foo";
     // [ERROR] truncation detected in
     //         assignment:
     //         alpha[3] :> num[2]
  }

  (x : num[2]) := (x : num[3]|alpha[3]) + 23;
  // [HINT] type of ’x’ is ambiguous in
  //        expression at right-hand of
  //        assignment: assuming
  //        initialization type num[2]
  // [WARNING] possible truncation
  //           detected in assignment:
  //           num[3] :> num[2]
}
where x : num[2] := 11
```

In the first statement, where `x` is incremented by 1, the type of the variable is annotated both in its usage as an expression term and as the target on the left side of an assignment. In the right-hand case its type is the initialization type `num[2]` that appears in the global declaration, which happens to be its current type at the beginning of the program; in the left-hand case `x` should be given a wider numeric type, because the result of the sum of a `num[2]` and a literal whose type is `num[1]` would lead to `num[3]`[1], but it gets truncated in order to fit the initialization type as COBOL run-time would do and therefore, being the resulting storage class still `num`, its final type happens to be equivalent to its initialization type.

The system tracks the type that variable are supposed to have from a type-flow point of view, i.e. as if data movements were tracked across expressions and statements and the type of *what variables are supposed to contain* is recorded.

Encountering the `if` statement makes the analyzer descend into its `then` block: a truncation is detected therein, being `alpha[3]` surely wider than the target type `num[2]`, and the truncated type `alpha[2]` is given to `x`, which fits the initialization type. Such information must be then merged to that previously collected before branching: hence the reason why the type of `x` in the expression at the right hand of the assignment after the `if` block is not a simple type. The flow-type has grown here due to the merge and it now consists of all possible types `x` might have at the moment. That leads to an ambiguous choice when typing the sum operation and so the system needs to recover to the initial type declaration - which might seem odd, but is in fact a viable solution, as in COBOL every variable strictly adheres to its picture declaration, thus falling back to it is not an unsafe decision in case a better information cannot be reconstructed.

1.2 Comparisons and Motivation

As already mentioned, the legacy software analysis system thoroughly presented in (Moonen, 2003)[2] rely on mechanisms for producing information over types that mainly serve Program Understanding techniques, Concept Analysis (Kuipers and Moonen, 2000) and other high-level elaborations. In general, its scope is wider than ours and not entirely overlapping. Nonetheless there is something in common, that is giving somehow interesting types to COBOL variables, that can be taken into consideration for making a comparison with what we believe is the most advanced system for COBOL analysis based on types available to date.

- We translate COBOL into a simpler intermediate language as (van Deursen and Moonen, 1998) does, though without leaving out important language constructs whose behavior is relevant to typing real-world programs, such as `goto`, `perform` and `perform-thru` jump statements, call-by-reference procedure calls and `if` statements.
- Our type syntax is more complete, clearer and open to more orthodox type manipulation, as it doesn’t provide just a plain *AST-ization* of COBOL picture declarations[3].
- The type inference[4] rules given in (van Deursen and Moonen, 2001) are sometimes trivial. We define a type-system that reconstruct more detailed type information, e.g. our type rules for arithmetic operators in table 6 recalculate the resulting type format length in order to include within the type itself as much information as possible about changes in value ranges.
- We don’t infer a type equivalence when two or more types are expected to be the same (as would happen in ML in a homogeneous binary application, for example). Our system rather falls back to a variable initialization type in case a type mismatch or ambiguity is detected. This trade off makes type derivations simpler, does not necessarily imply a loss of information and reflects COBOL run-time semantics better.

[1] In general, a number made of 2 digits plus a number made of 1 digit could possibly lead to a number made of 3 digits, as in $99+9=109$. See type rules for expressions in table 6 for details on how arithmetic operations formally affect numeric type formats.

[2] That is a Ph.D. thesis collecting previous works on the same subject and anticipating some that yet had to come. In general, that system has been proposed several times in more articles with some additions - we might therefore refer to either (van Deursen and Moonen, 1998), (van Deursen and Moonen, 2001), (van Deursen and Moonen, 2000), (Kuipers and Moonen, 2000) or (Moonen, 2003).

[3] Syntax of types in (van Deursen and Moonen, 1998) oddly carries along the variable identifiers and picture format strings *as is*, leaving unclear how the type environment and type comparisons formally related to them.

[4] That system uses the word *inference*, with a clear reference to the world of ML and functional languages, though we’d prefer *reconstruction*, as there is actually no use of type variables and unification for resolving a set of constraints over type equations.

- The system in (Kuipers and Moonen, 2000) represents the inferred set of type relations via a Relational Algebra and resolves them applying an algorithm written in Grok (Holt, 2008): the resolution is actually a *simplification* process performing iterative unification. This approach is rather inefficient and does not take into account type dynamics due to control-flow jumps. Our system performs a code analysis at typing-time by following jumps[5], thus detects a wider range of possible type anomalies and variable reuses.
- According to (van Deursen and Moonen, 2001), *pollution* occurs whenever a type-equivalence involves types that are not equivalent or subtypes: we do not handle this as a special case, but it automatically comes from *non-singleton choices* within flow-types, which are natively supported by our type-system and do not require any further processing.
- Our *subtype* relation deals with the in-memory representation of a wider range of type formats and qualifiers that are very common in COBOL programs, such as all `COMP` fields (translated into native integer, floating point and binary-coded-decimal types), signed/unsigned numeric formats and mixed alphabetic/alphanumeric strings.
- In (van Deursen and Moonen, 2001) there is no mention on how COBOL references[6] are handled, nor on how COBOL run-time data conversions affect type rules of commands that manipulate different picture formats and computational fields (e.g. the `COMPUTE` instruction). A major feature of our system is reproducing such behaviors at typing-time by giving *temporary types* to R-value expressions[7] and eventually *promoting* them to storage types when a type coercion in invoked (see definitions 3.1 and 3.6).

Let's now apply our system to a COBOL code fragment mentioned in (van Deursen and Moonen, 1998) and other papers of the series:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 N000.
  05 N100-N PIC S9(03) COMP-3.
01 TAB000.
  05 TAB100-NAME-PART.
    10 TAB100-POS PIC X(01) OCCURS 40.
  05 TAB100-MAX PIC S9(03) COMP-3 VALUE 40.
  05 TAB100-FILLED PIC S9(03) COMP-3 VALUE ZERO.
01 RAR001-RECORD.
  03 RAR001-VAST.
   05 RAR001-INITIALS PIC X(05).

PROCEDURE DIVISION.
R210-INITIALS.
  MOVE RAR001-INITIALS TO TAB100-NAME-PART
  PERFORM R300-COMPOSE-NAME
  EXIT.

R300-COMPOSE-NAME.
  MOVE TAB100-MAX TO N100.
  MOVE ZERO TO TAB100-FILLED.
  PERFORM UNTIL N100 EQUAL ZERO
    IF TAB100-POS (N100) EQUAL SPACE
      SUBTRACT 1 FROM N100
    ELSE
      MOVE N100 TO TAB100-FILLED
      MOVE ZERO TO N100
    END-IF
  END-PERFORM.
```

