Aggregate Structure Identification
and its Application to Program Analysis

G. Ramalingam John Field Frank Tip
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598, USA
{rama,jfield,tip}@watson.ibm.com

Abstract

In this paper, we describe an efficient algorithm for lazily decomposing aggregates such as records and arrays into simpler components based on the access patterns specific to a given program. This process allows us both to identify implicit aggregate structure not evident from declarative information in the program, and to simplify the representation of declared aggregates when references are made only to a subset of their components. We show that the structure identification process can be exploited to yield the following principal results:

- A fast type analysis algorithm applicable to program maintenance applications such as date usage inference for the "Year 2000" problem.
- An efficient algorithm for *atomization* of aggregates. Given a program, an aggregate atomization decomposes all of the data that can be manipulated by the program into a set of disjoint *atoms* such that each data reference can be modeled as one or more references to atoms without loss of semantic information. Aggregate atomization can be used to adapt program analyses and representations designed for scalar data to aggregate data. In particular, atomization can be used to build more precise versions of program representations such as SSA form or PDGs. Such representations can in turn yield more accurate results for problems such as program slicing.

Our techniques are especially useful in weakly-typed languages such as Cobol (where a variable need not be declared as an aggregate to store an aggregate value) and in languages where references to statically-defined subranges of data such as arrays or strings are allowed.

1 Introduction

Many algorithms for static analysis of imperative programs make the simplifying assumption that the data manipulated by a program consists of simple atomic values, when in reality aggregates such as arrays and records are usually predominant. There are several straightforward approaches to adapting analysis algorithms designed for scalars to operate on aggregates:

1. Treat each aggregate as a single scalar value.
2. Decompose each aggregate into a collection of scalars, each of which represents one of the bytes (or bits!) comprising the aggregate.
3. Use the declarative information in the program to break up each aggregate into a collection of scalars, each of which represents a declared component of the aggregate containing no additional substructures of its own.

Unfortunately, each of these approaches has drawbacks. (1) can yield very imprecise results. While (2) is likely to produce precise results it can be prohibitively expensive.

At first blush, (3) appears to be the "obvious" solution. However, it is unsatisfactory in weakly-typed languages such as Cobol, where a variable need not be explicitly declared as an aggregate in order for it to contain composite data. Even in more strongly-typed languages, declarative information alone can be insufficient because (i) loopholes in the type system (such as typecasts) may permit aggregate values to interoperate with non-aggregate values; and (ii) programmers may pack several scalars, each encoded using one or more bits, into a single word. Moreover, (3) may produce unnecessarily many scalar components when the program only accesses a subset of those components. In addition, in the presence of "unions" this approach can produce scalars that *overlap* one another in storage inexactly. The operation of determining whether two scalars in a program refer to overlapping storage (such checks are often required in the inner loops of analysis algorithms) can be costly.

In this paper, we present an efficient algorithm for lazily decomposing aggregates into simpler components based on the access patterns *specific to a given program*. This process allows us both to identify implicit aggregate structure not evident from declarative information, and to simplify the representation of declared aggregates when references are made only to a subset of their components. After atomization, each reference to an aggregate can be expressed as a set of references to disjoint *atoms*. The resulting atoms may then be treated as scalars for the purposes of analysis, and checks for overlapping storage reduce to equality tests on atoms. Atomization can thus serve as an "enabling" technique for performing various program analyses (e.g., computing reaching definitions [1] and program slicing [15]), as well as constructing their underlying representations (e.g., PDG [6] or SSA form [4]) in the presence of aggregates.

We also present a variant of the algorithm that can be used to efficiently solve certain type analysis problems. One instance of such a problem is date usage inference for programs affected by the "Year 2000" problem. This is an instance of a general class of problems that require inferring undeclared but related usages of type information for various software maintenance and verification activities [10]. The type analysis algorithm described here has been incorporated into several recent IBM products[1].

[1] IBM VisualAge 2000 for Cobol and IBM VisualAge 2000 for PL/I.

```
01 A.
    05 F1 PIC 99.
    05 F2 PIC 99.
    05 F3 PIC XX.
    05 F4 PIC XX.
01 B PIC X(8).
01 C PIC X(8).
01 D.
    05 F5 PIC 99.
    05 F6 PIC 99.
    05 F7 PIC XX.
    05 F8 PIC XX
01 RESULT PIC 99.

MOVE 17 TO F1.
MOVE 18 TO F2.
MOVE A TO B.
MOVE B TO C.
MOVE C TO D.
MOVE F5 TO RESULT.
```

Figure 1: Example Cobol program illustrating assignments between aggregate and non-aggregate variables.

```
01 DATA-RECORD.
   02 DATE.
      03 YY        PIC 99. // year
      03 MM        PIC 99. // month
      03 DD        PIC 99. // day
   02 PRINTABLE-DATE REDEFINES DATE PIC X(6).
   02 ...

01 OUTPUT-BUFFER.
   02 LINE         PIC X(80).
   02 COLUMNS REDEFINES LINE.
      05 COLUMN-1  PIC XX.
      05 COLUMN-2  PIC XX.
      05 ...

01 PRODUCT-INFORMATION.
   02 COLUMN-1-INFOPIC XX.
   02 COLUMN-2-INFOPIC XX.
   02 ...

MOVE FUNCTION CURRENT-DATE TO DATE OF DATA-RECORD.
...
MOVE PRINTABLE-DATE(1:2) TO COLUMN-1.
...
MOVE PRODUCT-INFORMATION TO OUTPUT-BUFFER.
```

Figure 2: Example illustrating type analysis for the Y2K problem.

1.1 Motivating Examples

Consider the Cobol fragment shown in in Fig. 1. For Cobol-illiterati, the declarations in the example behave as follows: The program contains top-level declarations of variables `A`, `B`, `C`, `D`, and `RESULT`. Variables `A` and `D` are both declared as records of four fields: `F1` through `F4`, and `F5` through `F8`, respectively. The types of these fields are declared using so-called *picture* clauses, which are a compact means of expressing both the length and the allowable types of the sequence of characters that constitute the fields. The characters that follow the keyword `PIC` specify the types of characters that are allowed at corresponding locations in the field. For instance, a '`9`' character indicates numerical data, whereas an '`X`' character indicates that any character is allowed. Hence, variables `A` and `D` both consist of 4 numeric characters followed by 4 unconstrained characters. A picture character may be followed by a parenthesized *repetition factor*. The non-aggregate variables `B` and `C` thus each consist of eight unconstrained characters. The example program contains a number of assignments. Note that in Cobol, it is not necessary to name parent structures in data references when field references alone are unambiguous (e.g., in the assignment of 17 to field `F1` of `A`).

Suppose we are interested in computing the backwards *program slice* [17, 15] with respect to the final value of `RESULT`, i.e., the set of assignments that could affect the final value of `RESULT`. Since our example program does not contain any control flow constructs, the slice contains any statement on which the final value of `RESULT` is transitively *data-dependent*. We assume that the following model is used to compute these data dependences:

- All variables are decomposed into disjoint *atoms* by some means.
- Each MOVE statement is modeled as a set of atomic assignments between corresponding atoms.
- Data dependences are determined by tracing def-use chains between atomic assignments.

Clearly, an atomization that is too crude will lead to redundant statements in the slice. For example, treating the statement `MOVE B TO C` as a scalar assignment between two atomic variables[2] will lead to the inclusion of the superfluous statement `MOVE 18 TO F2` in the slice. On the other hand, if the atomization is too fine-grained, the number of data dependences that must be traced to compute the slice will be larger than necessary and representations that capture these dependences (such as PDGs) will also be larger than necessary. For example, breaking up each variable into character-sized atoms leads to the desired slice (one that omits `MOVE 18 TO F2`). However, the same result can be achieved with the following, much coarser-grained atomization, which is produced by our atomization algorithm:

$$
\begin{aligned}
\textit{atomization}(\mathtt{A}) &= (\mathtt{A}[1{:}2], \mathtt{A}[3{:}4], (\mathtt{A}[5{:}8]) \\
\textit{atomization}(\mathtt{B}) &= (\mathtt{B}[1{:}2], \mathtt{B}[3{:}4], \mathtt{B}[5{:}8]) \\
\textit{atomization}(\mathtt{C}) &= (\mathtt{C}[1{:}2], \mathtt{C}[3{:}4], \mathtt{C}[5{:}8]) \\
\textit{atomization}(\mathtt{D}) &= (\mathtt{D}[1{:}2], \mathtt{D}[3{:}4], \mathtt{D}[5{:}8])
\end{aligned}
$$

Here, we use array notation to indicate subranges of the characters occupied by a variable. E.g., `B`[3:4] denotes the subrange consisting of character 3 and character 4 of variable `B`. There are a few interesting things to note about this solution:

- Fields `F1` and `F2` cannot be merged into a single atom without a loss of precision, and therefore correspond to separate atoms.
- Field `F3` and `F4` are merged, because the distinction between these fields is irrelevant for this particular program[3]. In general, merging fields can lead to faster dataflow analysis and more compact program representations.

[2] Note that this is a very reasonable choice, especially if we use only declarative information to perform the atomization.

[3] Unused fields occur frequently in Cobol applications. Cobol-based systems typically consist of a collection of persistent databases and a collection of programs that manipulate these databases. Although the declared record structure reflects the format of the database, a single application typically only uses a subset of the fields of the retrieved records. Hence, analysis of individual applications can benefit by coalescing or eliminating uninteresting fields.

