Dependent Types for Program Understanding

Raghavan Komondoor*, G. Ramalingam, Satish Chandra, and John Field

IBM Research

Abstract. Weakly-typed languages such as Cobol often force programmers to represent distinct data abstractions using the same low-level physical type. In this paper, we describe a technique to recover implicitly-defined data abstractions from programs using type inference. We present a novel system of dependent types which we call *guarded types*, a path-sensitive algorithm for inferring guarded types for Cobol programs, and a semantic characterization of correct guarded typings. The results of our inference technique can be used to enhance program understanding for legacy applications, and to enable a number of type-based program transformations.

1 Introduction

Despite myriad advances in programming languages, libraries, and tools since business computing became widespread in the 1950s, large-scale legacy applications written in Cobol still constitute the computing backbone of many businesses. Such applications are notoriously difficult and time-consuming to update in response to changing business requirements. This difficulty very often stems from the fact that the logical structure of the code and data manipulated by these applications is not apparent from the program text. Two sources for this phenomenon are the lack in Cobol of modern abstraction mechanisms, and the fragmentation of the physical realization of logical abstractions due to repeated ad-hoc maintenance activities. In this paper, we focus on the problem of *recovering* certain data abstractions from legacy Cobol applications. By doing so, we aim to facilitate a variety of program maintenance activities that can benefit from a better understanding of logical data relationships.

Cobol is a *weakly-typed* language both in the sense that it has few modern type abstraction constructs[1], and because those types that it does have are for the most part not statically (or dynamically) enforced. For example:

- Cobol has no notion of scalar user-defined type; programmers can declare only the representation type of scalar variables (such variables are usually

* Contact author: komondoo@us.ibm.com.

[1] Modern versions of Cobol address some of these shortcomings; however, the bulk of existing legacy programs are written in early dialects of Cobol lacking type abstraction facilities.

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 157–173, 2005.

character or digit sequences). Hence, there is no means to declaratively distinguish among variables that store data from distinct logical domains, e.g., quantities and serial numbers.
- Cobol allows allows multiple record-structured variables to be declared to occupy the same memory. This "redefinition" feature can be used both to create different "views" on the same runtime variable, or to store data from different logical domains at different times, often distinguished by a tag value stored elsewhere. However, there is no explicit mechanism to declare which idiom is actually intended.
- Cobol programmers routinely store values in variables whose declared structures do not fully reflect the logical structure of the values being stored. One reason why programmers do this is the one already mentioned: to simulate subtyping by storing data from different logical domains (that are subtypes of some base domain) in a variable at different times.

As part of the *Mastery* project at IBM Research our long-term goal is to recover logical data models from applications at a level of abstraction similar to that found in expressive design-level languages such as UML [8] or Alloy [5], to alleviate language limitations, and to address the physical fragmentation alluded to above. Here, we describe initial steps toward this goal by describing a *type inference* technique for recovering abstractions from Cobol programs in the form of *guarded types*. Guarded types may contain any of the following classes of elements:

Atomic types: Domains of scalar values. In many cases, distinct atomic types will share the same physical representation; e.g., Quantity and SerialNumber. Atomic types can optionally be constrained to contain only certain specific runtime values.

Records: Domains consisting of fixed-length sequences of elements from other domains.

Guarded disjoint unions: Domains formed by the union of two or more logically disjoint domains, where the constituent domains are distinguished by one or more atomic types constrained to contain distinct *guard* or tag values.

The principal contributions of the paper are the guarded type system used to represent data abstractions; a formal characterization of a correct guarded typing of a program; and a path-sensitive algorithm to infer a valid guarded typing for any program (path-sensitivity is crucial to inferring reasonably accurate guarded union types). Although our techniques are designed primarily to address data abstraction recovery for Cobol programs, we believe our approach may also be applicable to other weakly-typed languages; e.g., assembly languages.

1.1 Introduction to MiniCobol and Motivating Example

We will illustrate our typing language and inference algorithm using the example programs in Fig. 1. These examples are written in a simple language MiniCobol, which contains the essential features of Cobol relevant to this paper. Consider the

```
    01 PAY-REC.
      05 PAYEE-TYPE PIC X.
      05 DATA PIC X(13).
    01 IS-VISITOR PIC X.
    01 PAY PIC X(4).
/1/ READ PAY-REC FROM IN-F.              ['E':Emp ⊗ EId ⊗ Salary ⊗ Unused ⊕
                                          !{'E'}:Vis ⊗ SSN5 ⊗ SSN4 ⊗ Stipend]
/2/ MOVE 'N' TO IS-VISITOR.              ['N':VisNo]
/3/ IF PAYEE-TYPE = 'E'                  ['E':Emp ⊕ !{'E'}:Vis]
/4/   MOVE DATA[8:11] TO PAY.            [Salary]
    ELSE
/5/   MOVE 'Y' TO IS-VISITOR.            ['Y':VisYes]
/6/   MOVE DATA[10:13] TO PAY.           [Stipend]
    ENDIF
/7/ WRITE PAY TO PAY-F.                  [Salary ⊕ Stipend]
/8/ IF IS-VISITOR = 'Y'                  ['N':VisNo ⊕ 'Y':VisYes]
/9/   WRITE DATA[6:9] TO VIS-F.          [SSN4]
                                         (a)
```

```
    01 ID.
      05 ID-TYPE PIC X(3).
      05 ID-DATA PIC X(9).
      05 SSN PIC X(9) REDEFINES ID-DATA.
      05 EMP-ID PIC X(7) REDEFINES ID-DATA.
/1/ READ ID.                             [ 'SSN':SSNTyp ⊗ SSN ⊕
                                           !{'SSN'}:EIdTyp ⊗ EId ⊗ Unused]
/2/ IF ID-TYPE = 'SSN'                   ['SSN':SSNTyp ⊕ !{'SSN'}:EIdTyp]
/3/  WRITE SSN TO SSN-F                  [SSN]
    ELSE
/4/  WRITE EMP-ID TO EID-F.              [EId]
    ENDIF
                                         (b)
```

```
    01 SSN.
    01 SSN-EXPANDED REDEFINES SSN.
      05 FIRST-5-DIGITS X(5).
      05 LAST-4-DIGITS X(4).
/1/ READ SSN FROM IDS-F.                 [SSN5 ⊗ SSN4]
/2/ WRITE LAST-4-DIGITS.                 [SSN4]
                                         (c)
```

Fig. 1. Example programs with guarded typing solutions produced by the inference algorithm of Sec. 3

fragment depicted in Fig. 1(a). The code for the program is shown in TYPEWRITER font, while the type annotations inferred by our inference algorithm are shown within square brackets. The initial part of the program contains variable declarations. Variables are prefixed by *level numbers*; e.g., 01 or 05. A variable with level 01 can represent either a scalar or a record; it is a record if additional variables with higher level numbers follow it, and a scalar otherwise. A variable with level greater than 01 denotes a record or scalar field nested within a previously-declared variable (with lower level). Clauses of the form PIC X(n) denote the fact that the corresponding variable is a character string of length n (n defaults to 1 when not supplied). A REDEFINES clause after a variable declaration indicates that two variables refer to the same storage. For example, in the program fragment in Fig. 1(b), variables ID-DATA, SSN, and EMP-ID all occupy the same storage. Note that the variable declarations reveal the total memory size required by a program (19 bytes, in the case of the example in Figure 1(a)), as well as the beginning and ending offset within memory of each variable.