The whole code above is translated into one single annotated IL program showing operations among types and warning messages[8]:

```
{
 R210-INITIALS: // main code
 {
  (TAB100 : { NAME-PART : alphanum[40]; .. })
    .NAME-PART :=
      RAR100-RECORD.VAST.INITIALS;
  // [WARNING] reverse subsumption
  // detected in assignment: right-hand
  // type is smaller that left-hand type
  perform R300-COMPOSE-NAME;
  return;
 }

 R300-COMPOSE-NAME:
 {
  N00.N := TAB100.MAX;
  TAB000.FILLED := 000;
   __loop0:
   {
    if N000.N = 000 then goto __loop0_exit;
    else
    {
      if (TAB100 : { NAME-PART :
        { POS : alphanum[1] array[40] };..})
          .NAME-PART.POS[N00.N] = ' '
      // [WARNING] possible access to
      // corrupted data: accessing TAB100
      // with its initialization type
      // but its type had changed to
      // { NAME-PART : alphanum[40]; .. }
      // [WARNING] possible error in array
      // subscript: type 'num.bcd[S3]' has
      // signed format
      then {
        (N000 : { N : num.bcd[S3] }).N :=
            N000.N - 1;
        // [WARNING] possible truncation
        // detected in assignment:
        // num.bcd[S4] :> num.bcd[S3]
      }
      else
      {
        TAB000.FILLED := N000.N;
        (N000 : { N : num.bcd[S3] }).N :=
            000;
      }
      goto __loop0;
    }
   }
   __loop0_exit: {}
 }
}
where N000   : { N : num.bcd[S3] };
```

[5] Until a convergence in the topological environment is detected (see section 2.4).

[6] According to COBOL syntax specification in (IBM, 2009), accessible memory cells are called *references*. We renamed them as *Left Values* in our intermediate language for the sake of symmetry with imperative languages such as C that define them as a sub-class of expressions that can appear at the left side on an assignment and refer to an in-memory value (Kernighan and Ritchie, 1988).

[7] Symmetrically, right values are expressions that can stand on the right side of an assignments, hence evaluate to a temporary value (Stroustrup, 2000).

[8] We omit type annotations where flow-types do not differ from the previous variable occurrence or from its initialization type.

```
TAB000 : {
  NAME-PART : {
    POS : alphanum[1] array[40] };
  MAX : num.bcd[S3] := 40;
  FILLED : num.bcd[S3] := 0 };
RAR001-RECORD : {
  VAST : { INITIALS : alphanum[5] } }
```

(Moonen, 2003) unfortunately does not contain practical code samples of pollution or other anomalies, thus we can't compare how the two systems behave in that regard. As a matter of facts, though, our system seems ultimately more involved in accurate typing and detecting error-proneness rather than program reasoning and collecting statistics.