`YY OF DATE OF DATA-RECORD`	$\rightarrow$	{*year*}
`MM OF DATE OF DATA-RECORD`	$\rightarrow$	{*notYear*}
`DD OR DATE OF DATA-RECORD`	$\rightarrow$	{*notYear*}
`PRINTABLE-DATE[1:2] OF DATA-RECORD`	$\rightarrow$	{*year*}
`PRINTABLE-DATE[3:4] OF DATA-RECORD`	$\rightarrow$	{*notYear*}
`PRINTABLE-DATE[5:6] OF DATA-RECORD`	$\rightarrow$	{*notYear*}
`LINE[1:2] OF OUTPUT-BUFFER`	$\rightarrow$	{*year*}
`COLUMN-1 OF COLUMNS OF OUTPUT-BUFFER`	$\rightarrow$	{*year*}
`COLUMN-1-INFO OF PRODUCT-INFORMATION`	$\rightarrow$	{*year*}

Figure 3: Result of type analysis applied to example in Fig. 2

- Although variables `B` and `C` are both *declared* as scalar variables, both must be partitioned into three atoms in order to obtain precise slicing results.

Fig. 2 shows a program fragment that manipulates dates in ways similar to those of Cobol programs affected by the "Year 2000" ("Y2K") problem. Here, `DATA-RECORD` represents a record containing date and non-date information. The storage for date information is redefined in two different ways: `DATE` is a structured record containing separate fields for month, day, and year digits, while `PRINTABLE-DATE` is an unstructured "string" of unconstrained characters intended to be used for input or output. Since the `YY` field of `DATE` is only two digits long, it would have to be expanded to four digits to account for post-1999 dates. In addition, `COLUMN-1` of `OUTPUT-BUFFER` (here representing a multi-purpose string used for input/output purposes) would have to be expanded to account for the fact that years are now larger. This could in turn affect `PRODUCT-INFORMATION` as well, since even though the latter never actually contains a year value, it would probably have to be updated to account for the fact that the first column of `OUTPUT-BUFFER` is now two characters wider.

Section 5 discusses how our aggregate structure identification algorithm can be extended to assist in remediating "field expansion" problems such as the Y2K problem by viewing it as a flow-insensitive, bidirectional *type analysis* problem. The basic idea is as follows: We first define a semi-lattice of *abstract types*. In the case of the Y2K problem, a lattice of subsets of the set { *year*, *notYear* } (where *year* and *notYear* are atomic types representing fields inferred to be year-related or not year-related, respectively) would suffice, although more complicated lattices could also be used. Known sources of year-related values, such as the year characters returned by the `CURRENT-DATE` library function in Fig. 2 are initialized to *year.* Sources of values known not to contain years (e.g., the non-year characters returned by `CURRENT-DATE`) are initialized to *notYear.* After applying the algorithm described in 5, the results of the type analysis are depicted in Fig. 3.

The interesting aspect of our analysis is not the type lattice itself, which is trivial, but the way in which the analysis is carried out efficiently and accurately on aggregates. This kind of analysis is applicable not only to the Y2K problem, but to other problems in which similar types must be propagated through aggregates, e.g., any problem involving field expansion of variables holding values of a particular logical (i.e., non-declared) type.

Fig. 4 depicts a more problematic example, in which an array, `MONTH`, is overlaid with a record, `MONTHS-BY-NAME`. Each field of `MONTHS-BY-NAME` corresponds to an element of the array `MONTH`). Overlaying of records and arrays is a fairly common idiom in Cobol. This allows programmers to refer to array elements by name as well as by index (e.g., when iterating uniformly through the collection represented by the array), and is also used to initialize arrays, as in this example. The use of such idioms makes it desirable to avoid overly conservative treatment of such

```
01 M.
   02 MONTH OCCURS 12 TIMES.
      05 NAME     PIC X(3).
      05 NUM-DAYS PIC 9(2).
   02 MONTHS-BY-NAME REDEFINES MONTH.
      05 JAN.
         10 NAME      PIC X(3) VALUE IS "JAN".
         10 NUM-DAYS  PIC 9(2) VALUE IS 31.
      05 FEB.
         10 NAME      PIC X(3) VALUE IS "FEB".
         10 NUM-DAYS  PIC 9(2) VALUE IS 28.
      ...
      ...
MOVE NUM-DAYS OF MONTH(I) TO ND.
```

Figure 4: Example Cobol program illustrating overlaid arrays and records.

overlays in the context of program analysis. For instance, in the context of reaching-definitions analysis, it is desirable to infer that the initializing definition of `NAME OF JAN` will not reach the use of `NUM-DAYS OF MONTH[I]`, but that the initializing definition of `NUM-DAYS OF JAN` might reach the same use.

Our aggregate structure identification algorithm differentiates between references to the array as a whole, references to array subranges with *statically-determinable* indices (references to the elements of `MONTHS-BY-NAME` in the example of Fig. 4 are treated as single-element instances of subrange references), and references to *arbitrary* elements via indices computed at run-time. These distinctions can be exploited to yield better atomizations that accurately differentiate among these cases.

1.2 Overview

The remainder of the paper proceeds as follows: In Section 2, we describe a tiny programming language that contains only the language features relevant to our results. Section 3 outlines the basic ideas behind the structure identification algorithm in terms of solving *equivalence constraints* on ranges of abstract memory locations; this algorithm manipulates a new data structure called the *Equivalence DAG*.

In Section 4, we observe that the algorithm of Section 3 can be viewed as computing the solution to a *unification* problem. Among other things, this alternate view allows certain other problems that can be expressed in terms of unification (e.g., Steensgaard's flow-insensitive pointer analysis [11, 12]) to be incorporated smoothly into our framework.

Sections 5 and 6 cover two refinements of the basic algorithm and their applications: Section 5 extends the framework of Section 3 to add inequality constraints involving elements of an abstract type lattice. The resulting type analysis algorithm is applicable to the Y2K problem. In Section 6, we formalize the atomization problem, and provide a solution based on another extension to the framework of Section 3.

The complexity of the Equivalence DAG construction algorithm (in its most general form) is discussed in Section 7. Extensions to the basic algorithm, including representation of variables of indeterminate length, pointer analysis, and uses of SSA renaming, are covered in Section 8. Section 9 is devoted to related work. Section 10 discusses possible future work. Finally, the appendix provides the details of the type analysis and atomization algorithms in pseudocode form.

$$\begin{array}{rcl} Pgm & ::= & \epsilon \mid Stmt\ Pgm \\ Stmt & ::= & DataRef \leftarrow DataRef \\ DataRef & ::= & ProgVars \mid \\ & & DataRef\,[\mathrm{Int}_+ : \mathrm{Int}_+] \mid \\ & & DataRef \setminus \mathrm{Int}_+ \end{array}$$

Figure 5: The mini-language under consideration. Here, Int_+ denotes the set of positive integers, and *ProgVars* denotes a set of "top level" program variables.

2 A Mini Language

In order to facilitate the discussion of problems studied and the algorithms presented in this paper, we will use a small language, the grammar of which is shown in Fig. 5. The language of Fig. 5 may be thought of as a traditional imperative language trimmed down to the bare essentials necessary to discuss the problems at hand. Since the algorithms we present are flow-insensitive, control-flow aspects of a program are irrelevant, and we consider a program $P \in Pgm$ to simply consist of a set of statements. A statement $d_1 \leftarrow d_2 \in Stmt$ represents an assignment which copies the contents of data reference d_2 into d_1.

A data reference $d \in DataRef$ is a reference to some sequence of abstract locations ("bytes") and takes one of the following forms:

- a *program variable* $x \in ProgVars$ (the length of which will be denoted by $|x|$)
- a *subrange* $d[i{:}j]$ of locations i through j of some data reference d
- a single, statically indeterminate *element* of an array of n elements, denoted by $d\backslash n$, where d is a data reference representing the complete array

Subranges are used to represent a sequence of locations at a *statically-determined position* in a data reference. For example, if d refers to a record, then $d[i{:}j]$ can be used represent a reference to a field of the record. A data-reference created by indexing an array is abstracted in our mini language into a reference of the form $d\backslash n$, where d identifies the complete array, and n is the number of elements in the array. (Thus, our abstraction omits the actual array index expression. If the index expression is a constant, however, the data reference can alternatively be represented as a subrange $d[i{:}j]$ of the array, where i and j delimit the range of locations occupied by the single array element denoted by the index expression.) The notation $d\backslash n$ is intended to suggest that if we break up the sequence of locations that d denotes into n subsequences of equal lengths, then $d\backslash n$ denotes *one* of these n different subsequences.

We now define the set of all *locations* as:

$$Loc = \{\langle x,\ i\rangle \ \mid\ x \in ProgVars,\ 1 \le i \le |x|\}$$

(Different elements of *ProgVars* thus represent disjoint sets of locations.) For convenience, we will denote location $\langle x,\ i\rangle$ as simply $x[i]$. At execution time, every data-reference denotes a sequence of locations. At analysis time, however, we may not know the precise sequence of locations referred to by a data-reference d in the general case (e.g., due to a reference to an array element at a statically-indeterminate index). Hence, we treat a data-reference d as a reference to *one* of a set of sequences of locations, and we will denote this set by $\mathcal{D}[d]$. Formally, we define $\mathcal{D}$ by:

$$\begin{array}{rcl} \mathcal{D}[x] & = & \{\, x[1]\cdot x[2]\cdots x[|x|]\,\} \text{ if } x \in ProgVars \\ \mathcal{D}[d[i{:}j]] & = & \{\, \sigma[i]\cdot\sigma[i+1]\cdots\sigma[j] \ \mid\ \sigma \in \mathcal{D}[d]\} \\ \mathcal{D}[d\backslash n] & = & \{\, \sigma[s]\cdot\sigma[s+1]\cdots\sigma[e] \ \mid \\ & & \quad \sigma \in \mathcal{D}[d], 1 \le i \le n, \\ & & \quad s = (i-1)*(|\sigma|/n)+1, e = i*(|\sigma|/n)\,\} \end{array}$$

where $\sigma[i]$ indicates the i-th element of sequence σ, and $|\sigma|$ denotes the length of a sequence σ. Note that all elements of $\mathcal{D}[d]$ have the same length, which we will denote $|d|$.