The code following the data declarations contains the executable statements. MiniCobol contains MOVE statements, which represent assignments, READ and WRITE statements, as well as the usual control-flow constructs such as statement

sequencing, conditional statements, loops, and go-to statements. During program execution the value of each variable is a string of 1-byte characters, as is each program constant and the contents of each file. (Cobol follows the same approach, for the most part, e.g., representing numbers as strings of decimal digits). In other words, the program state at any point during execution of a program P is represented by a string of size |P| (in addition to the "program counter"), where |P| is the total memory required by P. A program P's execution begins with an implicit `READ` of |P| characters which initializes the state of the program.

`MOVE` statements have operands of equal length. The statement `READ` *var* `FROM` *file* reads |*var*| bytes from *file*, where |*var*| is the declared length of *var*, and assigns this value to *var* (we assume in this paper that programs are always given inputs that are "long enough", so `READ` *var* `FROM` *file* always gets |*var*| bytes). Similarly, `WRITE` *var* `TO` *file* appends the contents of *var* to *file*. In MiniCobol a *data reference* is a reference to a variable, or to a part of a variable identified by an explicit range of locations within the variable; e.g., `DATA[8:11]` refers to bytes 8 through 11 in the 13 byte variable `DATA`. We will use the term *variable occurrence* to denote an occurrence of a data-reference in a program.

The program in Fig. 1(a) reads a payment record from file `IN-F` and processes it. A payment record may pertain to an employee (`PAYEE-TYPE` = `'E'`), or to a visitor (`PAYEE-TYPE` ≠ `'E'`). For an employee, the first 7 bytes of `DATA` contain the employee ID number, the next four bytes contain the salary, and the last two bytes are unused. For a visitor, however, the first 9 bytes of `DATA` contain a social security number, and the next four bytes contain a stipend. The program checks the type of the payment record and copies the salary/stipend into `PAY` accordingly; it writes out `PAY` to file `PAY-F` and, in the case of a visitor, writes the last four digits of the social security number to `VIS-F`.

1.2 Inferring Guarded Types

The right column of Fig. 1 depicts the guarded typing solutions inferred by the algorithm in Sec. 3. For each line, the type shown between square brackets is the type assigned to the underlined variable at the program point *after* the execution of the corresponding statement or predicate. Guarded types are built from an expression language consisting of *(constrained) atomic types* and the operators '⊗' (concatenation) and '⊕' (disjoint union), with '⊗' binding tighter than '⊕'. Constrained atomic types are represented by expressions of the form *constr* : tvar, where *constr* is a *value constraint* and tvar is a *type variable*. A value constraint is either a literal value (in MiniCobol, always a string literal), an expression of the form !(*some set of literals*) denoting the set of all values *except* those enumerated in the set, or an expression of the form !{} denoting the set of all values. If the value constraint is omitted, then it is assumed to be !{}. The atomic type variables in the example are shown in sans serif font; e.g., Emp, Eld, Salary, and Unused. Our type inference algorithm does not generate meaningful names for type variables (the names were supplied manually for expository purposes); however, heuristics could be used to suggest names automatically based on related variable names. The inference process assigns a type to each *occur-*

rence of a data reference; thus different occurrences in the program of the same data reference may be assigned different types. By inspecting the guarded types assigned to data references in Fig. 1, we can observe that the inference process recovers data abstractions *not evident from declared physical types*, as follows:

Domain Distinctions. The typing distinguishes among distinct logical domains not explicitly declared in the program. For example, the references to `DATA[8:11]` in statement 4 and `DATA[6:9]` in statement 9 are assigned distinct type variables Salary and SSN4, respectively, although the declaration of `DATA` makes no such distinction.

Occurrence Typing and Value Flow. Different *occurrences* of variable `PAY` have distinct types, specifically, type Salary at statement 4, Stipend at statement 6, and Salary $\oplus$ Stipend at statement 7. This indicates that there is no "value flow" between statements 4 and 6, whereas there is potential flow between statements 4 and 7 as well as statements 6 and 7.

Scalar Values vs. Records. The typing solution distinguishes scalar types from record types; these types sometimes differ from physical structure of the declared variable. For example, `PAY-REC` at statement 1 has a type containing the concatenation operator '$\otimes$', which means it (and `DATA` within it) store structured data at runtime, while other variables in the program store only scalars. Note that although `DATA` is declared to be a scalar variable, it really stores record-structured data (whose "fields" are accessed via explicit indices). Note that an occurrence type can contain information about record structure that is inferred from definitions or uses elsewhere in the program of the value(s) contained in the occurrence, including program points following the occurrence in question. So, for example, the record structure of the occurrence of `PAY-REC` is inferred from uses of (variables declared within) `PAY-REC` in subsequent statements.

Value Constraints and Disjoint Union Tags. The constraints for the atomic types inside the union type associated with `IS-VISITOR` in statement 8 indicate that the variable contains either `'N'` or `'Y'` (and no other value). More interestingly, constrained atomic types inside records can be interpreted as *tags* for the disjoint unions containing them. For example, consider the type assigned to `PAY-REC` in statement 1. That type denotes the fact that `PAY-REC` contains *either* an employee number (EId) followed by a Salary and two bytes of of unused space, where the `PAYEE-TYPE` field is constrained to have value `'E'`, *or* a social security number followed by a stipend, with with the `PAYEE-TYPE` field constrained to contain `'E'`.

Overlay Idioms. Finally, we observe that the typing allows distinct data abstraction patterns, both of which use the `REDEFINES` overlay mechanism, to be distinguished by the inference process. Consider the example programs in Figures 1(b) and (c). Program (b) reads an `ID` record, and, depending on the value of the `ID-TYPE` field, interprets `ID-DATA` either as as a social security number or as an employee ID. Here, `REDEFINES` is used to store elements of a standard disjoint union type, and the type ascribed to `ID` makes this clear. By contrast,

example (c) uses the overlay mechanism to to provide two *views* of the same social security number data: a "whole" view, and a 2-part (first 5 digits, last 4 digits) view.