2 TYPE SYSTEM

2.1 Storage Types and Flow-types

COBOL picture declarations in the Working Storage section of the Data Division define data instances along with their own storage format: they're not type declarations for instantiating data elsewhere as most modern languages do. Our system must of course reproduce this design, but mapping COBOL picture format strings into types. For example, consider the following picture declaration:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
        01 A PIC 9(3) COMP-3 OCCURS 10.
        01 N PIC COMP 9(8).
        01 R1.
                02 R1-S PIC A(2).
                02 R1-B PIC X(3)9(2)A(3).
    01 R2 OCCURS 7.
                02 X PIC S99V9 COMP-2.
```

We translate it into more orthodox type bindings that are quite self-explanatory:

$$
\begin{array}{rcl}
A & : & num_{bcd}[3]\ array[10] \\
N & : & num_{int_{32}}[8] \\
R1 & : & \{S : alpha[2];\ B : alphanum[8]\} \\
R2 & : & \{X : num_{float_{64}}[S2.1]\}\ array[7]
\end{array}
$$

Picture format strings are mapped into either numeric, pure alphabetic or alphanumeric types according to their structure; arrays and records are also first-class citizens of the type language in our system and can therefore be nested at will, yielding to types that resemble those of modern functional languages. Moreover, numeric types carry along detailed information on their in-memory representation at machine level, sign and length of both integral and fractional parts; while arrays and alphabetic/alphanumeric strings simply carry their length.

The full syntax of the type-system follows:

$$
\begin{array}{llll}
\tau & := & & \text{storage types} \\
 & & num_q[\rho] & \text{numeric} \\
 & | & alpha[n] & \text{alphabetic string} \\
 & | & alphanum[n] & \text{alphanumeric string} \\
 & | & \tau\ array[n] & \text{array} \\
 & | & \{x_1 : \tau_1\ ..\ x_n : \tau_n\} & \text{record} \\
 & & & \\
\sigma & := & & \text{temporary types} \\
 & & \tau & \\
 & | & bool & \text{boolean} \\
 & | & \overline{num}[\rho] & \text{abstract numeric} \\
 & & & \\
q & := & & \text{numeric storage qualifier} \\
 & & ascii & \text{display } \textit{or} \text{ ASCII} \\
 & | & bcd & \text{binary-packed decimal} \\
 & | & int_{16|32|64} & \text{native integer} \\
 & | & float_{32|64} & \text{native float} \\
 & & & \\
\rho & := & [S]n.d & \text{numeric format} \\
 & & & \\
\varphi & := & \{\tau_1\ ..\ \tau_k\} & \text{flow-item } \textit{or} \text{ choice} \\
 & & & \\
\Phi & := & \langle\varphi;\tau\rangle & \text{flow-type}
\end{array}
$$

$$\texttt{where } k \geq 1,\ n \in \mathbb{N}^*,\ d \in \mathbb{N}$$

There are two distinct classes of types:

- τ is the type of storage variables and L-values in general, i.e. the type of data that stands in memory and has some representation[9];
- σ, where $\sigma \supset \tau$, is the type given to expression terms only and is never produced by picture translation, serving just as a temporary light-weight type whose in-memory representation is yet to be known in that context.

As typing rules will show, such temporary types are eventually promoted to ordinary τ types as soon as the storage type of an actual variable becomes known, for example when an expression that's given a temporary is then assigned to a L-value or passed as a call-by-ref argument in an procedure call.

Finally, a *flow-type* is a simply a pair of possibly multiple storage types (those a variable may concurrently have following statically undecidable conditional branches in the program flow, as stated in section 1) and an additional single storage type, which is the type initially declared for the variable in the global environment. We'll be often referring to the first component of a flow-type as *flow-item* or *choice*.

[9] ASCII is the default qualifier for numeric types: whenever unspecified this one holds, as in `num[3]` for example.

2.2 Environments

Type judgements operate over a number of environments.

Type Environment Γ maps variable identifiers to flow-types: this environment is initially populated with global type declarations and its bindings are then updated when the current flow-type changes during typing. It contains bindings of form $x : \Phi$.

Topological Environment Θ collects all annotations produced by the type analyzer by mapping labeled variable occurrences x^{κ} to its flow-type at that program point. It represents also the status of the typing function in the detection of loop termination. It contains bindings of form $x^{\kappa} : \Phi$.

Procedure Environment Π maps procedure names to signatures (see definition 3.8). It contains bindings of form $p \mapsto \langle y_1 : \tau_1^p .. y_n : \tau_n^p ; \Gamma_p \rangle$.

Block Environment Σ maps label identifiers to blocks of statements. It contains bindings of form $l \mapsto \underline{\{} st_1 .. st_n \underline{\}}$.

Type environments also support a binary function `merge`, used by rules IF and IF-ELSE, which recompacts the bindings collected in separate environments by the typing of program branches, as informally introduced by section 1. Such `merge` function is alone responsible for the growth of the flow-item component within a flow-type.

2.3 Coercion of L-Values

Take the following example:

```
{
a[0].l := "boo";
}
where a : { l : num[2]; m : alpha[10] } array[5]
```

And its annotated form resulting from the type analysis:

```
{
(a : { { l : alpha[2]; m : alpha[10] } array[5])[0].l
    := "boo";
}
where a : { l : num[2]; m : alpha[10] } array[5]
```

The literal `"boo"` having type `alpha[3]` is assigned to field `l` of a record within a cell of an array. The flow-type of variable `a` needs to be updated here somehow with the type of the right-hand of the assignment - and of course it's not to `a` that such type must naively be given, but to the record field `l` nested within. Nonetheless the environment binds variable identifiers to flow-types, thus there is no way to update the type of a record label (as `l` in our case) or of an array cell alone. Therefore the whole type of a variable must be updated keeping the original structure layout and replacing the appropriate bit nested within it. Hence, the whole type of `a` in the example becomes `{ l : alpha[2]; m : alpha[10] } array[5]`.

This shows also that the expected type `alpha[3]` of the literal `"boo"` has been adapted to *fit* into the initialization type `num[2]`: coercion in assignments needs therefore both to replace a piece of a type and to resize it accordingly, keeping the original storage class (`num` in our example) and recalculating the format in such a way that the overall size of the new resulting type fits the initialization one.

For this reasons, judgements for L-value terms are slightly different: $\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv : \tau \backslash \theta^{x^{\kappa}} \triangleright \Theta_1$ means that the L-value lv has a storage type τ coercible by the substitution $\theta^{x^{\kappa}}$, where x is the root variable of lv (formally $x = \Re(lv)$ as of definition 3.7) and x^{κ} is its labeled occurrence. θ is a function from storage types to storage types that can be passed by typing rules that need to update the type of the root variable of an L-value to the coerce function $\mathscr{C}$ (see definition 3.6), which performs the proper fit operation among other things.

2.4 Loops and Convergence

As informally stated in section 1, the type analyzer follows `goto` and `perform` statements unless already visited and a convergence in the status of the typing function is detected. In subsection 2.2 we said that this status actually consists of the topological environment Θ. The typing function at step i of the analysis can be defined as a function taking the statement fetched at that step and the topological environment:

$$\mathcal{T}_i(st_{B,p}, \Theta_i) = \Theta_{i+1}$$

where $st_{B,p}$ is the statement located within block B at position p.

Each time the typing function encounters a jump statement, it performs a number of operations. Say a jump statement $st_{A,q} \equiv$ `goto` l is encountered by $\mathcal{T}$ at step i while typing block $A = \underline{\{} st_{A,1} .. st_{A,n} \underline{\}}$ (with $q \in [1,n]$):

1. it saves the topological environment Θ_i built up so far, binding it to the current program location;
2. it looks up the destination block of statements from the block environment, hence $B = \underline{\{} st_{B,1} .. st_{B,m} \underline{\}} = \Sigma(l)$;
3. it continues the analysis from there, i.e. from statement $st_{B,1}$.

Let's consider that later at step j (obviously $j > i$) $\mathcal{T}$ reaches the jump statement $st_{A,q}$ again: then the new current topological environment Θ_j is compared against Θ_i, which had formerly been saved at that program location. If $\Theta_j \sqsubseteq \Theta_i$ (see definition 3.11) then it means that no further type information has been collected during the second pass and we can therefore assume that the analysis can safely skip the jump statement $st_{A,q}$ and continue from $st_{A,q+1}$. Else, the new topological environment Θ_j is saved (replacing the old Θ_i previously stored) and the analysis continues from the jump statement destination $st_{B,1}$ again.

We observed that even in complex *spaghettish* scenarios with several `goto` statements within nested conditional blocks the system detects a convergence pretty soon: averagely in 1 and anyway in up to 3 reiterations of the same piece of code. The reason is twofold:

- the topological environment cannot by definition be subject to binding removal, hence $\forall x^\kappa \in \Theta_i.\ x^\kappa \in \Theta_{i+1}$ at any given step i;
- flow-types bound to variable occurrences in the topological environment can only grow - they can never diminish in width. Given we're dealing with types and not values, the stability is certain: storage types of variables do not change from pass to pass for obvious reasons and the only thing that could change and modify the status Θ of the typing function $\mathcal{T}$ is the flow-item φ part of flow-types bound to variable occurrences. φ is defined as a set of storage types τ in table 2.1 and it is subject to a single operation: the `merge` function as of definition 3.9, which basically consists in a set-union between flow-items. Duplicate types can therefore never occur and no element could be removed.

2.5 Ambiguity

Having non-singleton flow-items within flow-types is indeed a central feature of this system, signaling that the programmer reused a variable in different ways along the program. Nonetheless, that makes judgments for L-values problematic: how are we supposed to type an L-value appearing in an expression, for instance, if its current flow-type says that it could have many storage types at the same time? In fact, we can't - that's exactly what flow-types stand for: detecting anomalous scenarios that may lead to unwanted results at run-time.

In our code example in section 1, imagine the system had output another hint message for the ambiguous statement claiming that among the possible choices `num[3]` would have been suitable. And the typing then proceeded selecting `num[3]` as candidate, leading to a different type for `x` - not the one shown in the original example.