For example, let $x, y \in ProgVars$. Then, $x[3{:}5]$ denotes the singleton set $\{x[3]\cdot x[4]\cdot x[5]\}$. A more interesting example is $((y[1{:}10])\backslash 2)[2{:}3]$. Here, $y[1{:}10]\backslash 2$ is a reference to an arbitrary element of a 2-element array; the array as a whole occupies the first 10 locations of y. The subrange [2:3] (which could, e.g., represent a single field when the array element is a record) is then selected from the element. As a result, the set of locations referred to consists of $\{y[2]\cdot y[3], y[7]\cdot y[8]\}$. In other words, $((y[1{:}10])\backslash 2)[2{:}3]$ is a reference to *either* locations $y[2]$ *and* $y[3]$ *or* locations $y[7]$ *and* $y[8]$.

We will now define an abstract semantics $\mathcal{S}(d_1 \leftarrow d_2)$ for the assignment statement $d_1 \leftarrow d_2$, which simply consists of the set of all pairs of locations (l_1, l_2), written symbolically as $l_1 \leftarrow l_2$, such that the assignment statement might copy the contents of location l_2 to location l_1. This semantics is defined as follows:

$$\begin{array}{rl} \mathcal{S}(d_1 \leftarrow d_2) = & \{\, \sigma_1(i) \leftarrow \sigma_2(i) \ \mid\ \sigma_1 \in \mathcal{D}[d_1], \sigma_2 \in \mathcal{D}[d_2], \\ & \quad 1 \le i \le \min(|\sigma_1|, |\sigma_2|)\,\} \end{array}$$

In the rest of the paper, we will assume that for every statement $d_1 \leftarrow d_2$, $|d_1| = |d_2|$.

The abstract semantics for assignments that occur in a given program $P \in Pgm$ can be used to define an equivalence relation on locations that will be useful in the sequel (e.g., as the basis for inferring equivalent types). To this end, we first define:

$$E = \bigcup_{d_1 \leftarrow d_2 \in P} \mathcal{S}(d_1 \leftarrow d_2)$$

Now, let $\equiv_P$ denote the smallest equivalence relation containing the set of pairs of locations E (*i.e.*, the equivalence closure of E). We will omit the subscript P if no confusion is likely.

3 The Equivalence DAG: The Basic Ideas Behind the Algorithm

In this section we focus on the problem of computing the equivalence relation $\equiv_P$, given a program P. The goal of this section is to give the reader an understanding of the essential algorithmic contributions of this paper, primarily through examples. We will describe extensions and applications of this algorithm in subsequent sections.

Rather than generate an explicit representation of the equivalence relation $\equiv_P$, we will actually generate a more compact representation of the equivalence relation that can be used to answer queries about whether two locations are equivalent or not. We will also refer to a statement $d_1 \leftarrow d_2$ as an equivalence *constraint* $d_1 \simeq d_2$ for notational convenience.

The Simple Equivalence Closure Problem

We start with a simple version of the problem, where every constraint has the form $x \simeq y$, given $x, y \in ProgVars$. In this case, $\simeq$ induces an equivalence relation on *ProgVars*. This is sufficient to answer questions of equivalence of locations since $\langle x,\ i\rangle \equiv_P$

$\langle y,\ j\rangle$ if and only if variables x and y are in the same equivalence class and $i = j$. Thus, in this case, the set of equivalence classes of *ProgVars* provides a compact representation of the equivalence relation on *locations*.

The partitioning of *ProgVars* can be done in the standard way: initially place every program variable $x \in$ *ProgVars* in an equivalence class by itself, and then process the equivalence constraints one by one; a constraint $x \simeq y$ is processed by merging the equivalence classes to which x and y belong into one equivalence class, using the well-known union-find data structure[4] (see [14, 3]).

The Range Equivalence Closure Problem

Now, consider a version of the problem where every constraint is of the form $x[i{:}j] \simeq y[k{:}l]$, where $x, y \in$ *ProgVars*. There are two aspects to the original solution that we would like to preserve when we address this generalized problem. The first aspect is that the algorithm processes every constraint in C exactly once, instead of using an iterative (*e.g.*, transitive-closure-like) algorithm. The second is that we would like to identify "ranges" of locations that are equivalent to each other and partition them into equivalence classes. This can represent $\equiv_C$ more compactly than a partition of the set of all locations into equivalence classes.

We now illustrate through an example how we can achieve these goals. Assume that $W, X, Y, Z \in$ *ProgVars*, and that $|W| = 6$, $|X| = 12$, $|Y| = 8$, and $|Z| = 12$. Assume that C consists of three equivalence constraints, $X[5{:}8] \simeq Y[1{:}4]$, $Z[1{:}6] \simeq W[1{:}6]$, and $X[3{:}12] \simeq Z[1{:}10]$. We begin by placing every variable in an equivalence class by itself. We then process the first constraint $X[5{:}8] \simeq Y[1{:}4]$ as follows. We "split" the range $X[1{:}12]$ into three sub-ranges $X[1{:}4]$, $X[5{:}8]$, and $X[9{:}12]$ and place them each in an equivalence class by itself. We refer to this as adding "breakpoints". We similarly split range $Y[1{:}8]$ into two sub-ranges $Y[1{:}4]$ and $Y[5{:}8]$, placing them each in an equivalence class by itself. We then merge the equivalence classes to which $X[5{:}8]$ and $Y[1{:}4]$ belong into one.

Given this kind of a partition of every program-variable into a collection of sub-ranges, every location belongs to a unique sub-range of the partition. We can map every location l into a pair (e_l, o_l) where e_l is the equivalence class of the unique sub-range containing l and o_l is the offset of the location within that sub-range. Further, locations l_1 and l_2 are equivalent with respect to the relation $\equiv_P$ if and only if the $e_{l_1} = e_{l_2}$ and $o_{l_1} = o_{l_2}$. For example, location $X[6]$ will be mapped to $(ec(X[5{:}8]), 2)$ where $ec(r)$ denotes the equivalence class containing sub-range r. Similarly, location $Y[2]$ will be mapped to $(ec(Y[1{:}4]), 2)$. Since $ec(X[5{:}8]) = ec(Y[1{:}4])$, these two locations are equivalent.

Let us re-visit the step where we "split" a range, say $X[1{:}12]$, into a sequence of sub-ranges, say $X[1{:}4]$, $X[5{:}8]$, and $X[9{:}12]$. It turns out to be convenient to keep both the original range and the new sub-ranges around, and to capture the "refinement" relation between these into a tree-like representation (rather than, for instance, *replacing* the original range by the new sub-ranges). Fig. 6(a) and Fig. 6(b) illustrate how we represent the refinement of X and Y for the above example. Each rectangle in the figure, which we will refer to as a "node", denotes an equivalence class of sub-ranges, and the number inside indicates the length of each sub-range contained in the equivalence class. Fig. 6(c) indicates that the equivalence classes containing the nodes representing $X[5{:}8]$ and $Y[1{:}4]$ have been merged into a single equivalence class[5].

[4] It can be done even more efficiently using the linear time algorithm for computing the connected components of an undirected graphs. However, we will need the flexibility of the union-find data structure in a generalized version of the problem.

[5] Note that edges whose targets are nodes representing equivalence classes to be merged are not literally redirected to a common node, instead, the union-find data structure is used to merge the classes to which the edges refer.

The next constraint $(Z[1{:}6], W[1{:}6])$ is processed just like the first constraint, as illustrated by Fig. 6(d-e).

In the general case, processing a constraint $d_1 \simeq d_2$ consists of the following steps. (i) We first add break-points to the representation before the starting-location and after the ending-location of both d_1 and d_2. (ii) The sub-ranges d_1 and d_2 can then be represented by a *sequence* of nodes, say $\sigma_1 = [s_1, \cdots, s_k]$ and $\sigma_2 = [t_1, \cdots, t_m]$ respectively. We make these two sequences equivalent to each other as follows: if s_1 and t_1 denote ranges of the same length, we simply merge the two into one equivalence class and proceed with the remaining elements of the sequence. If the two denote ranges of different lengths, we then split the bigger range, say s_1, into two sub-ranges, s_1' and s_1'', such that s_1' has the same length as t_1. We then merge s_1' with t_1, and continue on, making the sequences $[s_1'', s_2, \cdots, s_k]$ and $[t_2, \cdots, t_m]$ equivalent.

The third constraint $X[3{:}12] \simeq Z[1{:}10]$ illustrates the more general scenario described above. After adding the necessary breakpoints, the range $Z[1{:}10]$ is represented by the sequence $[s_1, s_2]$ (see Fig. 6(f)), while the range $X[3{:}12]$ is represented by the sequence $[t_1, t_2, t_3]$. s_1 is longer than t_1, and is broken up into subranges s_1' and s_1'', as shown in Fig. 6(g). We then merge t_1 with s_1', t_2 with s_1'', and t_3 with s_2. Fig. 6(h) shows the resulting representation.

Clearly, given a location l, we can "walk" down the DAG (shown in Fig. 6(h)), from the appropriate root to a leaf e_l to map the location to a pair (e_l, o_l) such that $l_1 \equiv l_2$ if and only if $(e_{l_1}, o_{l_1}) = (e_{l_2}, o_{l_2})$. We call the representation generated by this algorithm an *Equivalence DAG*.