1.3 Applications

In addition to facilitating program understanding, data abstraction recovery can also be used to facilitate certain common program transformations. For example, consider a scenario where employee IDs in example Fig. 1(a) must be expanded to accommodate an additional digit. Such *field expansion* scenarios are quite common. The guarded typing solution we infer helps identify variable occurrences that are affected by a potential expansion. For example, if we wish to expand the implicit "field" of `DATA` containing Eld, only those statements that have references to Eid or other type variables in the same union component as Eid (e.g., Salary) are affected. Note that the disjoint union information inferred by our technique identifies a smaller set of affected items than previous techniques (e.g., [7]) which do not infer this information.

A number of additional program maintenance and transformation tasks can be facilitated by guarded type inference, although details are beyond the scope of this paper. Such tasks include: separating code fragments into modules based on which fragments use which types (which is a notion of *cohesion*); porting from weakly-typed languages to object-oriented languages; refactoring data declarations to make them reflect better how the variables are used (e.g., the overlaid variables `SSN` and `SSN-EXPANDED` in the example in Fig. 1(c) may be collapsed into a single variable); and migrating persistent data access from flat files to relational databases.

1.4 Related Work

While previous work on recovering type abstractions from programs [6, 3, 10, 7] has addressed the problem of inferring atomic and record types, our technique adds the capability of inferring disjoint union types, with constrained atomic types serving as tags. To do this accurately, we use a novel *path sensitive* analysis technique, where value constraints distinguish abstract dataflow facts that are specific to distinct paths. Since the algorithm is flow-sensitive, it also allows distinct occurrences of the same variable to be assigned different types. To see the strengths of our approach, consider again the example in Fig. 1(a). The algorithm uses the predicate `IF PAYEE-TYPE = 'E'` to split the dataflow fact corresponding to `PAY-REC` into two facts, one for the "employee" case (`PAYEE-TYPE = 'E'`) and the other for the "visitor" case (`PAYEE-TYPE` $\neq$ `'E'`). As a result, the algorithm infers that `DATA[8:11]` (at one occurrence) stores a Salary while the `DATA[10:13]` stores a Stipend (at a different occurrence) even though these two memory intervals are overlapping. We are aware of no prior abstraction inference technique that is capable of making this distinction. Note that our approach can in many cases maintain correlations between values of variables, and hence correlate fragments of code that are not even controlled by predicates that have common variables. For example, our approach recognizes that statements 5 and

9 in Fig. 1(a) pertain to the "visitor" case, even though the controlling predicates for each statement do not share a common variable.

The flow-insensitive approach of [10] is able to infer certain subtyping relationships; these are similar in some respects to our union types. In particular, when a single variable is the target of assignments from different variables at different points, e.g., the variable `PAY` in statements 4 and 6 in Fig. 1(a), their approach infers that the types of the source variables are subtypes of the type of the target. Our approach yields similar information in this case. However, our technique uses path sensitivity to effectively identify subtyping relationships in additional cases; e.g., a supertype (in the form of a disjoint union) is inferred for `PAY-REC` in statement 1, even though this variable is explicitly assigned only once in the program.

Various approaches based on analysis techniques other than static type inference, e.g., concept analysis, dynamic analysis, and structural heuristics, have been proposed for the purpose of extracting logical data models (or aspects of logical data models) from existing code [1, 2, 4, 9]. Previous work in this area has not, to the best of our knowledge, addressed extraction of type abstractions analogous to our guarded types (in particular, extraction of union/tag information). However, much of this work is complementary in the sense that it recovers different classes of information (invariants, clusters, roles, etc.) that could be profitably combined with our types.

Our guarded types are *dependent types*, in the sense that they incorporate a notion of value constraint. While dependent types have been applied to a number of problems (see [11] for examples), we are unaware of any work that has used dependent types to recover data abstractions from legacy applications, or that combine structural inference with value flow information.

The rest of the paper is structured as follows. Section 2 specifies the guarded type language and notation. Section 3 presents our type inference algorithm. Following that, we present the correctness characterization for the guarded type system in Section 4, along with certain theorems concerning correct type solutions. We conclude the paper in Section 5 with a discussion on future work.

2 The Type System

Let $AtomicTypeVar = \cup_{i>0} V_i$ denote a set of type variables. A type variable belonging to V_i is said to have *length* i. We will use symbols α, β, γ, etc., (sometimes in subscripted form, as in α_i) to range over type variables. Type variables are also called atomic types.

As the earlier examples illustrated, often the specific value of certain *tag* variables indicate the type of certain other variables. To handle such idioms well, types in our type systems can capture information about the values of variables. We define a set of value constraints *ValueAbs* as follows, and use symbols c, d, c_1, d_2, etc., to range over elements of *ValueAbs*:

$$ValueAbs ::= s \mid !\{s_1, s_2, \ldots, s_k\}, \text{ where } s \text{ and each } s_i \text{ are } Strings$$

While the value constraint s is used to represent that a variable has the value s, the value constraint $!\{s_1, s_2, \ldots, s_k\}$ is used to represent that a variable has a value different from s_1 through s_k. In particular, the value constraint $!\{\}$ represents any possible value, and we will use the symbol $\top$ to refer to $!\{\}$.

We define a set of type expressions $\mathcal{TE}$, built out of type variables, and value constraints using *concatenation* and *union* operators, as follows:

$$\mathcal{TE} ::= (\mathit{ValueAbs}, \mathit{AtomicTypeVar}) \mid \mathcal{TE} \otimes \mathcal{TE} \mid \mathcal{TE} \oplus \mathcal{TE}$$

We refer to a type expression of the form $(\mathit{ValueAbs}, \mathit{AtomicTypeVar})$ as a *leaf* type-expression. We refer to a type expression containing no occurrences of the union operator '$\oplus$' as a *union-free* type expression.

We will use the notation $\alpha^{|i|}$ to indicate that variable α has length i, and the notation $c{:}\alpha^{|i|}$ to represent a leaf type-expression $(c, \alpha^{|i|})$. In contexts where there is no necessity to show the *ValueAbs* component we use the notation $\alpha^{|i|}$ to denote a leaf type-expression itself. Where there is no confusion we denote concatenation implicitly (without the $\otimes$ operator).

A *type mapping* for a given program is a function from variable occurrences in the program, denoted *VarOccurs*, to $\mathcal{TE}$.

3 Type Inference Algorithm

3.1 Introduction to Algorithm

Input: The input to our algorithm is a control flow graph, generated from the program and preprocessed as follows. All complex predicates (involving logical operators) are decomposed into simple predicates and appropriate control flow. Furthermore, predicates P of the form "`X == s`" or "`X != s`", where `s` is a constant string, are converted into a statement "`Assume` P" in the *true* branch and a statement "`Assume` $!P$" in the *false* branch. Other simple predicates are handled conservatively by converting them into no-op statements that contain references to the variables that occur in the predicate. The program has a single (structured) variable `Mem` (if necessary, a new variable is introduced that contains all of the program's variables as substructures or fields). We assume, without loss of generality, that a program has a single input file and a single output file.