```
{
(x : num[3]) := (x : num[2]) + 1;
// [WARNING] possible truncation detected
//           in assignment:
//              num[3] :> num[2]

if (x : num[3]) > 0 then
{
    (x : alpha[3]) := "foo";
    // [ERROR] truncation detected in
    //         assignment:
    //          alpha[3] :> num[2]
}

(x : num[6]) := (x : num[3]|alpha[3]) + 23;
// [HINT] type of ’x’ is ambiguous in
//        expression at right-hand of
//        assignment: choice num[3]
//        would fit
// [WARNING] possible truncation detected
//           in assignment:
//              num[6] :> num[2]
}
where x : num[2] := 11
```

What if more than one type was suitable, though? The flow-type would literally explode for tracking several implications among possible typing paths and in the end it would hardly be useful.

Our proposal in such situations is to do the simplest thing: falling back to the initial type of the variable; and of course notifying the choice with a hint message. However, this leads to a duplication of the type rule for variables, as table 4 shows.

3 FORMAL SPECIFICATION

In this section we give the full specification of the type-system described in section 2. A number of definitions is given below that will be used by type rules.

Definition 3.1 (Promote). *The promotion $[\![\sigma]\!]^\tau$ of a temporary type σ to a storage type τ produces a storage type that transform σ into a storable type inheriting the characteristics of τ. The promotion function is defined as follows (top-down closest-match rule on the left hand holds):*

$$\begin{array}{rcl}
[\![\overline{num}[\rho_2]]\!]^{num_q[\rho_1]} & = & num_q[\rho_2] \\
[\![\overline{num}[\rho]]\!]^{\tau} & = & num_{ascii}[\rho] \\
[\![bool]\!]^{\tau} & = & [\![\overline{num}[1.0]]\!]^{\tau} \\
[\![\tau_2]\!]^{\tau_1} & = & \tau_2
\end{array}$$

Definition 3.2 (Representation). *We define a function $\texttt{rep} : \tau \to \mathbb{N}$ for calculating the in-memory byte size of a storage type:*

$$\begin{array}{rcl}
\texttt{rep}(num_{ascii}[n.d]) & = & n+d \\
\texttt{rep}(num_{bcd}[n.d]) & = & \lceil \frac{n+d+1}{2} \rceil \\
\texttt{rep}(num_{int_b}[\rho]) & = & b/8 \\
\texttt{rep}(num_{float_b}[\rho]) & = & b/8 \\
\texttt{rep}(alpha[n]) & = & n \\
\texttt{rep}(alphanum[n]) & = & n \\
\texttt{rep}(\tau\ array[n]) & = & \texttt{rep}(\tau) * n
\end{array}$$

$$\texttt{rep}(\{x_1 : \tau_1 \, .. \, x_n : \tau_n\}) \quad = \quad \sum_{i=1}^{n} \texttt{rep}(\tau_i)$$

Definition 3.3 (Subtype). *We define a total-order between storage types such that the relation $\tau_1 \preceq \tau_2$ holds when $\texttt{rep}(\tau_1) \leq \texttt{rep}(\tau_2)$.*

Definition 3.4 (Var-Bound Substitution). *A substitution θ^{x^κ} is a function from storage types to storage types that carries along a labeled identifier x^κ which stands for the variable occurrence whose type the substitution has been built from and is supposed to replace*[10].

Definition 3.5 (Fit). *The fit $\lfloor \tau_1 \rfloor_{\tau_2}$ of a storage type τ_1 to a storage type τ_2 produces a storage type whose storage class is equivalent to that of τ_1 and whose size fits into that of τ_2. The fit function is defined as follows:*

$$\begin{array}{rcl}
\lfloor num_q[\rho] \rfloor_\tau & = & num_q[\rho'] \\
 & & \textit{for some } \rho' \textit{ such that} \\
 & & \texttt{rep}(num_q[\rho']) = \texttt{rep}(\tau) \\[1ex]
\lfloor alpha[n] \rfloor_\tau & = & alpha[n'] \\
 & & \textit{for some } n' \textit{ such that} \\
 & & \texttt{rep}(alpha[n']) = \texttt{rep}(\tau) \\[1ex]
\lfloor alphanum[n] \rfloor_\tau & = & alphanum[n'] \\
 & & \textit{for some } n' \textit{ such that} \\
 & & \texttt{rep}(alphanum[n']) = \texttt{rep}(\tau) \\[1ex]
\lfloor \tau_a \; array[n] \rfloor_\tau & = & \tau'_a \; array[n'] \\
 & & \textit{for some } \tau'_a \textit{ and } n' \textit{ such that} \\
 & & \texttt{rep}(\tau'_a \; array[n']) = \texttt{rep}(\tau) \\[1ex]
\lfloor \{l_1 : \tau_1 .. l_n : \tau_n\} \rfloor_\tau & = & \{l_1 : \tau'_1 .. l_n : \tau'_n\} \\
 & & \textit{for some } \tau'_1 .. \tau'_n \textit{ such that} \\
 & & \texttt{rep}(\{l_1 : \tau'_1 .. l_n : \tau'_n\}) = \texttt{rep}(\tau)
\end{array}$$

Definition 3.6 (Coerce). *The coerce function $\mathscr{C}$ updates the given type and topological environments by applying a given substitution function θ^{x^κ} to the types a given flow-item φ consists of; it produces a new pair of form $\langle \Gamma; \Theta \rangle$ consisting of the type and topological environments endowed with updated bindings for the variable x and the occurrence label κ annotated on the substitution function θ^{x^κ} itself:*

$$\mathscr{C}(\varphi, \theta^{x^\kappa}, \Gamma, \Theta) \quad = \quad \langle \Gamma, x : \Phi'; \Theta, \kappa : \Phi' \rangle$$

where

$$\begin{array}{rcl}
\langle \varphi; \tau_x \rangle & = & \Gamma(x) \\
\Phi' & = & \langle \{\tau'_i \mid \forall \tau_i \in \varphi . \tau'_i = \lfloor \theta^{x^\kappa}(\tau_i) \rfloor_{\tau_x}\}; \tau_x \rangle
\end{array}$$

Definition 3.7 (Root Variable). *Given an L-value lv, its root variable is the identifier x evaluated by the recursive function defined as:*

$$\begin{array}{rcl}
\Re(x) & = & x \\
\Re(lv[e]) & = & \Re(lv) \\
\Re(lv.l) & = & \Re(lv)
\end{array}$$

Definition 3.8 (Signature). *A signature is a pair $\langle Y_p; \Gamma_p \rangle$ where p is a procedure name, Y_p are its formal parameters $y_1 : \tau_1^p \, .. \, y_n : \tau_n^p$ and Γ_p is the output type environment returned by typing the body of p.*

Definition 3.9 (Type Environment Merge). *The binary function $\oplus$ merges two given type environments into one as follows:*

$$\Gamma_1 \oplus \Gamma_2 \quad = \quad \Gamma^* \cup (\Gamma_1 \backslash \Gamma_2) \cup (\Gamma_2 \backslash \Gamma_1)$$

where

$$\begin{array}{rlcl}
\Gamma^* = & \{x : \langle \varphi_1 \cup \varphi_2; \tau_1 \rangle & | & \Gamma_1(x) = \langle \varphi_1; \tau_1 \rangle \\