In the above description of the algorithm, we assumed that the nodes in the sequences σ_1 and σ_2 were "leaf" nodes. Even if that were true when the processing of the two sequences begins, when we get around to processing elements s_i and t_j, the processing of the earlier elements of the sequences could have had the effect of adding breakpoints to either s_i or t_j or both, converting them into "internal" nodes. Our algorithm handles this by converting the subranges involved into a sequence of leaf nodes lazily rather than eagerly.

A simple example that illustrates this is the single constraint $A[1{:}12] \simeq A[5{:}16]$. Adding breakpoints corresponding to the endpoints of the two subranges $A[1{:}12]$ and $A[5{:}16]$ generates the representation shown in Fig. 7(b). The processing of the constraint then proceeds as below:

$$
\begin{array}{rcl}
A[1{:}12] & \simeq & A[5{:}16] \\
 & \Downarrow & \\
[u_2] & \simeq & [u_5, u_3] \\
 & \Downarrow & \text{(replace } u_2 \text{ by its children)} \\
[u_4, u_5] & \simeq & [u_5, u_3] \\
 & \Downarrow & \text{(split } u_5 \text{ into } u_6 \text{ and } u_7\text{)} \\
[u_4, u_5] & \simeq & [u_6, u_7, u_3] \\
 & \Downarrow & \text{(merge } u_4 \text{ and } u_6\text{)} \\
[u_5] & \simeq & [u_7, u_3] \\
 & \Downarrow & \text{(replace } u_5 \text{ by its children)} \\
[u_6, u_7] & \simeq & [u_7, u_3] \\
 & \Downarrow & \text{(merge } u_6 \text{ and } u_7\text{)} \\
[u_7] & \simeq & [u_3] \\
 & \Downarrow & \text{(merge } u_7 \text{ and } u_3\text{)} \\
[] & \simeq & []
\end{array}
$$

This example illustrates the motivation behind our representation. Note that if we maintained for each variable only the list of subranges into which it has been refined (instead of the tree

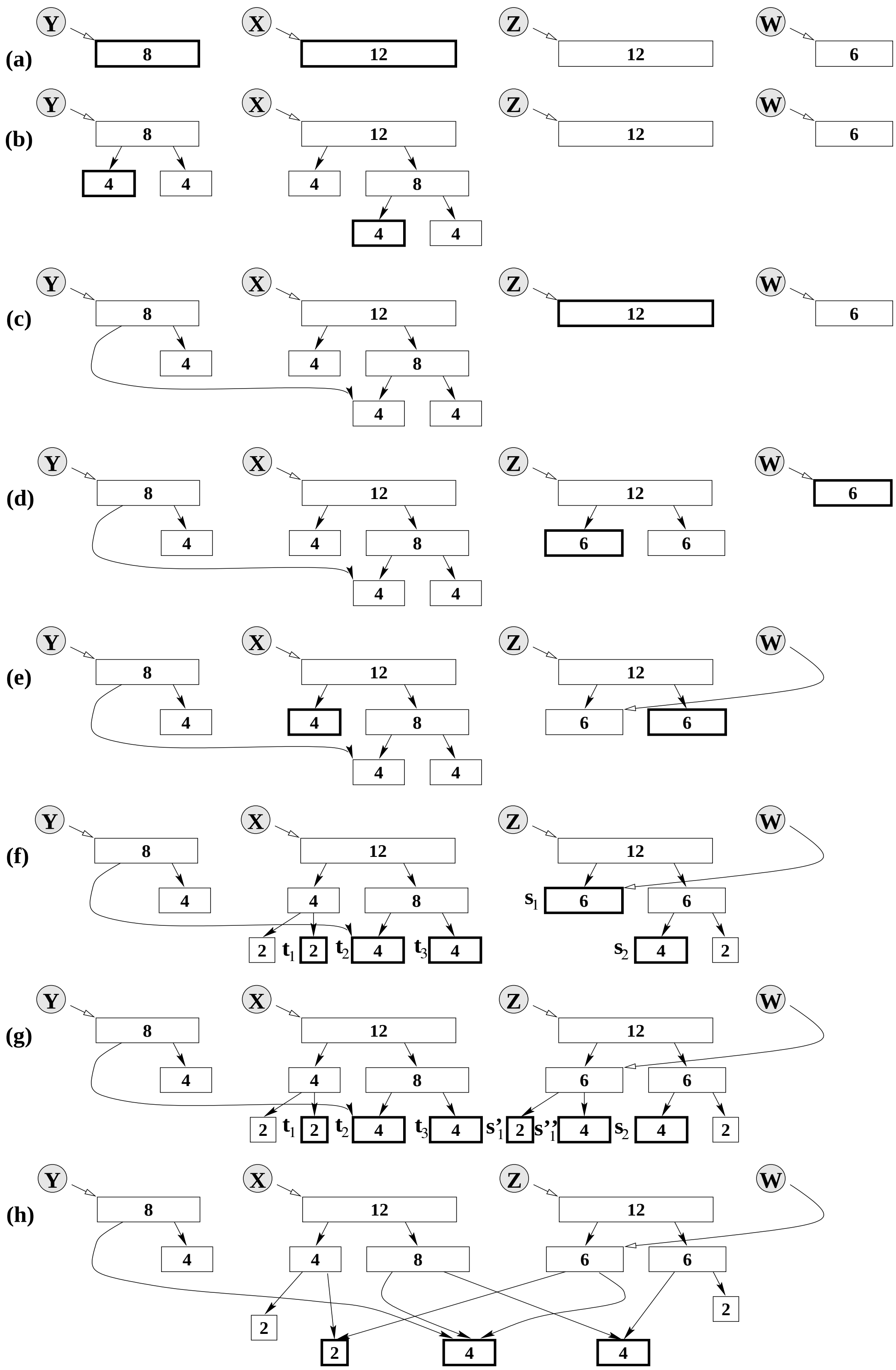

Figure 6: An example illustrating our range equivalence algorithm.

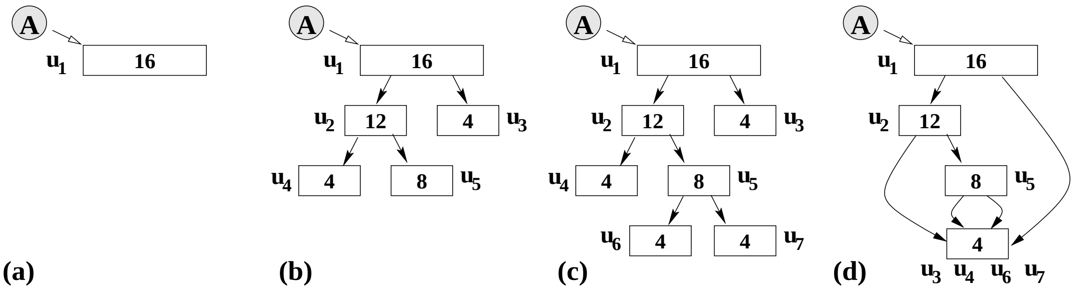

Figure 7: Another example illustrating our range equivalence algorithm.

representation of the refinement), processing constraints such as $A[1{:}12] \simeq A[5{:}16]$ will be more difficult. Our algorithm may be easier to understand if it is viewed as a sort of unification, with the leaf nodes denoting unbound variables and internal nodes denoting bound variables. We will explore this connection briefly in Section 4.

The General Problem

In the most general version of the problem we consider here, an equivalence constraint $d_1 \simeq d_2$ may consist of arbitrary data references as defined by the grammar in Fig. 5, including references to (statically indeterminate) array elements. Consider, for example, a case in which we have $P, Q, R \in$ *ProgVars*, with $|P| = 20$, $|Q| = 10$, and $|R| = 2$. Assume we have two constraints $P[1{:}10] \simeq Q[1{:}10]$ and $(P[1{:}20])\backslash 10 \simeq R[1{:}2]$. The first constraint is processed as before, producing the representation shown in Fig. 8(a). Processing the second constraint, which includes an array reference, produces the representation shown in Fig. 8(b). The nodes labeled u and v represent arrays consisting of 5 elements of size 2 each.

We will explain in detail how our algorithm handles arrays and similar constructs in Section 6. A complete algorithm for the general version of the problem appears in pseudocode form in the appendix.

4 The Equivalence DAG as a Unifier

Readers familiar with unification may have observed that our algorithm has a unification flavor. Our algorithm can, in fact, be thought of as unifying terms belonging to a term language $\Gamma_{\mathcal{V}}$ (defined below), with the following distinction: unlike in standard unification, we do not treat the operators $\oplus$ and $\otimes$ in this term language as free, i.e., uninterpreted operators; instead, we are interested in unification with respect to a specific interpretation of these operators. We explore this connection briefly in this section.

For any set X, let Γ_X denote set of terms defined by:

$$\Gamma_X ::= X \mid \Gamma_X \oplus \Gamma_X \mid \mathrm{Int}_+ \otimes \Gamma_X \tag{1}$$

where Int_+ denotes the set of positive integers. Occasionally we will omit $\otimes$, abbreviating $i \otimes \tau$ to $i\tau$. Let $\mathcal{V} = \cup_{i>0} V_i$ denote a set of variables. A variable belonging to V_i is said to have a *length* i. We will use the notation $x : i$ to indicate that a variable x has a length i.

Consider now the set of terms $\Gamma_{\mathcal{V}}$. Observe that we may interpret the "trees" rooted at any node in the Equivalence DAG as terms belonging to $\Gamma_{\mathcal{V}}$: leaves are interpreted as variables belonging to $\mathcal{V}$; internal nodes denoting the concatenation of two ranges, such as those in Figure 6, may be interpreted as uses of the operator $\oplus$; nodes such as u and v of Fig. 8, representing arrays, are interpreted as uses of the operator $\otimes$.