Solution Computed by the Algorithm: For every statement `S`, the algorithm computes a set `S.inType` of union-free types (see Section 2), which describes the type of variable `Mem` before statement `S`. Specifically, the set $\{f_1, f_2, \cdots, f_k\}$, where each f_i is a union-free type, is the representation used by the algorithm for the type $f_1 \oplus f_2 \oplus \cdots \oplus f_k$. The algorithm represents each union-free type in right-associative normal form (i.e., as a sequence of leaf type-expressions). When the algorithm is finished each `S.inType` set contains the type of the variable `Mem` at the program point before statement `S`. Generating a type mapping for all variables from this is straightforward, and is based on the following characteristic of the computed solution: for each variable `X` that occurs in `S` and each union-free type f in `S.inType`, f contains a projection $f[X]$ (i.e., a subsequence of f,

which itself is a sequence of leaf type-expressions) that begins (resp. ends) at the same offset position as X begins (resp. ends) within `Mem`. We omit the details of generating the type mapping due to space constraints.

Key Aspects of the Algorithm: We now describe the essential conceptual structure of our inference algorithm. The actual algorithm, which is presented in Figures 2 and 3, incorporates certain optimizations and, hence, has a somewhat different structure. Recall that `READ`s and literal `MOVE`s (`MOVE` statements whose source operand is a constant string) are the only "origin" statements: i.e., these are the only statements that introduce new values during execution (other statements use values, or copy them, or write them to files). For each origin statement `S`, our algorithm maintains a set `S.readType` of union-free types, which represents the type of the values originating at this statement.

At the heart of our algorithm is an iterative, worklist-based, dataflow analysis that, given `S.readType` for every origin statement `S`, computes `S1.inType` for every statement `S1` in the program. An element $\langle$`S`, $f\rangle$ in the worklist indicates that f belongs to `S.inType`. The analysis identifies how the execution of `S` transforms the type f into a type f' and propagates f' to the successors of `S`. We will refer to this analysis as the *inner loop analysis*.

The whole algorithm consists of an *outer loop* that infers `S.readType` (for every origin statement `S`) in an iterative fashion. Initially, the values originating at an origin statement `S` are represented by a single type variable $\alpha_{\mathtt{S}}$ whose length is the same as that of the operand of `S`. In each iteration of the outer loop analysis, an inner loop analysis is used to identify how the values originating at statement `S` (described by the set `S.readType`) flow through the program. During this inner loop analysis, two situations (described below) identify a *refinement* to `S.readType`. When this happens, the inner loop analysis is (effectively) stopped, `S.readType` is refined as necessary, and the next iteration of the outer loop is started. The algorithm terminates when an instance of the inner loop analysis completes without identifying any further refinement to `S.readType`.

We now describe the two possible ways in which `S.readType` may be refined. The first type of refinement happens when the inner loop analysis identifies that there is a reference in a statement `S2` to a *part* of a value currently represented by a type variable β. When this happens, the algorithm *splits* β into new variables of smaller lengths such that the portion referred to in `S2` corresponds exactly to one of the newly obtained variables. More specifically, let `S` be the origin statement for β (i.e., `S.readType` includes some union-free type that includes β). Then, `S.readType` is refined by replacing β by a sequence $\beta_1\beta_2$ or a sequence $\beta_1\beta_2\beta_3$ as appropriate. The intuition behind splitting β is that the reference to the portion of β in `S2` is an indication that β is really *not* an atomic type, but a structured type (that contains the β_i's as fields).

The second type of refinement happens when the inner loop analysis identifies that a value represented by a leaf type, say γ, may be compared for equality with a constant l. When this happens, the leaf type is *specialized* for constant l. Specifically, if the leaf type originates as part of a union-free type, say $\gamma\delta\rho$, in `S.readType`, then $\gamma\delta\rho$ is replaced by two union-free types $(l:\gamma_1)\delta_1\rho_1$ and

Procedure `Main`
  Initialize worklist to { $\langle entry, \top{:}\alpha^{|m|}\rangle$}, where *entry* is the entry statement of the program, α is a new type variable, and m is the size of memory. Initialize `S.inType` to ϕ for all statements `S`.
  for all statements `S` = `READ Y` **do** {`X` and `Y` are used to denote *variable occurrences* (see Sec. 1.1)}
    Create a new type variable $\alpha_{\mathtt{S}}^{|l|}$, where l is the size of `Y`. Initialize `S.readType` to $\{\top{:}\alpha_{\mathtt{S}}\}$.
  for all statements `S` = `MOVE` s `TO Y`, where s is a string literal **do**
    Create a new type variable $\alpha_{\mathtt{S}}^{|l|}$, where l is the length of s (and of `Y`). From this point in the algorithm treat `S` as if it were the statement "`READ Y`". Initialize `S.readType` to $\{s{:}\alpha_{\mathtt{S}}\}$.
  while worklist is not empty **do**
    Extract some $\langle$`S, t`$\rangle$ from worklist. Call `Process(S, t)`.
Procedure `Process(S` : statement, `ft` : union-free type for `Mem)`
  for all variable occurrences `X` in `S` **do**
    if `Subseq(ft, X)` is undefined **then**
      Call `Split(ft, X)`. Call `Restart`. **return**.
  if `S = MOVE X TO Y` **then**
    Call `Propagate(Succ, Subst(ft, Y, Subseq(ft,X)))`, for all successors `Succ` of `S`.
  else if `S = READ Y` **then**
    for all union-free types `ftY` in `S.readType` **do**
      Call `Propagate(Succ, Subst(ft, Y, ftY))`, for all successors `Succ` of `S`.
  else if `S = ASSUME X ==` s **then**
    Let `ret` = `evalEquals(Subseq(ft,X)`, s).
    if `ret` = *true* **then**
      Call `Propagate(Succ, ft)`, for all successors `Succ` of `S`.
    else if `ret` = *false* **then**
      do nothing {`Subseq(ft,X)` is *inconsistent* with s – hence no fact is propagated}
    else {`ret` is of the form (α, s_i)}
      Call `Specialize`(α, s_i). Call `Restart`. **return**.
  else if `S = ASSUME X !=` s **then**
    Let `ret` = `evalNotEquals(Subseq(ft, X)`, s).
    if `ret` = *true* **then**
      Call `Propagate(Succ, ft)`, for all successors `Succ` of `S`.
    else {`ret` = *false*}
      do nothing {`Subseq(ft,X)` has the constant value s – hence no fact is propagated}
    else {`ret` is of the form (α, s_i)}
      Call `Specialize`(α, s_i). Call `Restart`. **return**.
Function `Subseq(ft` : union-free type for `Mem, X` : (portion of) program variable)
  if a sequence `ftX` of leaf type-expressions within `ft` begins (ends) at the same position within `ft` as `X` does within `Mem` **then return** `ftX` **else** *Undefined*
Function `Subst(ft` : union-free type for `Mem, X` : (portion of) program variable, `ftX` : union-free type) {|`ft`| = |`Mem`|, |`ftX`| = |`X`|, and `Subseq(ft, X)` is defined. }
  Replace the subsequence `Subseq(ft, X)` within `ft` with `ftX` and return the resultant union-free type.
Procedure `Propagate(S` : statement, `ft` : union-free type for `Mem)`
  if $\langle$`S, ft`$\rangle \notin$ `S.inType` **then** Add $\langle$`S, ft`$\rangle$ to worklist, and to `S.inType`.
Procedure `Restart`
  for all `READ` statements `S` **do**
    for all union-free types $\mathtt{ft}_p$ in `S.inType` **do**
      add $\langle$`S`, $\mathtt{ft}_p\rangle$ to the worklist