 & & \wedge & \Gamma_2(x) = \langle \varphi_2; \tau_2 \rangle \\
 & & \wedge & \tau_1 = \tau_2\}
\end{array}$$

Definition 3.10 (Partial Ordering of Flow-Types). *We define a partial order between flow-types such that $\Phi_1 \sqsubseteq \Phi_2$ holds when, let $\Phi_1 = \langle \varphi_1; \tau_1 \rangle$ and $\Phi_2 = \langle \varphi_2; \tau_2 \rangle$, then $\varphi_1 \subseteq \varphi_2 \wedge \tau_1 = \tau_2$.*

Definition 3.11 (Partial Ordering of Topological Environments). *We define a partial order between topological environments such that $\Theta_1 \sqsubseteq \Theta_2$ holds when $\forall x : \Phi_1 \in \Theta_1 . \; x \in dom(\Theta_2) \wedge \Phi_1 \sqsubseteq \Phi_2$, where $\Phi_2 = \Theta_2(x)$.*

3.1 Type Rules

Syntax-directed type rules are divided by category. Rules for Programs are shown in table 1, for Statements in table 2, for Expressions in table 6, for Arguments in table 3 and for Literals in table 5.

Most judgements give a type to a term of the language in a context consisting of a tuple of environments and output the updated Γ and Θ, except judgements for Statements and Programs that give no type and simply update the environments. As a general rule, the topological environment Θ is always forwarded to and returned by all judgements (except literals), because flow-types must be annotated recursively on each variable occurring in any subterm of the program. While the type environment Γ is output only by rules that actually update it: consider it as returned back untouched when there's no mention of it among outputs.

Judgements are of a number of forms, each

[10] Substitution functions are recursively defined by type rules for L-Values as shown in table 4. They're meant for generically replacing a term nested within a storage type of arbitrary complexity by reproducing its original structure of recursive type terms and changing the innermost part only.

Table 1: Type Rules for Programs and Body.

Main

$$\frac{\Pi;\emptyset;\Theta_0 \vdash_B B \triangleright \Gamma;\Theta_1}{\Pi;\Theta_0 \vdash_P B \triangleright \Theta_1}$$

Proc

$$\frac{\begin{array}{c}\Gamma_p = \emptyset, y_1 : \langle\{\tau_i^p\};\tau_i^p\rangle \;..\; y_n : \langle\{\tau_n^p\};\tau_n^p\rangle \\ \Pi;\Gamma_p;\Theta_0 \vdash_B B \triangleright \Gamma'_p;\Theta_1 \\ \Pi, p \mapsto \langle y_1 : \tau_1^p \;..\; y_n : \tau_n^p;\Gamma'_p\rangle;\Theta_1 \vdash_P P \triangleright \Theta_2\end{array}}{\Pi;\Theta_0 \vdash_P \underline{\texttt{proc}}\; p(y_1 : \tau_1^p \;..\; y_n : \tau_n^p)\; B\; \underline{\texttt{in}}\; P \triangleright \Theta_2}$$

Body

$$\frac{\begin{array}{c}\forall i \in [1,n].\ \Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{lit} lit_i \;:\; \sigma_i \wedge [\![\sigma_i]\!]^{\tau_i} \preceq \tau_i \\ \Pi;\emptyset;\Gamma_0, x_1 : \langle\{\tau_1\};\tau_1\rangle \;..\; x_n : \langle\{\tau_n\};\tau_n\rangle;\Theta_0 \vdash_{st} st \triangleright \Gamma_1;\Theta_1\end{array}}{\Pi;\Gamma_0;\Theta_0 \vdash_B st\; \underline{\texttt{where}}\; x_1 : \tau_1 := lit_1 \;..\; x_n : \tau_n := lit_n \triangleright \Gamma_1;\Theta_1}$$

Table 2: Type Rules for Statements.

Assign

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e \;:\; \sigma_e \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma_0;\Theta_1 \vdash_{lv} lv \;:\; \tau_{lv}\backslash\theta^{x^\kappa} \triangleright \Theta_2 \\ x^\kappa = \Re(lv) \\ \langle\Gamma_1;\Theta_2\rangle = \mathscr{C}([\![\sigma_e]\!]^{\tau_{lv}}, \theta^{x,\kappa}, \Gamma_0, \Theta_2)\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} lv := e \triangleright \Gamma_1;\Theta_2}$$

If

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e \;:\; bool \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma_0;\Theta_1 \vdash_{st} st \triangleright \Gamma_1;\Theta_2 \\ \Gamma_2 = \Gamma_0 \oplus \Gamma_1\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{if}}\; e\; \underline{\texttt{then}}\; st_1 \triangleright \Gamma_2;\Theta_2}$$

If-Else

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e \;:\; bool \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma_0;\Theta_1 \vdash_{st} st_1 \triangleright \Gamma_1;\Theta_2 \\ \Pi;\Sigma;\Gamma_0;\Theta_2 \vdash_{st} st_2 \triangleright \Gamma_2;\Theta_3 \\ \Gamma_3 = \Gamma_1 \oplus \Gamma_2\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{if}}\; e\; \underline{\texttt{then}}\; st_1\; \underline{\texttt{else}}\; st_2 \triangleright \Gamma_3;\Theta_3}$$

Perform

$$\frac{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \Sigma(l) \triangleright \Gamma_1;\Theta_1}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{perform}}\; l \triangleright \Gamma_1;\Theta_1}$$

Perform-Thru

$$\frac{\forall i \in [a,b).\ \Pi;\Sigma;\Gamma_{i-a};\Theta_{i-a} \vdash_{st} \Sigma(l_i) \triangleright \Gamma_{i-a+1};\Theta_{i-a+1}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{perform}}\; l_a\; l_b \triangleright \Gamma_{b-a-1};\Theta_{b-a-1}}$$

Goto

$$\frac{\begin{array}{c}l_n \in dom(\Sigma) \mid \nexists l_m.\ m > n \\ \forall i \in [k,n].\ \Pi;\Sigma;\Gamma_{i-k};\Theta_{i-k} \vdash_{st} \Sigma(l_i) \triangleright \Gamma_{i-k+1};\Theta_{i-k+1}\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{goto}}\; l_k \triangleright \Gamma_{n-k};\Theta_{n-k}}$$

Call

$$\frac{\begin{array}{c}\langle y_1 : \tau_1^p \;..\; y_n : \tau_n^p;\Gamma_p\rangle = \Pi(p) \\ \forall i \in [1,n].\ \Pi;\Sigma;\Gamma_{i-1};\Theta_{i-1} \vdash_a a_i \;:\; \tau_i^p \triangleright \Gamma_i;\Theta_i\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} p\underline{(}a_1 \;..\; a_n\underline{)} \triangleright \Gamma_n;\Theta_n}$$

Block