Let X^* denote the set of sequences of elements from X. Given a term $\tau \in \Gamma_X$, the value $[\![\tau]\!] \in X^*$ is obtained from τ by interpreting $\oplus$ as sequence concatenation and $\otimes$ as repeated concatenation (of a sequence with itself, as many times as indicated)

Define the *length* of a term $\tau \in \Gamma_{\mathcal{V}}$ to be the sum of the lengths of all variables in the sequence $[\![\tau]\!]$. A *substitution* σ is a length-preserving mapping from $\mathcal{V}$ to $\Gamma_{\mathcal{V}}$: *i.e.*, a function that maps every variable to a term that has the same length as the variable. The Equivalence DAG can be thought of as a substitution, restricted to a set of variables denoting program variables. For example, the DAG in Fig. 8(b) represents a substitution $\{x_Q \mapsto 5x_R, x_P \mapsto (5x_R)\oplus(5x_R), x_R \mapsto x_R\}$.

Two substitutions σ_1 and $sigma_2$ are said to be equivalent if $[\![\sigma_1(x)]\!] = [\![\sigma_2(x)]\!]$ for all x. Every substitution σ can be extended to map every term $\tau \in \Gamma_{\mathcal{V}}$ to a term $\sigma(\tau) \in \Gamma_{\mathcal{V}}$. A substitution σ is said to be a *unifier* for a set of *unification constraints* $S \subseteq \Gamma_{\mathcal{V}} \times \Gamma_{\mathcal{V}}$ if $[\![\sigma(\tau_1)]\!] = [\![\sigma(\tau_2)]\!]$ for every $(\tau_1, \tau_2) \in S$. Further, it is said to be a *most general unifier* for S if every other unifier σ_1 for S can be expressed as the composition of some substitution σ_2 with a substitution σ_3 that is equivalent to σ: $\sigma_1 = \sigma_2 \circ \sigma_3$.

The translation rules in Fig. 9, specified in the Natural Semantics style, show how a set of unification constraints can be generated from a program. In particular, the translation rule S shows how a statement $d_1 \leftarrow d_2$ can be translated into a set of unification constraints of the form $\tau_1 \cong \tau_2$ where $\tau_1, \tau_2 \in \Gamma_{\mathcal{V}}$. The auxiliary rules D_1, D_2, and D_3 show how a data-reference $d \in$ *DataRef* can be translated into a variable $x \in \mathcal{V}$ and a set of unification constraints C constraining variable x (which we denote by $d \Rightarrow_D (x, C)$). For example, if we have a data reference of the form $d\backslash n$ (rule D_3), then we represent d by a variable, say x, and $d\backslash n$ by a variable, say y, where the variables x and y are related by the constraint $n \otimes y \cong x$. The constraints generated from a program are simply the union of the constraints generated from the statements in the program.

Let us illustrate this using the example of Figure 8. Here we have $P, Q, R \in$ *ProgVars*, with $|P| = 20$, $|Q| = 10$, and $|R| = 2$. We have two constraints $P[1{:}10] \simeq Q[1{:}10]$ and $(P[1{:}20])\backslash 10 \simeq R[1{:}2]$. We represent every program variable $\mathtt{V} \in$ *ProgVars* by a constraint variable $x_{\mathtt{V}}$ of length $|\mathtt{V}|$. Processing the constraint $P[1{:}10] \simeq Q[1{:}10]$ produces the substitution $\{x_Q \mapsto u : 10, x_P \mapsto (u : 10)\oplus(v : 10)\}$, where u and v are two new variables, which is represented by the Equivalence DAG of Figure 8(a). Now consider the constraint $(P[1{:}20])\backslash 10 \simeq R[1{:}2]$. This is effectively translated into the unification constraint $x_P \cong 10 \otimes x_R$ (ignoring some superfluous variables that may be generated by a direct application of the translation rules of Figure 9). Since x_P is already bound to $(u{:}10)\oplus(v{:}10)$, our algorithm unifies $(u{:}10)\oplus(v{:}10)$ with $10 \otimes (x_R{:}2)$. This requires splitting up $10\otimes(x_R{:}2)$ into $5\otimes(x_R{:}2)\oplus 5\otimes(x_R{:}2)$. The subsequent unification binds both u and v to $5 \otimes (x_R{:}2)$.

It is is easy to see that the Equivalence DAG construction algorithm can be interpreted as computing a unifier for the constraints

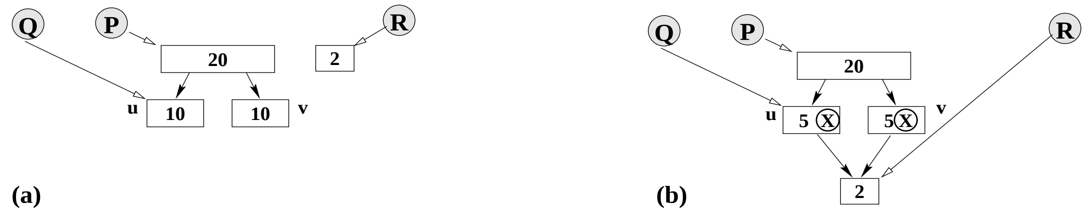

Figure 8: An example illustrating our array equivalence algorithm.

D_1	$\dfrac{\mathtt{V} \in \mathit{ProgVars}}{\mathtt{V} \Rightarrow_D (x_{\mathtt{V}}, \{\})}$	where a unique variable $x_{\mathtt{V}}$ is used for every program variable V
D_2	$\dfrac{d \Rightarrow_D (x, C)}{d[i:j] \Rightarrow_D (x_2, C \cup \{x_1 \oplus x_2 \oplus x_3 \cong x\})}$	where x_1, x_2, and x_3 are fresh variables of lengths $i-1$, $j-i+1$, and $\lvert x\rvert - j$ respectively
D_3	$\dfrac{d \Rightarrow_D (x, C)}{d \backslash n \Rightarrow_D (y, C \cup \{n \otimes y \cong x\})}$	where y is a fresh variable of length $\lvert x\rvert / n$
S	$\dfrac{d_1 \Rightarrow_D (x_1, C_1),\ d_2 \Rightarrow_D (x_2, C_2)}{d_1 \leftarrow d_2 \Rightarrow_S C_1 \cup C_2 \cup \{x_1 \cong x_2\}}$	

Figure 9: Generating unification constraints from a program.

generated in Figure 9. We also believe that the unifier it computes is most general, although we have not as of yet constructed a formal proof.

5 Application I: Type Analysis

The Problem

In this section we extend the basic algorithm of Section 3 to address a generalization of the Y2K type analysis problem discussed in Section 1. Consider the example depicted in Fig. 6. Assume the programmer wants to modify the representation of a field of the variable W, say $W[1{:}2]$. The question we wish to answer is "What other variables are likely to be affected by this change, requiring corresponding modifications to their own representation?"

We now present a more precise formulation of the problem. Let $\mathcal{L}$ denote some semi-lattice with a join operator $\cup$. We may think of the elements of $\mathcal{L}$ as denoting abstract types. An example is the lattice of subsets of the set $\{\mathit{year}, \mathit{notYear}\}$ used for year usage inference in the example of Fig. 2 of Section 1.

A function π from *Loc* to $\mathcal{L}$ represents a typing for the set of locations. We say that π is *valid* with respect to a program P if $\pi(l_1) = \pi(l_2)$ for all $l_1 \equiv_P l_2$. In other words, a typing π is valid with respect to P if the expressions on both sides of every assignment in P have the same type under π.

Now consider a constraint of the form $d \geq c$, where $d \in \mathit{DataRef}$ and $c \in \mathcal{L}$. We say that π satisfies this constraint iff

$$\pi(l) \geq c \text{ for every } l \in \bigcup_{\sigma \in \mathcal{D}[d]} set(\sigma)$$

where $set(\sigma)$ denotes the set of elements in a sequence σ. Given two typing functions π_1 and π_2, we say that $\pi_1 \geq \pi_2$ iff $\pi_1(l) \geq \pi_2(l)$ for all $l \in \mathit{Loc}$.

Given a program P and a set C of constraints of the form $d \geq c$, where $d \in \mathit{DataRef}$ and $c \in \mathcal{L}$, we are interested in computing the least typing valid with respect to P that satisfies every constraint in C.

The Solution

We now illustrate how we can efficiently compute the desired solution, using the data structure and algorithm presented in Section 3. We first process the equivalence constraints induced by program P to produce the Equivalence DAG. We then associate every leaf of the resulting DAG with a value from $\mathcal{L}$, which is initially the least element of $\mathcal{L}$. We then process every constraint $d \geq c$ in C as follows. The data-reference d can be mapped onto a sequence of leaves in the Equivalence DAG (possibly after adding new breakpoints). We update the value associated with each of these leaves to the join of their current value and c. The example in Fig. 10 illustrates this. Assume we start with the Equivalence DAG of Fig. 6(h), and process constraint $W[1{:}2] \geq \{\mathit{yearRelated}\}$. (We are using the lattice $\mathcal{L}$ of subsets of the set $\{\mathit{yearRelated}, \mathit{notYearRelated}\}$ described above.) $W[1{:}2]$ corresponds to the single leaf u (shown as a bold rectangle in Fig. 10), whose value is then updated to $\{\mathit{yearRelated}\}$.