Fig. 2. Type inference algorithm – procedures `Main,Process, Subseq, Subst, Propagate,` and `Restart`

$(!l : \gamma_2)\delta_2\rho_2$ (consisting of new type variables) in `S.readType`. In the general case, repeated specializations can produce more complex value constraints (see Figures 2 and 3 for a complete description of specialization). The benefit of specializing a type by introducing copies is that variable occurrences in the *then* and *else* branches of `IF` statements cause the respective copies of the type to refined, thus improving precision.

The algorithm infers a type mapping for every program. It always terminates, intuitively because the inner-loop analysis is monotonous, and because the mem-

Procedure `Split`(`ft` : union-free type for `Mem`, `X` : (portion of) program variable)

Let `a` $= \alpha^{|l|}$ be a leaf type-expr within `ft` and `off` be an offset within `a` such that the prefix or suffix of `a` bordering `off` occupies an interval within `ft` that is non-overlapping with the interval occupied by `X` within `Mem`. Create two new type variables $\alpha_1^{|l_1|}$ and $\alpha_2^{|l_2|}$, where l_1 = `off` and $l_2 = l -$ `off`. Let `S` be the `READ` statement such that there exists a union-free type $\texttt{ft}_\texttt{S} \in$ `S.readType` such that a leaf type-expr $\texttt{b} = c{:}\alpha^{|l|}$ is in $\texttt{ft}_\texttt{S}$.
if c is a string s **then**
Split s in to two strings s_1 and s_2 of lengths l_1 and l_2, respectively.
Let $\texttt{b}_{split} = s_1{:}\alpha_1^{|l_1|} s_2{:}\alpha_2^{|l_2|}$.
else {c is of the form $!some\ set$}
Let $\texttt{b}_{split} = \top{:}\alpha_1^{|l_1|} \top{:}\alpha_2^{|l_2|}$.
Create a copy $\texttt{ft}'_\texttt{S}$ of $\texttt{ft}_\texttt{S}$ that is identical to $\texttt{ft}_\texttt{S}$ except that `b` is replaced by $\texttt{b}_{split}$.
Call `Replace`(`S`, $\texttt{ft}_\texttt{S}$, $\{\texttt{ft}'_\texttt{S}\}$).

Procedure `Specialize`($\alpha^{|l|}$: type variable, s : string of length l)

Let `S` be the `READ` statement such that there exists a union-free type $\texttt{ft}_\texttt{S} \in$ `S.readType` such that a leaf type-expr $\texttt{b} = c{:}\alpha$ is in $\texttt{ft}_\texttt{S}$. Pre-condition: c is of the form `!Q`, where `Q` is a set that does not contain s. Create two new copies of $\texttt{ft}_\texttt{S}$, $\texttt{ft1}_\texttt{S}$ and $\texttt{ft2}_\texttt{S}$, such that each one is identical to $\texttt{ft}_\texttt{S}$ except that it uses new type variable names. Replace the leaf type-expr corresponding to `b` in $\texttt{ft1}_\texttt{S}$ with $s{:}\alpha_1^{|l|}$, and the leaf type-expr corresponding to `b` in $\texttt{ft2}_\texttt{S}$ with $!(\texttt{Q} + \{s\}){:}\alpha_2^{|l|}$, where α_1 and α_2 are two new type variables. Call `Replace`(`S`, $\texttt{ft}_\texttt{S}$, $\{\texttt{ft1}_\texttt{S}, \texttt{ft2}_\texttt{S}\}$).

Procedure `Replace`(`S` : a `READ` statement, `ft` : a union-free type in `S.readType`, `fts` : a set of union-free types)

Set `S.readType` = `S.readType` − {`ft`} + `fts`.
for all all type variables α occurring in `ft` **do**
Remove from `S.inType`, for all statements `S`, all union-free types that contain α. Remove from the worklist all facts ⟨`Sa`, `fta`⟩, where `fta` is a union-free type that contains α.

Procedure `evalEquals`(`ft` : union-free type, `s` : string of the same length as `ft`)

Say $\texttt{ft} = c_1{:}\alpha_1^{|l_1|} c_2{:}\alpha_2^{|l_2|} \ldots c_m{:}\alpha_m^{|l_m|}$.
Let $s_1, s_2, \ldots s_m$ be strings such that $s = s_1 s_2 \ldots s_m$ and s_i has length l_i, for all $1 \leq i \leq m$.
if for all $1 \leq i \leq m$: $c_i = s_i$ **then**
return *true* {`ft`'s value is s}
else if for some $1 \leq i \leq m$: $c_i =$`!S`, where `S` is a set that contains s_i **then**
return *false* {`ft` is inconsistent with s}
else {`ft` is consistent with s – therefore, specialize `ft`}
Let i be an integer such that $1 \leq i \leq m$ and c_i is `!S`, where `S` is a set that does *not* contain s_i.
return (α_i, s_i)

Procedure `evalNotEquals`(`ft` : union-free type, `s` : string of the same length as `ft`)

Say $\texttt{ft} = c_1{:}\alpha_1^{|l_1|} c_2{:}\alpha_2^{|l_2|} \ldots c_m{:}\alpha_m^{|l_m|}$.
Let $s_1, s_2, \ldots s_m$ be strings such that $s = s_1 s_2 \ldots s_m$ and s_i has length l_i, for all $1 \leq i \leq m$.
if for all $1 \leq i \leq m$: $c_i = s_i$ **then**
return *false* {`ft`'s value is equal to s}
else
if $m > 1$ *OR* $m = 1$ and $c_1 =$!(*some set containing* s_1) **then return** *true* **else return** (α_1, s_1).