$$\frac{\begin{array}{c}\Sigma' = \Sigma,\ l_j \mapsto \underline{\{}st_{j,1} \;..\; st_{j,n_j}\underline{\}}..\ \ (\forall j \mid st_{0,j} \equiv l_j\underline{:}\underline{\{}st_{j,1} \;..\; st_{j,n_j}\underline{\}}) \\ \forall i \in [1,n].\ \Pi;\Sigma';\Gamma_{i-1};\Theta_{i-1} \vdash_{st} st_i \triangleright \Gamma_i;\Theta_i\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} [l_0\underline{:}]\ \underline{\{}\ st_{0,1} \;..\; st_{0,n_0}\ \underline{\}} \triangleright \Gamma_n;\Theta_n}$$

syntactic category having its own, though most of them are quite self-explanatory. For example, $\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : \sigma \triangleright \Theta_1$ denotes that, in the given environments, expression e is given a temporary type σ and the topological environment Θ_1 is output.

Judgements for Arguments probably need some extra words. Call-by-ref calls need to update the type environment of the the caller because the flow-type of argument might be modified by the invoked procedure. The procedure environment Π stores the type environment Γ_p for each procedure p of the program, thus the flow-type of a variable passed by reference to p can be updated according to the flow-type of the corresponding formal parameter bound in Γ_p. Such update is carried on by the coerce function $\mathscr{C}$, as shown by rule BYREF in table 3. The mechanism resembles that in rule ASSIGN in table 2: call-by-reference argument application indeed behaves like an assignment (call-by-value doesn't).

Rules for Arguments have form $\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_a a : \tau_i^p \triangleright \Gamma_1;\Theta_1$, meaning that, in the given environments, the actual argument a has type τ_i^p, which is the type of the i-th formal parameter of procedure p.

As a final notice, for the sake of simplicity we assume that all labels in the program are named in order of occurrence: if l_n and l_m are two labels and $m > n$,

Table 3: Type Rules for Arguments.

ByVal

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e : \sigma \triangleright \Theta_1 \\ [\![\sigma]\!]^{\tau} \preceq \tau_i^p\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_a \underline{\texttt{val}}\, e : \tau_i^p \triangleright \Gamma_0;\Theta_1}$$

ByRef

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{lv} lv : \tau' \backslash \theta^{x^{\kappa}} \triangleright \Theta_1 \\ x = \Re(lv) \qquad \tau' \preceq \tau_i^p \\ \langle y_1 : \tau_1^p \mathinner{..} y_n : \tau_n^p ; \Gamma_p \rangle = \Pi(p) \\ \langle \varphi_i^p ; \tau_i^p \rangle = \Gamma_p(y_i) \\ \langle \Gamma_1;\Theta_2 \rangle = \mathscr{C}(\varphi_i^p, \theta^{x,\kappa}, \Gamma_0, \Theta_1)\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_a \underline{\texttt{ref}}\, lv : \tau_i^p \triangleright \Gamma_1;\Theta_2}$$

Table 4: Type Rules for L-Values.

Var-Init

$$\frac{\begin{array}{c}\Gamma(x) = \Phi = \langle \{\tau_1\ \tau_2 \mathinner{..} \tau_n\}; \tau_0 \rangle \\ \Theta_1 = \Theta_0, x^{\kappa} : \Phi \qquad \theta^{x^{\kappa}}(\overline{\tau}) = \overline{\tau}\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} x^{\kappa} : \tau_0 \backslash \theta^{x^{\kappa}} \triangleright \Theta_1}$$

Var-Curr

$$\frac{\begin{array}{c}\Gamma(x) = \Phi = \langle \{\tau_1\}; \tau_0 \rangle \\ \Theta_1 = \Theta_0, x^{\kappa} : \Phi \qquad \theta^{x^{\kappa}}(\overline{\tau}) = \overline{\tau}\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} x^{\kappa} : \tau_1 \backslash \theta^{x^{\kappa}} \triangleright \Theta_1}$$

Subscript

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : \overline{num}[\rho] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_{lv} lv : \tau\, array[n] \backslash \theta_0^{x^{\kappa}} \triangleright \Theta_2 \\ x = \Re(lv) \\ \theta^{x^{\kappa}}(\overline{\tau}) = \theta_0^{x^{\kappa}}(\overline{\tau}\, array[n])\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv\underline{[}e\underline{]} : \tau \backslash \theta^{x^{\kappa}} \triangleright \Theta_2}$$

Select

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv : \{z_1 : \tau_1 \mathinner{..} z : \tau \mathinner{..} z_n : \tau_n\} \backslash \theta_0^{x^{\kappa}} \triangleright \Theta_1 \\ x = \Re(lv) \\ \theta^{x^{\kappa}}(\overline{\tau}) = \theta_0^{x^{\kappa}}(\{z_1 : \tau_1 \mathinner{..} z : \overline{\tau} \mathinner{..} z_n : \tau_n\})\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv\underline{.}z : \tau \backslash \theta^{x^{\kappa}} \triangleright \Theta_2}$$

then l_m appear *below* l_n in the program. That makes type rules for jump statements simpler.

4 RESULTS AND CONCLUSIONS

Our implementation of the system, as already said, can detect a number of type misuses and mismatches besides producing flow-type annotations for each variable occurrence. At the time of writing several tests have been run over real-world legacy business code, mainly written in COBOL85 for z/OS during the 1990s and owned by a big local company within the mechanical vehicle industry. The following considerations and evidences have emerged:

- variable reuse involves up to 30% of overall variable usage in COBOL programs
  - nearly 90% of these, though, accumulate less than 5 storage types simultaneously within their flow-type; averagely 3
  - remaining 10% however unlikely grow wider than 8
  - 75% of non-singleton flow-types indicates reuse of numeric types
    - 80% of these come from in-place arithmetic operations possibly exceeding target variable space, such as the typical scenario `(x : num[3]) := (x :  num[2]) + 1`
    - probably few of such operations are potentially risky at run-time, because programmers typically declare pictures wider than actually needed for their numerics
    - remaining 20% are re-assignments or data movements, i.e. assignments where variables on the right-hand do not appear in left-hand
- 25% of non-singleton flow-types indicates reuse of non-numeric types
  - 70% of these are alphanumeric-strings-to-array type switches and viceversa
  - 10% involve complex data types, such as nested records overlapping arrays