The resulting DAG can be viewed as a compact representation of the desired solution. In particular, it can be interpreted as a function π from locations to $\mathcal{L}$: the value associated with any location l is obtained by traversing the DAG to map location l to a leaf e_l in the DAG (as explained earlier), whose value yields $\pi(l)$. In particular, the DAG in Fig. 10 maps locations $X[3]$, $X[4]$, $Z[1]$, $Z[2]$, $W[1]$, and $W[2]$ to type $\{\mathit{yearRelated}\}$ and every other location to type $\{\}$. Equivalently, the DAG in Fig. 10 may be viewed as a function mapping every program variable to a type term belonging to $\Gamma_{\mathcal{L}}$, where Γ is defined as in Equation 1.

6 Application II: Atomization

In this section, we address the aggregate atomization problem through another extension of the basic algorithm of Section 3. We first consider some examples that illustrate several of the more subtle aspects of the problem.

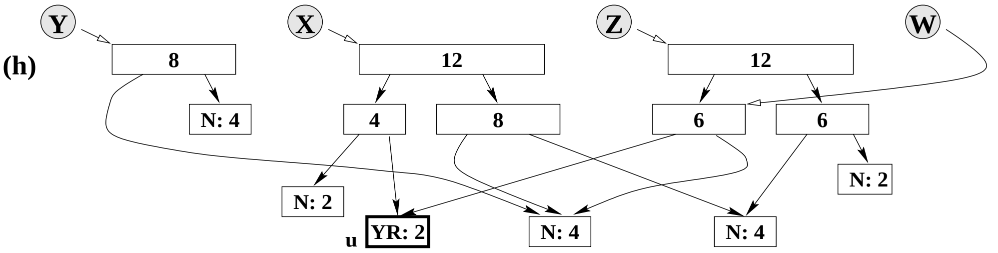

Figure 10: An example illustrating our type inference algorithm. We have used **N** as an abbreviation for $\{\}$ and **YR** as an abbreviation for $\{$*yearRelated*$\}$.

Motivation

Example: Overlapping Data References Consider the problem of computing the *reaching definitions* for the use of `NUM-DAYS OF MONTH[I]` in the example shown in Figure 4. In the absence of aggregates, determining whether any two direct (i.e., non-pointer) data references in a program can refer to the same set of locations is usually straightforward. However, in the example of Figure 4, Cobol's aggregate model considerably complicates matters: we note that the initialization of `NAME OF JAN` (M[1:3] in our mini-language) does not reach the use of ((M[1:60] $\setminus$ 12) [4:5] in our mini-language), but the initialization of `NUM-DAYS OF JAN` (M[4:5] in our mini-language) does. This follows from the fact that M[4:5] overlaps (M[1:60] $\setminus$ 12) [4:5], while M[1:3] does not. It should be evident from this example that testing for overlap between two data references has the potential to be quite expensive, especially in the presence of arrays.

Partially Killing Definitions Reaching definitions analysis is further complicated by the fact that one definition may "partially" kill another definition. Consider the following example in our mini language:

$$\text{S1: } x[1{:}10] \leftarrow y[1{:}10]$$
$$\text{S2: } x[1{:}5] \leftarrow z[1{:}5]$$
$$\text{S3: } w[1{:}5] \leftarrow x[1{:}5]$$
$$\text{S4: } z[1{:}5] \leftarrow x[6{:}10]$$

Clearly, the definition of x[1:10] at S1 does not reach the use of x[1:5] at S3, but it does reach the use of x[6:10] at S4, because the definition of x[1:5] at S2 partially kills the definition of x[1:10] at S1. Although reaching definitions analysis can be performed in several different ways, the example illustrates the need to handle partially killing definitions accurately in order to compute precise results.

The goal of atomization is to transform the input program into a semantically equivalent program in which all data references are atomic, thereby simplying program analyses such as the computation of reaching definitions. In the case of the example above, we can transform statements S1–S4 so that each of the assignments is defined in terms of atoms $x1$, $x2$, $y1$, $y2$, $z1$, and $w1$. The set of atoms partitions the set of locations such that every atom identifies a set of locations, and distinct atoms refer to disjoint sets. For instance, atom $x1$ identifies the set of locations x[1:5], and atom $x2$ identifies the set of locations x[6:10]. (The statement S1 can be thought of as an abbreviation for two assignment statements.)

$$\text{S1: } (x1, x2) \leftarrow (y1, y2)$$
$$\text{S2: } x1 \leftarrow z1$$
$$\text{S3: } w1 \leftarrow x1$$
$$\text{S4: } z1 \leftarrow x2$$

As with reaching definitions analysis, many other standard program analysis techniques or transformations (e.g., partitioned analysis, SSA-form construction, or program slicing) are not immediately applicable to programs containing overlapping data references or partially killing definitions. However, once a program is transformed into an equivalent one containing only operations on atoms, these complications can be ignored, since the atoms can be treated as simple scalar variables.

Atomization: The Basic Ideas

It should be clear from the preceding examples that an atom is intended to denote a set of locations that may be treated as a single logical unit for purposes of program analysis (we will make this notion more precise in the sequel). Recall from Section 5 that the leaves of the Equivalence DAG identify subranges of locations that can be treated as a single logical unit during type analysis. It should therefore not be surprising that the Equivalence DAG can also be used as the starting point for atomization.

However, we need to exercise some caution in treating the leaves of the Equivalence DAG as the atoms in the program. Due to the sharing present in the DAG structure, a leaf may identify a number of different subranges of locations in the program, and each of these subranges must be treated as a distinct atom. This can be done by "flattening" the Equivalence DAG into a forest first (by duplicating shared subgraphs), then treating the leaves of the resulting forest as atoms.

To formalize this intuition, we first note that the Equivalence DAG can be interpreted as a function φ mapping every $\mathtt{v} \in$ *ProgVars* to a term in $\Gamma_{\mathcal{V}}$. Let φ_r be the substitution obtained by renaming the variable *occurrences* in the set of terms $\{\varphi(x) \mid x \in \textit{ProgVars}\}$ such that no variable occurs twice. For example, if *ProgVars* = $\{A, B\}$, and $\varphi = \{A \mapsto w \oplus y \oplus w, B \mapsto y \oplus z\}$, then a suitable renaming is $\varphi_r = \{A \mapsto x_1 \oplus x_2 \oplus x_3, B \mapsto x_4 \oplus x_5\}$. By renaming multiple occurrences of the same variable, φ_r abstracts away type equivalence information, while allowing us to determine for any program variable the aggregate structure that was inferred during the construction of the Equivalence DAG. In the absence of arrays, we can then identify atoms with *the variables occurring in the set of terms* $\{\varphi_r(x) \mid x \in \textit{ProgVars}\}$; we will use *Atoms* to denote this set.

Once the atoms have been identified, the next step is to express all data references in the program in terms of the atoms. In particular, we can replace every data reference by a *sequence* of atomic references. In the above example, the reference to x[1:10] can be replaced by the sequence $(x1, x2)$. Our atomization algorithm guarantees that every assignment statement in the resulting program will have the form $(a_1, a_2, \cdots, a_n) \leftarrow (b_1, b_2, \cdots, b_n)$, where every a_i and b_i is an atom. If desired, such a statement can be replaced by n simpler statements of the form $a_i \leftarrow b_i$.

Dealing With Arrays

Consider the following example:

```
L1:  MOVE ...  TO A(5).
L2:  MOVE ...  TO A(I).
L3:  MOVE ...  TO A(6).
L4:  MOVE A(5) TO ...
```

A precise reaching definitions analysis can establish that definitions L1 and L2 reach the use in L4, while definition L3 does not. To extend the atomization approach discussed in the previous section to arrays in such a way that no information is lost, we must ensure that our atomization distinguishes references to statically indeterminate array elements from references to statically determinate array elements. Among other things, this means that `A(5)` and `A(6)` must be represented by distinct atoms (say, a_5 and a_6), and that `A(I)` must refer to one of a *set* of atoms (containing a_5, a_6, and additional atoms required to represent the other elements of `A`).

More generally, consider the example of Figure 8. Here we must be able to model a reference to an entire array (P[1:20]), a reference to an array subrange (P[1:10]), as well as a reference to a statically indeterminate arbitrary array element (P[1:20]$\backslash 10$.

To properly account for the various classes of array references discussed above, we must slightly relax the assumption in the previous section that all data references in the original program could be redefined entirely in terms of sets of atoms. Borrowing from the notation for array element references in our mini-language, we will instead replace data references in the original program by *atomic data references*, which are defined as follows:

$$\mathit{DataRef}_{Atomic} \quad ::= \quad \mathit{Atoms} \setminus \mathrm{Int}_{+}$$

Intuitively, $x\backslash n \in \mathit{DataRef}_{Atomic}$ represents an indeterminate reference to exactly one element of a *set* of n determinate references. We will refer to n as the *multiplicity* of the reference. Note that the *union* of locations referred to by $x\backslash n$ need not itself be contiguous sequence of locations, e.g., when $x\backslash n$ refers to a single field of an indeterminate element of an array of multi-field records. In the limit case, $x\backslash 1$ is used for all determinate references. Not unexpectedly, such references include contiguous ranges of locations such as simple scalar variables and references to entire arrays. However, determinate references may also denote noncontiguous ranges of locations, e.g., given an array of records, each of which contains two fields `F1` and `F2`, the collection of references to the `F2` fields of all array elements.