Fig. 3. Type inference algorithm – other procedures

ory requirement (and hence, the number of refinement steps) for any program is fixed. The actual algorithm described in Figures 2 and 3 differs from the above conceptual description as follows: Rather than perform an inner loop analysis from scratch in each iteration of the outer loop, results from the previous execution of the inner loop analysis that are still valid are reused. Therefore, the two loops are merged into a single loop.

3.2 Illustration of Algorithm Using Example in Figure 1(a)

Figure 4 illustrates a trace of the algorithm when applied to the example in Figure 1(a). Specifically, the figure illustrates (a subset of) the state of the

Time	Statement S	S.inType	S.readType
t_1:	/1/ READ PAY-REC FROM IN-F.	{Initial$^{\|19\|}$}	{PayRec$^{\|14\|}$}
t_2:	/1/ READ PAY-REC FROM IN-F.	{Init$_1^{\|14\|}$Init$_2^{\|5\|}$}	{PayRec$^{\|14\|}$}
	/2/ MOVE 'N' TO IS-VISITOR.	{PayRec$^{\|14\|}$Init$_2^{\|5\|}$}	{'N':VisNo$^{\|1\|}$}
t_3:	/1/ READ PAY-REC FROM IN-F.	{Init$_1^{\|14\|}$Init$_3^{\|1\|}$Init$_4^{\|4\|}$}	{PayRec$^{\|14\|}$}
	/2/ MOVE 'N' TO IS-VISITOR.	{PayRec$^{\|14\|}$Init$_3^{\|1\|}$Init$_4^{\|4\|}$}	{'N':VisNo$^{\|1\|}$}
	/3/ IF PAYEE-TYPE = 'E'	{PayRec$^{\|14\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$}	
t_4:	/1/ READ PAY-REC FROM IN-F.	{Init$_1^{\|14\|}$Init$_3^{\|1\|}$Init$_4^{\|4\|}$}	{'E':Emp$^{\|1\|}$PayRec$_3^{\|13\|}$, !{'E'}:Vis$^{\|1\|}$PayRec$_4^{\|13\|}$}
	/2/ MOVE 'N' TO IS-VISITOR.		{'N':VisNo$^{\|1\|}$}
	/3/ IF PAYEE-TYPE = 'E'		
t_5:	/1/ READ PAY-REC FROM IN-F.	{Init$_1^{\|14\|}$Init$_3^{\|1\|}$Init$_4^{\|4\|}$}	{'E':Emp$^{\|1\|}$PayRec$_3^{\|13\|}$, !{'E'}:Vis$^{\|1\|}$PayRec$_4^{\|13\|}$}
	/2/ MOVE 'N' TO IS-VISITOR.	{'E':Emp$^{\|1\|}$PayRec$_3^{\|13\|}$Init$_3^{\|1\|}$Init$_4^{\|4\|}$, !{'E'}:Vis$^{\|1\|}$PayRec$_4^{\|13\|}$Init$_3^{\|1\|}$Init$_4^{\|4\|}$}	{'N':VisNo$^{\|1\|}$}
	/3/ IF PAYEE-TYPE = 'E'	{'E':Emp$^{\|1\|}$PayRec$_3^{\|13\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$, !{'E'}:Vis$^{\|1\|}$PayRec$_4^{\|13\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$}	
t_6:	/3/ IF PAYEE-TYPE = 'E'	{'E':Emp$^{\|1\|}$PayRec$_3^{\|13\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$, !{'E'}:Vis$^{\|1\|}$PayRec$_4^{\|13\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$}	
	/4/ MOVE DATA[8:11] TO PAY. ELSE	{'E':Emp$^{\|1\|}$PayRec$_3^{\|13\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$}	
	/5/ MOVE 'Y' TO IS-VISITOR.	{!{'E'}:Vis$^{\|1\|}$PayRec$_4^{\|13\|}$'N':VisNo$^{\|1\|}$Init$_4^{\|4\|}$}	{'Y':VisYes$^{\|1\|}$}

Fig. 4. Illustration of algorithm using example in Figure 1(a)

algorithm at selected seven points in time $(t_1, t_2, \ldots, t_7)$. The second column in the figure shows a statement S, the third column shows the value of S.inType, while the last column shows the value of S.readType if S is an origin statement.

Initially, a type variable is created for each origin statement. As explained in Section 1.1, a MiniCobol program has an implicit READ Mem at the beginning. Though we do not show this statement in Figure 4, it is an origin statement, with a corresponding type variable Initial$^{|19|}$, representing the initial state of memory, in its readType. In the figure /1/.inType represents the readType of the implicit READ. Similarly, /1/.readType contains PayRec$^{|14|}$, which is the initial type assigned by the algorithm to PAY-REC. (We use the notation /n/ to denote the statement labeled n in Figure 1(a).)

The first row shows the state at time point t_1, when the worklist contains the pair $\langle$/1/, Initial$^{|19|}\rangle$. Notice that statement 1 (READ PAY-REC) has a variable occurrence (PAY-REC) that corresponds to a portion (the first 14 bytes) of Initial$^{|19|}$, which is the type variable for the entire memory. Therefore, as described in Section 3.1, Initial$^{|19|}$ is "split" into Init$_1^{|14|}$Init$_2^{|5|}$. This split refinement updates the readType associated with the implicit initialization READ M and terminates the first inner loop analysis and initiates the second inner loop analysis.

In the next inner loop analysis, $\langle$/1/, Init$_1^{|14|}$Init$_2^{|5|}\rangle$ is placed in the worklist. Processing this pair requires no more splitting; therefore, Init$_1^{|14|}$ is replaced by PayRec$^{|14|}$, which is the type in /1/.readType. The resultant type f = PayRec$^{|14|}$Init$_2^{|5|}$ is placed in /2/.inType and is propagated to statement /2/ (by placing $\langle$/2/, $f\rangle$ in the worklist). The resulting algorithm state is shown in Figure 4 at time point t_2.

(In general, for any origin statement S that refers to a variable X, processing a pair $\langle$S, $f\rangle$ involves replacing the portion of f that corresponds to X ($f[X]$) with t_X, for each type t_X in S.readType, and propagating the resultant type(s) to the program point(s) that follow S.)

Next, the worklist item $\langle /2/, \mathsf{PayRec}^{|14|}\mathsf{Init}_2^{|5|}\rangle$ is processed. As statement /2/ refers to IS-VISITOR, which corresponds to a portion of $\mathsf{Init}_2^{|5|}$, this type variable is split into $\mathsf{Init}_3^{|1|}\mathsf{Init}_4^{|4|}$ and a new inner loop analysis is started.