  - only 2% occurs between incompatible types, thus probably leading to data corruption and bugs
  - remaining 18% involve data movements implying no truncation, thus might be bad code but does not lead to run-time unwanted behaviors
- 80% of jump statements require up to 3 visits (including the first one, hence 2 re-visits) to reach

Table 5: Type Rules for Literals.

NUM-U

$$\frac{n = len(n_1) \qquad d = len(n_2)}{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} n_1[.n_2] \;:\; \overline{num}[n.d]}$$

NUM

$$\frac{n = len(n_1) \qquad d = len(n_2)}{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} -n_1[.n_2] \;:\; \overline{num}[Sn.d]}$$

STRING-ALPHANUM

$$\frac{\begin{array}{c}\{0\,..\,9\} \cap \texttt{"str.."} \neq \emptyset \\ n = len(str)\end{array}}{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} \texttt{"str.."} \;:\; alphanum[n]}$$

STRING-ALPHA

$$\frac{n = len(\texttt{"str.."})}{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} \texttt{"str.."} \;:\; alpha[n]}$$

TRUE

$$\frac{}{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} true \;:\; bool}$$

FALSE

$$\frac{}{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} false \;:\; bool}$$

Table 6: Type Rules for Expressions.

DEMOTE-NUM

$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e \;:\; num_q[\rho] \triangleright \Theta_0}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e \;:\; \overline{num}[\rho] \triangleright \Theta_0}$$

LV

$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv \;:\; \tau \backslash \theta^{x^{\kappa}} \triangleright \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e lv \;:\; \tau \triangleright \Theta_1}$$

LIT

$$\frac{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} lit \;:\; \sigma}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e lit \;:\; \sigma \triangleright \Theta_0}$$

NEG-S

$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e \;:\; \overline{num}[Sn.d] \triangleright \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e -e \;:\; \overline{num}[Sn.d] \triangleright \Theta_1}$$

NEG-U

$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e \;:\; \overline{num}[n.d] \triangleright \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e -e \;:\; \overline{num}[Sn.d] \triangleright \Theta_1}$$

NOT

$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e \;:\; bool \triangleright \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e \underline{\texttt{not}}\; e \;:\; bool \triangleright \Theta_1}$$

PLUS-U

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; \overline{num}[n_1.d_1] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; \overline{num}[n_2.d_2] \triangleright \Theta_2 \\ n = max(n_1,n_2) \qquad d = max(d_1,d_2)\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 + e_2 \;:\; \overline{num}[Sn.d] \triangleright \Theta_2}$$

PLUS-MINUS-S

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; \overline{num}[S_1 n_1.d_1] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; \overline{num}[S_2 n_2.d_2] \triangleright \Theta_2 \\ S = S_1 \vee S_2 \qquad n = max(n_1,n_2)+1 \\ d = max(d_1,d_2)\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1\,(+\mid -)\,e_2 \;:\; \overline{num}[Sn.d] \triangleright \Theta_2}$$

MULT

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; \overline{num}[S_1 n_1.d_1] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; \overline{num}[S_2 n_2.d_2] \triangleright \Theta_2 \\ S = S_1 \vee S_2 \\ n = n_1 + n_2 \qquad d = d_1 + d_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 * e_2 \;:\; \overline{num}[Sn.d] \triangleright \Theta_2}$$

DIV

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; \overline{num}[S_1 n_1.d_1] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; \overline{num}[S_2 n_2.d_2] \triangleright \Theta_2 \\ S = S_1 \vee S_2 \\ n = n_1 + d_2 \qquad d = d_1 + n_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \,/\, e_2 \;:\; \overline{num}[Sn.d] \triangleright \Theta_2}$$

BIN-REL-NUM

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; \overline{num}[S_1 n_1.d_1] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; \overline{num}[S_2 n_2.d_2] \triangleright \Theta_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1\; op_r\; e_2 \;:\; bool \triangleright \Theta_2}$$

BIN-REL-ALPHANUM

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; alphanum[n1] \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; alphanum[n2] \triangleright \Theta_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1\; op_r\; e_2 \;:\; bool \triangleright \Theta_2}$$

BIN-LOGIC

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 \;:\; bool \triangleright \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 \;:\; bool \triangleright \Theta_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1\; op_l\; e_2 \;:\; bool \triangleright \Theta_2}$$

a convergence in the typing function status; averagely 2, hence 1 re-visit

– 98% of those are actually pretty ordinary loops coming from COBOL iterative constructs; just 2% are weird custom cycles created by the programmer

– remaining 20% of jump statements need anyway up to 5 visits before a convergence occurs

   – 70% the latter are actually just nested conditional loops that COBOL iterative constructs cannot express and are explicitly written by programmers via `IF` and `GOTO` statements.

All this suggests that type-flow analysis is actually able to detect a number of possible errors in COBOL programs coming from bad reuse of variables or incompatible data movements. Either ways lead to data truncation or corruption, which are the major sources of run-time bugs. And, by the way, the statistics above do not differ a lot from those collected and shown by (Moonen, 2003).

In the following example we show how a data move from a smaller type to a larger one might lead to unwanted scenarios where previous data has not been replaced by new one:

```
{
  a := r;
  // [WARNING] reverse subsumption detected in
               assignment: right-hand type
               is smaller that left-hand type

  n := a[3];
  // [WARNING] possible access to corrupted
               data: accessing ’a’ with its
               initialization type
               ’alphanum[2] array[4]’ but its
               content and type have changed
}
where a : alphanum[2] array[4];
      r : { x : num[3];
            y : alphanum[2];
            z : num[2] };
      n : alphanum[2]
```

Record `r` is 7-bytes long and array `a` is 8 bytes, therefore, once `r` is copied into `a`, accesses to the latter as its initialization array type would lead to unwanted data in case the last byte is accessed. Although in the example we used a literal in the subscript, in general the analyzer cannot know what is accessed and therefore the warning is output.

For this matter, static evaluation of constant expressions has been implemented in our prototype, even though we haven’t considered it in this article - that would avoid the warning in case the assignment was `n := a[1]` and we generally noted that it does slightly reduce the number of messages logged by the analyzer, overall. Also, a GUI front-end is under development for letting users browse annotated source programs and understand complex flow-type more easily.

Finally, we’re considering to extend the system with the following features:

- dealing with unknown statements in some interesting way, type-wise, such as adding weak types to the type-system indicating that type assumptions might get broken whenever a variable is used by a COBOL command whose semantics are unknown
- support for COBOL language extensions such as SQL, introducing the notion of cursor and table types within the system for detecting possible inconsistencies between declared records and actual row layout in the database
- adding some form of data-flow analysis over value domains and ranges
- designing some custom Program Understanding approaches, such as pattern recognition over identifier names or code snippets for making the system aware of typical COBOL programming trends, styles, practices and design patterns

REFERENCES

F. Nielson, H.R. Nielson, C. H. (1999). *Principles of Static Analysis*. Springer Verlag.

Holt, R. C. (2008). WCRE 1998 most influential paper: Grokking software architecture. In *WCRE (Working Conference on Reverse Engeneering)*, pages 5–14. IEEE.

IBM (2009). Cobol z/OS language reference. Website. http://publib.boulder.ibm.com/ infocenter/ pdthelp/ v1r1/ index.jsp?topic=/ com.ibm.debugtool.doc_7.1/ eqa7rm0293.htm.

Kernighan, B. W. and Ritchie, D. (1988). *The C Programming Language, Second Edition*. Prentice-Hall.

Kuipers, T. and Moonen, L. (2000). Types and concept analysis for legacy systems. In *IWPC*, pages 221–230. IEEE Computer Society.

Moonen, L. (2001). Generating robust parsers using island grammars. In *WCRE (Working Conference on Reverse Engeneering)*.

Moonen, L. (2003). Exploring software systems. In *ICSM*, pages 276–280. IEEE Computer Society.

Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.

van Deursen, A. and Moonen, L. (1998). Type inference for cobol systems. In *WCRE (Working Conference on Reverse Engeneering)*, pages 220–230.

van Deursen, A. and Moonen, L. (1999). Understanding cobol systems using inferred types. In *IWPC*. IEEE Computer Society.

van Deursen, A. and Moonen, L. (2000). Exploring legacy systems using types. In *WCRE (Working Conference on Reverse Engeneering)*, pages 32–41.

van Deursen, A. and Moonen, L. (2001). An empirical study into cobol type inferencing. *Sci. Comput. Program.*, 40(2-3):189–211.

van Deursen, A. and Moonen, L. (2006). Documenting software systems using types. *Sci. Comput. Program.*, 60(2):205–220.