Consider, for example, the Equivalence DAG of Figure 8, which represents the substitution

$$\varphi = \{x_Q \mapsto 5x_R, x_P \mapsto (5x_R)\oplus(5x_R), x_R \mapsto x_R\}$$

Renaming the variables gives us

$$\varphi_r = \{x_Q \mapsto 5a_1, x_P \mapsto (5a_2)\oplus(5a_3), x_R \mapsto a_4\}$$

with atoms a_1, a_2, a_3, and a_4. Note that atoms a_2 and a_3 represent different parts of the array x_P. The data reference P[1:20]$\backslash 10$ represents an arbitrary element of the array. Consequently, its representation in terms of atomic data references is given by $\{a_2\backslash 5, a_3\backslash 5\}$. In contrast, the full array P[1:20] is represented by the singleton set $\{a_2\backslash 1 \cdot a_3\backslash 1\}$. We see from this example that if an array has been "fragmented" into several sub-arrays, then a reference to an arbitrary element of the original array must be represented as a reference to an element of *one of the* array subranges resulting from the fragmentation. In the general case, a data reference must be replaced by a *set* of sequence of atomic references.

Recall from Section 2 that we defined the semantics $\mathcal{D}[d]$ of a data reference d as a *set of sequence of locations*. The atomization process allows us to think of an arbitrary data references as a *set of sequence of atoms* instead of *set of sequence of locations*. Because of this connection, we denote the function that maps every data reference to a set of sequences of elements of $\mathit{DataRef}_{Atomic}$ by $\mathcal{D}_1$. The pseudocode in Fig. 13 shows how $\mathcal{D}_1$ is defined in its full generality.

Atomization versus Unification

It is worth noting that atomization imposes stricter requirements on the construction of the Equivalence DAG than does type analysis or unification. In the context of type analysis or unification, the two terms $\tau\oplus\tau$ and 2τ are completely equivalent. However, this is not true for atomization. The term $\tau\oplus\tau$ leads to twice as many atoms as the term 2τ, since it indicates that atomization should distinguish the first half of the array from the second half.

Consider the example in Fig 8. Unification of $(u{:}10)\oplus(v{:}10)$ and $10 \otimes (x_R{:}2)$ creates the bindings $u \mapsto 5 \otimes (x_R{:}2)$ and $v \mapsto 5 \otimes (x_R{:}2)$ and the unified terms can be represented by either $5 \otimes (x_R{:}2)\oplus 5 \otimes (x_R{:}2)$ or $10 \otimes (x_R{:}2)$. To perform atomization, however, it is important to choose $5 \otimes (x_R{:}2)\oplus 5 \otimes (x_R{:}2)$ as the representation of the unified terms. The algorithm presented in the appendix takes this stricter requirement into account.

Atomic Data References and Reaching Definitions

We note that two atomic data references $x\backslash i$ and $y\backslash j$ are disjoint whenever their atomic components differ, i.e., when $x \neq y$. Thus for the purposes of computing reaching definitions, two data references $x\backslash i$ and $y\backslash j$ can overlap if and only if $x = y$. However, when determining whether a definition of one atomic data reference *kills* (i.e., completely "covers") another, the multiplicity information comes into play: $x\backslash i$ kills $y\backslash j$ if and only if $x = y$ and $i = 1$. In other words, only a determinate reference can kill another reference. Thus for the purpose of computing reaching definitions, there is no reason to distinguish between different multiplicity values greater than 1. However, the full range of values will be useful in the sequel to establish certain formal properties of the atomization.

Correctness Properties

In this section, we formalize a notion of equivalence between a program and its transformed version, in which data references are replaced by corresponding atomic data references. We first define a function $\mathcal{S}_1$ that maps every statement $s \in \mathit{Stmt}$ to a set of statements $S \subseteq \mathit{Stmt}_{Atomic}$ where

$$\mathit{Stmt}_{Atomic} \quad ::= \quad \mathit{DataRef}_{Atomic} \leftarrow \mathit{DataRef}_{Atomic}$$

$$\begin{aligned}\mathcal{S}_1(d_1 \leftarrow d_2) = \\ \{\ \sigma_1(i) \leftarrow \sigma_2(i) \ \mid \ & \sigma_1 \in \mathcal{D}_1[d_1], \sigma_2 \in \mathcal{D}_1[d_2], \\ & 1 \leq i \leq \min(|\sigma_1|, |\sigma_2|)\ \}\end{aligned}$$

By ordering the set of locations identified by an atom a in ascending order, we get a sequence of locations which we denote $\mathcal{A}[a]$. Define a function $\mathcal{D}_2$ that maps every element of $\mathit{DataRef}_{Atomic}$ to a set of sequence of locations as follows:

$$\begin{aligned}\mathcal{D}_2[a\backslash n] = \{\ & \langle \sigma[s], \sigma[s+1], \cdots, \sigma[e]\rangle \ \mid \\ & 1 \leq i \leq n, \sigma = \mathcal{A}[a], \\ & s = (i-1) * (|\sigma|/n) + 1, e = i * (|\sigma|/n)\ \}\end{aligned}$$

Define function $\mathcal{S}_2$ mapping every element of $Stmt_{Atomic}$ to a set of ordered pairs of locations as follows:

$$\mathcal{S}_2(a_1 \leftarrow a_2) \;=\; \{\, (\sigma_1(i), \sigma_2(i)) \mid \sigma_1 \in \mathcal{D}_2[a_1], \sigma_2 \in \mathcal{D}_2[a_2], 1 \le i \le \min(|\sigma_1|, |\sigma_2|) \,\}$$

Observe that:

$$\mathcal{S}(d_1 \leftarrow d_2) = \bigcup_{(a_1,a_2) \in \mathcal{S}_1(d_1 \leftarrow d_2)} \mathcal{S}_2(a_1 \leftarrow a_2)$$

for every statement $d_1 \leftarrow d_2$ in the given program. Thus, we can think of our atomization algorithm as "decomposing" the semantic function $\mathcal{D}$ into $\mathcal{D}_1$ and $\mathcal{D}_2$. In particular, note that the abstract semantics $\mathcal{S}(s)$ of a statement s can be fully recovered from the atomized statement $\mathcal{S}_2(s)$. This formalizes the sense in which our atomization transformation is "lossless".

Note that $\mathcal{S}_1$ only models flow-insensitive program properties, since it does not distinguish between cases in which *one of* a set of possible assignments is executed, and cases in which *all of* a set of assignments are executed. It is straightforward to generalize $\mathcal{S}_1$ to yield a program transformation that correctly models flow-sensitive program properties.

7 Complexity Analysis

Let d denote the maximum number of atoms (as defined in Section 6) and arrays identified in a single aggregate. (For example, if we have a single aggregate whose atomization is $x_1 \oplus 5(x_2 \oplus x_3)$, then d is 4. Equivalently, we may think of d as the maximum size of the atomization-trees produced.) Let f denote the total number of atoms identified in the program and let s denote the total number of statements in the program. Our algorithm runs in time $O(sd \cdot \alpha(sd, f))$ in the worst case, where $\alpha(.,.)$ denotes the inverse Ackermann function.

8 Extensions

Variables of Unknown Length Our basic algorithm assumes that all variables have a statically-determined finite length. We can extend our algorithm to deal with variables of statically indeterminate length by representing them as variables of (potentially) infinite length. One interesting issue that comes up in this context is the need to do an "occurs check". Note that the algorithm presented in this paper binds variables only to terms of the same length. When all lengths are finite, this ensures that a variable can never be bound to a complex term containing the same variable. However, this is no longer true once variables of infinite lengths are allowed. We detect the creation of a cyclic term during unification, and replace it by a suitable "array" consisting of an unbounded number of elements. For example, unifying $(x : 2) \oplus (y : \infty)$ with $y : \infty$ results in binding y to $\infty \otimes (x : 2)$.

Pointer Analysis Our full algorithm incorporates a points-to algorithm similar to that of Steensgaard's[11, 12]. Since both our algorithm and the points-to algorithm are unification-style algorithms, it is straight forward to perform both the analyses in parallel, in a common framework. This is not only convenient, it turns out to be necessary since in the presence of both (implicit) aggregates and pointers, the points-to analysis depends on atomization information while the atomization algorithm requires points-to information.

The essential idea in this approach is as follows. For each pointer-valued variable p, we maintain a term τ_p describing the range of locations pointed to by p. Whenever two pointer-valued variables p and q need to be unified, the two corresponding pointed-to terms τ_p and τ_q are also unified. Note that *any* location (or range of locations) may potentially store a pointer value. Hence, we associate a points-to term τ_l with every leaf l of the Equivalence DAG. This effectively amounts to expanding our term language $\Gamma_{\mathcal{V}}$ to encode points-to information.

Using SSA Renaming to Improve Precision The flow-insensitive nature of our algorithm can introduce imprecision, especially when variables are used for completely different purposes in different parts of the program. The technique of *Static Single Assignment* [4] renaming can be used to improve the precision of the results produced by our algorithm.

One interesting issue that arises here is the interdependence between the atomization problem and the SSA renaming problem: our atomization algorithm can produce more precise results if applied after SSA renaming, while SSA renaming is easier to do after atomization since it does not have to deal with aggregates.

One possible solution to this issue is to run the atomization algorithm once, apply SSA renaming, and then run the atomization algorithm again to produce a more precise atomization. (Iterating any further will not improve the atomization results any.)

9 Related Work

A substantial body of work exists in the area of type inference, following [9]. While our algorithm belongs to this family, what distinguishes it is that it is aimed at low level languages, where aggregate structure has to be inferred from the way memory (locations) are accessed and used. The algorithm presented in this paper can be thought of as unification in the presence of an equational theory. Much previous work [8] has been done on unification in the presence of equational axioms (e.g, associativity) but we are unaware of previous work in this area for the specific equational theory that we are interested in.

Several other authors [11, 10, 7, 16] have explored the application of type inference based techniques to program maintenance and software reengineering as well as program analysis for imperative languages.

van Deursen and Moonen [16] present a type inference system for Cobol and describe its applications. What distinguishes our algorithm from theirs is the way we handle the unification of records. In their algorithm, the unification of two records causes the corresponding fields of the two records to be unified *only if the two records have the same structure*, i.e., only if they have the same number of fields, with corresponding fields having same length.