This analysis propagates the newly split type through statements /1/ and /2/. The result is that the type $\mathsf{PayRec}^{|14|}$'N':$\mathsf{VisNo}^{|1|}\mathsf{Init}_4^{|4|}$ reaches /3/.inType. The resulting state is shown as time point t_3. Statement /3/ causes a split once again, meaning a new inner loop analysis starts.

The next inner loop analysis eventually reaches the state shown as time point t_4, where the algorithm is about to process the pair $\langle /3/,\ \mathsf{PayRec}_1^{|1|}\mathsf{PayRec}_2^{|13|}$'N':$\mathsf{VisNo}^{|1|}\mathsf{Init}_4^{|4|}\rangle$ from the worklist. Because PAYEE-TYPE, which is of type $\mathsf{PayRec}_1^{|1|}$, is compared with the constant 'E', the algorithm *specializes* the type variable $\mathsf{PayRec}_1^{|1|}$ by replacing, in its origin /1/.readType, its container type $\mathsf{PayRec}_1^{|1|}\mathsf{PayRec}_2^{|13|}$ with two types $\{$'E' : $\mathsf{Emp}^{|1|}\mathsf{PayRec}_3^{|13|}$, !{'E'} : $\mathsf{Vis}^{|1|}\mathsf{PayRec}_4^{|13|}\}$. A new inner loop analysis now starts.

Using the predicate PAYEE-TYPE = 'E' to specialize /1/.readType is meaningful for the following reason: since statement /1/ is the *origin* of $\mathsf{PayRec}_1^{|1|}$ (the type of PAYEE-TYPE), the predicate implies that there are two kinds of records that are read in statement /1/, those with the value 'E' in the their PAYEE-TYPE field and those with some other value, and that these two types of records are handled differently by the program. The specialization of /1/.readType captures this notion.

Time point t_5 shows the algorithm state after the updated /1/.readType is propagated to /3/.inType by the new inner loop analysis. Notice that corresponding to the two types in /1/.readType, there are two types in /2/.inType and /3/.inType (previously there was only one type in those sets). The types in /3/.inType are (as shown): f_1 = 'E':$\mathsf{Emp}^{|1|}\mathsf{PayRec}_3^{|13|}$'N':$\mathsf{VisNo}^{|1|}\mathsf{Init}_4^{|4|}$ and f_2 =!{'E'}:$\mathsf{Vis}^{|1|}\mathsf{PayRec}_4^{|13|}$'N':$\mathsf{VisNo}^{|1|}\mathsf{Init}_4^{|4|}\}$.

The same inner loop analysis continues. Since f_1 and f_2 are now specialized wrt PAYEE-TYPE, the algorithm determines that type f_1 need only be propagated to the *true* branch of the IF predicate and that type f_2 need only be propagated to the *false* branch. The result is shown in time point t_6. This is an exhibition of path sensitivity, and it has two benefits. Firstly, the variables occurring in each branch cause only the appropriate type (f_1 or f_2) to be split (i.e, the two branches do not pollute each other). Secondly, the correlation between the values of the variables PAYEE-TYPE and IS-VISITOR is maintained, which enables the algorithm, when it later processes the final IF statement (statement /8/), to propagate only the type that went through the *true* branch of the first IF statement (i.e., f_1) to the *true* branch of statement /8/.

We finish our illustration of the algorithm at this point. The final solution, after the computed inType sets are converted into a type mapping for all variable

occurrences is shown in Figure 1(a). Notice that each type in /1/.`readType` (shown to the right of Statement 1) reflects the structure inferred from only those variables that occur in the appropriate branch of the `IF` statements.

4 Type System: Semantics, Correctness, and Properties

In this section we define the notion of a "correct" type mapping, which we call a *typing solution*. We state certain properties of typing solutions, and illustrate that as a result typing solutions provide information about flow of values in the program. Note that a program may, in general, have a number of typing solutions; our type inference algorithm finds one of them.

4.1 An Instrumented Semantics for MiniCobol

Since we are interested in tracking the flow of values, we define an instrumented semantics where every input-file- and literal-character value is tagged with an unique integer that serves as its identifier. Let *IChar* denote the set of *instrumented characters*, and *IString* denote the set of instrumented strings (sequences of instrumented characters). Thus, every instrumented string *is* contains a character string, $charSeq(is)$, which is its actual value, as well as an *integer sequence*, $intSeq(is)$.

It is straightforward to define an instrumentation function that takes a program P and an input string I and returns *instr*(P,I) – an instrumented program and an instrumented string – by converting every character in every string literal occurring in P as well as every character in I into an instrumented character with a unique id. Thus, *instr*(P,I) contains a set of instrumented strings, one corresponding to I, and the others corresponding to the string literals in P.

We define a collecting instrumented semantics $\mathcal{M}$ with the following signature:

$$\mathcal{M} : Program \rightarrow String \rightarrow VarOccurs \rightarrow 2^{IString}$$

Given a program P and an input (*String*) I, the instrumented semantics executes the instrumented program and input *instr*(P,I) much like in the standard semantics, except that every location now stores an instrumented character, and the instrumented program state is represented by an instrumented string. The collecting semantics $\mathcal{M}$ identifies the set of all values (*IString*s) each variable occurrence in the program can take.

4.2 Semantics for Type Expressions

We can give type-expressions a meaning with the signature

$$\mathcal{T} : \mathcal{TE} \rightarrow (AtomicTypeVar \rightarrow 2^{IString}) \rightarrow 2^{IString}$$

as follows: this definition extends a given $\sigma : AtomicTypeVar \rightarrow 2^{IString}$ that maps a type variable to a set of values (instrumented strings) of the same length

as the type variable, to yield the set of values represented by a $\mathcal{TE}$. Before defining $\mathcal{T}$, we define the meaning of value constraints via a function $\mathcal{C}$ which maps *ValueAbs* to 2^{String}:

$$\begin{aligned}
\mathcal{C}(s) &= \{s\} \\
\mathcal{C}(!\{s_1, s_2, \ldots, s_k\}) &= \{s \mid s \in String \wedge s \notin \{s_1, s_2, \ldots, s_k\}\} \\
\mathcal{T}[c{:}\alpha]\sigma &= \{v \mid v \in \sigma(\alpha) \wedge charSeq(v) \in \mathcal{C}(c)\} \\
\mathcal{T}[\tau_1 \otimes \tau_2]\sigma &= \{i_1 @ i_2 \mid i_i \in \mathcal{T}[\tau_1]\sigma, i_2 \in \mathcal{T}[\tau_2]\sigma\} \\
&\quad (@ \text{ represents concatenation}) \\
\mathcal{T}[\tau_1 \oplus \tau_2]\sigma &= \mathcal{T}[\tau_1]\sigma \cup \mathcal{T}[\tau_2]\sigma
\end{aligned}$$

4.3 Correct Type Mappings

Definition 1 (Atomization). *An* atomization *of an instrumented string s is a list of instrumented strings whose concatenation yields s: i.e., a list $[s_1, s_2, \cdots, s_k]$ such that $s_1 @ s_2 @ \cdots @ s_k = s$. We refer to the elements of an instrumented string's atomization as* atoms.

Definition 2 (Atomic Type Mapping). *Given a program P and an input string I, an atomic type mapping π for (P,I) consists of an atomization of each instrumented string in instr(P,I), along with a function mapping every atom to a type variable. We denote the set of atoms produced by π by atoms(π), and denote the type variable assigned to an atom a by just $\pi(a)$. Also, π^{-1} is the inverse mapping, from type variables to sets of atoms, induced by π.*

Definition 3 (Correct Atomic Type Mapping). *Let Γ be a type mapping for a program P, and let π be an atomic type mapping for instr(P,I), where I is an input string. (Γ, π) is said to be correct for (P,I) if for every variable occurrence v in P,*

$$\mathcal{T}[\Gamma(v)]\pi^{-1} \supseteq \mathcal{M}[P](I)(v).$$

For example, consider the given program P and type mapping Γ_b in Figure 1(b), and let input string I = 'EID1234567'. In this case *instr*(P,I) contains two instrumented strings, '$\overline{\text{SSN}}$' from P and '$\overline{\text{EID1234567}}$' from I; we omit, for brevity, the (unique) integer tags on the characters, and use an overline to indicate their presence. A candidate atomization and atomic type mapping π_b for this example is ['$\overline{\text{SSN}}$': SSNTyp, '$\overline{\text{EID}}$':EIdTyp, '$\overline{1234567}$':EId]. (Γ_b, π_b) is correct for the given (P, I).

Definition 4 (Typing Solution). *A type mapping Γ for a program P is said to be* correct *if for every input I there exists an atomic type mapping π such that (Γ, π) is correct for (P,I). We will refer to a type mapping that is correct as a* typing solution.

Because π maps each atom in the input string and program to a single type variable, it follows that in a typing solution distinct type variables correspond to distinct domains of values (atoms).

4.4 Properties of Correct Type Mappings

Theorem 1 (Atoms are Indivisible). *If (Γ, π) is correct for (P,I), then during execution of P on I no variable occurrence ever contains a* part of an atom *without containing the whole atom.*

For example, recall the pair (Γ_b, π_b) mentioned earlier, and recall that it was correct for the program in Figure 1(b) with input string 'EID1234567'. Then, the above theorem asserts that no variable occurrence in the program ever takes on a value that contains a proper substring of any of the atoms '$\overline{\text{SSN}}$', '$\overline{\text{EID}}$', and '$\overline{1234567}$' during execution of the program on the given input string. Thus, an atomization helps identify indivisible units of "values" that can be meaningfully used to talk about the "flow of values". The indivisibility also implies that in a typing solution each type variable corresponds to a *scalar* domain.

We now show how typing solutions tell us whether, for any two variable occurrences, there is *no* execution in which some instrumented value flows to both occurrences. The following definition formalizes this notion of "disjointedness".

Definition 5 (Disjointedness). *Two variable occurrences v and w in a program P are said to be disjoint if for any input I, for any $s_1 \in \mathcal{M}[P](I)(v)$ and $s_2 \in \mathcal{M}[P](I)(w)$, s_1 and s_2 do not have any instrumented character in common.*

We now introduce the notion of *overlap*, and then show how typing solutions yield information about disjointness.

Definition 6 (Overlap). *(a) Two value constraints c_1 and c_2 are said to* overlap *if they are not of the form s_1 and s_2, where $s_1 \neq s_2$ and not of the form s_1 and $!S$, where $s_1 \in S$. (b) Two leaf type-expressions $c_1{:}\alpha_1$ and $c_2{:}\alpha_2$ are said to overlap if $\alpha_1 = \alpha_2$ and c_1 and c_2 overlap.*

Theorem 2 (Typing Solutions Indicate Disjointedness). *Let Γ be a typing solution for a program P and let v and w be two variable occurrences in P. If $\Gamma(v)$ and $\Gamma(w)$ have no overlapping leaf type-expressions, then v and w are strongly disjoint.*

Consider the example program and typing solution in Figure 1(a). The two occurrences of `PAY` in lines 4 and 6, respectively, have non-overlapping types (Salary and Stipend, respectively). Theorem 2 thus tells us that these two occurrences are disjoint (even though they refer to the same variable). On the other hand each of these two occurrences is *non-disjoint* with the occurrence of `PAY-REC` in line 1; this is because the type expression assigned to the occurrence of `PAY-REC` in line 1 contains both Salary and Stipend.

5 Future Work

This paper describes an approach for inferring several aspects of logical data models such as atomic types, record structure based on usage of variables in the

code, and guarded disjoint unions. In the future we plan to work on inferring additional desirable aspects of logical data models such as associations between types (e.g., based on foreign keys).

Within the context of the approach described in this paper, future work includes expanding upon the range of idioms that programmers use to implement union types that the algorithm addresses, expanding the power of the type system and algorithm, e.g., by introducing more expressive notions of value constraints, handling more language constructs (e.g., arrays, procedures), improving the efficiency of the algorithm, and generating "factored" types in the algorithm instead of sets of union-free types (e.g., $\alpha(\beta \oplus \gamma)\delta$, instead of $\{\alpha\beta\delta, \alpha\gamma\delta\}$).

References

1. G. Canfora, A. Cimitile, and G. A. D. Lucca. Recovering a conceptual data model from cobol code. In *Proc. 8th Intl. Conf. on Softw. Engg. and Knowledge Engg. (SEKE '96)*, pages 277–284. Knowledge Systems Institute, 1996.
2. B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *Proc. 24th Intl. Conf. on Softw. Engg.*, pages 313–324. ACM Press, 2002.
3. P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte. Annodomini: from type theory to year 2000 conversion tool. In *Proc. 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14. ACM Press, 1999.
4. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. 21st Intl. Conf. on Softw. Engg.*, pages 213–224. IEEE Computer Society Press, 1999.
5. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.
6. R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *Proc. 19th intl. conf. on Softw. Engg.*, pages 338–348. ACM Press, 1997.
7. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proc. 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 119–132. ACM Press, 1999.
8. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual (2nd Edition)*. Addison-Wesley Professional, 2004.
9. A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st Intl. Conf. on Softw. Engg.*, pages 246–255. IEEE Computer Society Press, 1999.
10. A. van Deursen and L. Moonen. Understanding COBOL systems using inferred types. In *Proc. 7th Intl. Workshop on Program Comprehension*, pages 74–81. IEEE Computer Society Press, 1999.
11. H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie-Mellon University, 1998.