Our algorithm is also similar in some respects to a points-to algorithm presented by Steensgaard [11] which accommodates C-style `structs` and `unions`. The problem common to both our paper and Steensgaard's is the "unification" of two aggregates with *differing* structure. In our approach, the result of unifying two structures S_1 and S_2 is a structure that is *more refined* than both S_1 and S_2. For example, unifying $x{:}4 \oplus y{:}4 \oplus z{:}4$ with $a{:}4 \oplus b{:}2 \oplus c{:}6$ results in the structure $x{:}4 \oplus b{:}2 \oplus w{:}2 \oplus z{:}4$ with the additional bindings $a \mapsto x$, $y \mapsto b \oplus w$, $c \mapsto w \oplus z$. In Steensgaard's algorithm, on the other hand, the unification of S_1 and S_2 produces a structure that is *less refined* than both S_1 and S_2. In the above example, Steensgaard's algorithm will stop distinguishing between the fields y, z, b and c, and produce the unified structure $x{:}4 \oplus t{:}8$, with the a being bound to x, and y, z, b and c all being bound to t. Steensgaard[13], however, discusses the possibility of using more refined unification strategies.

As a result, our algorithm computes more precise results than Steensgaard's algorithm. Our algorithm was primarily designed to analyze legacy applications written in languages such as Cobol and PL/1, where variables are commonly used to store aggregate data without necessarily declaring the aggregate structure of the variables. We believe that in such a context our approach is preferable. However, whether our approach produces more precise results when applied to typical C or C++ applications remains to be seen.

O'Callahan and Jackson [10] use type inference to C programs to identify sets of variables that must share a common representation and outline various applications based on this.

10 Future Work

Future directions we wish to pursue include:

Other Notions of Atomization Does our atomization algorithm produce the optimal (i.e., the least refined) atomization? We believe that it does, with respect to one reasonable definition of atomization, though we have attempted no formal proof of this. However, if we relax the notion of an atom implicit in our algorithm, the atomization produced by our algorithm is not necessarily optimal. As an example, consider the program $x[1{:}10] \leftarrow y[1{:}10]; z[1{:}2] \leftarrow x[5{:}6]$. In this case, it is possible to generate the following atomized program $(x1, x2) \leftarrow (y1, y2); z1 \leftarrow x2$ where atom $x1$ denotes the *union* of the ranges $x[1{:}4]$ and $x[7{:}10]$, while atom $x2$ denotes the range $x[5{:}6]$. (The remaining atoms are defined in a corresponding manner.) However, our algorithm breaks up x into three atoms $x[1{:}4]$, $x[5{:}6]$, and $x[7{:}10]$, producing a more refined atomization than is necessary.

It is possible to take the atomization produced by our algorithm and to improve it further by applying an algorithm somewhat similar to the finite state minimization algorithm (grouping atoms that need not be distinguished from each other into equivalence classes). It would be more interesting to see if such an improvement can be integrated directly into our atomization algorithm.

Applications to Sparse Analysis The equivalence class partitioning of atoms produced by our algorithm can be used to construct (flow-insensitive) sparse evaluation representations for various analysis problems.

Disjoint Unions Cobol programs may use `REDEFINES` for two purposes: to define disjoint unions or to define multiple views of the same data. The inability to distinguish between these two usages forces our algorithm to handle `REDEFINES` conservatively. It would be worth developing analysis techniques to infer the use of disjoint unions so that they can be handled less conservatively.

More Sophisticated Type Systems One could extend the simple type framework introduced in Section 4 in various directions, including adding more complex constructors and incorporating inequality (or set) constraints [2, 5]. Such extensions might enable more precise treatment of data located at variable offsets, overloaded operators, and pointer arithmetic than are possible with our current approach.

References

[1] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] AIKEN, A., AND WIMMERS, E. Solving systems of set constraints. In *Symposium on Logic in Computer Science* (June 1992), pp. 329–340.

[3] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[4] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 451–490.

[5] FÄHNDRICH, M., AND AIKEN, A. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Symposium on Static Analysis* (September 1997), vol. 1302 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 114–126.

[6] FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (1987), 319–349.

[7] KAWABE, K., A. MATSUO, UEHARA, S., AND OGAWA, A. Variable classification technique for software maintenance and application to the year 2000 problem. In *Conference on Software Maintenance and Reengineering* (1998), P. Nesi and F. Lehner, Eds., IEEE Computer Society, pp. 44–50.

[8] KNIGHT, K. Unification: A multidisciplinary survey. *ACM Computing Surveys 21*, 1 (1989), 93–124.

[9] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[10] O'CALLAHAN, R., AND JACKSON, D. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering (ICSE'96)* (Boston, MA, May 1997), pp. 338–348.

[11] STEENSGAARD, B. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 1996 International Conference on Compiler Construction* (Linköping, Sweden, April 1996), vol. 1060 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 136–150.

[12] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[13] STEENSGAARD, B. Personal communication, Oct. 1998.

[14] TARJAN, R. E. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[15] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[16] VAN DEURSEN, A., AND MOONEN, L. Type inference for cobol systems. In *5th Working Conference on Reverse Engineering* (1998), IEEE Computer Society, pp. 220–230.

[17] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

Appendix

We now present a complete description of our algorithm in SML-like pseudo-code. The algorithm is based on the fast union-find data structure (*e.g.*, see [14, 3].) We assume that an implementation of the fast union-find data structure[14, 3] is available with the signature shown in Fig 11. The function `newvar` creates a new element not equivalent to any other element (*i.e.*, belonging to an equivalence class all by itself). The function `union` merges two equivalence classes into one, while the function `equivalent` indicates if two elements are equivalent or not. In addition, every equivalence class has a value associated with it, whose type is the parameter `'a` of the parametrized type `'a eqClass`. The value associated with an equivalence class can be retrieved and modified by the functions `findval` and `setval` respectively. The functions `newvar` and `union` take a parameter specifying the value to be associated with the newly created/merged equivalence class.

We also assume the existence of a semi-lattice `L` (of "types") with a join operator $\cup$.

In our implementation, we have a set of *term variables* (represented by the type $\mathtt{term}_{\mathtt{var}}$ in Fig 11), which are partitioned into a collection of equivalence classes (using the union-find data structure). Every equivalence class has an associated *value*, which has the type $\mathtt{term}_{\mathtt{value}}$.

The function $\lceil \mathtt{x} \rceil$ returns the length of a variable x. In an actual implementation it will be more efficient to store (cache) the length with the variable, rather than compute it every time it is needed.

The function $\boxed{\mathtt{v}}$ is a convenient wrapper for function `newvar`. It creates a new equivalence class with the initial value v, unless v denotes a one element array, in which case the array element itself is returned. The function `split(x,n)` adds a breakpoint to the

```
// An implementation of union-find with the
// following signature is assumed
type 'a eqClass
val newvar:  'a -> 'a eqClass
val union:  ('a eqClass * 'a eqClass * 'a) -> unit
val equivalent:  ('a eqClass * 'a eqClass) -> bool
val findval:  'a eqClass -> 'a
val setval:  ('a eqClass * 'a) -> unit

// A semi-lattice L of types is assumed, with a
// join/meet operator ∪
type L
val ∪ :  L * L -> L

datatype termvalue = atomic (L, int) |
                     termvar ⊕ termvar |
                     int ⊗ termvar
withtype termvar = termvalue eqClass

fun |x| =
   case (findval x) of
     atomic (c, l) => l |
     x1 ⊕ x2       => |x1| + |x2| |
     n ⊗ e         => n * |e|

fun v =
   case v of
     1 ⊗ e       => e |
     otherwise   => newvar v

fun split (x, n) =
   if (0 < n) and (n < |x|) then
     case (findval x) of
       atomic (c, l) =>
         setval(x,
            atomic (c, n) ⊕ atomic (c, l-n)
         ) |
       x1 ⊕ x2 =>
         if (n < |x1|) then
           split (x1,n)
         else
           split (x2,n-|x1|)
         fi |
       m ⊗ e =>
         let val p = max(1, n/|e|) in
           setval(x, p ⊗ e ⊕ (m-p) ⊗ e);
           split (x,n)
         end
   fi

fun refine (x, n) =
   case (findval x) of
     x1 ⊕ x2 => [x1, x2] |
     atomic (c, l) => split (x,n); refine (x,n) |
     m ⊗ e =>
         let val p = max(1, n/|e|) in
           setval(x, p ⊗ e ⊕ (m-p) ⊗ e);
           refine (x,n)
         end

fun update (x, c) =
   case (findval x) of
     atomic (c', l) =>
         setval(x,atomic (c ∪ c', l))) |
     x1 ⊕ x2 => update (x1, c); update (x2, c) |
     m ⊗ e => update (e, c)
```

Figure 11: Type definitions and auxiliary procedures.

```
fun unify (x,y) =
  if (not (equivalent(x,y))) then
    // Merge the two vars into one, setting the
    // value of the merged var to ...
    union(x, y,
      // ...  the join of the values of x and y
      case (findval x, findval y) of
        (atomic (c, l), _) => unifyatom(y,c) |
        (_, atomic (c, l)) => unifyatom(x,c) |
        (x1 ⊕ x2, _) => unifylist([x1,x2],[y]) |
        (_, y1 ⊕ y2) => unifylist([x],[y1,y2]) |
        (n1 ⊗ e1, n2 ⊗ e2) =>
              unifyarrays((n1,e1), (n2,e2))
    )
  fi;
  findval x // return the value associated with
            // the merged equivalence class

fun unifyatom(x,c) = update(x,c); findval x

fun unifylist(x1::r1, x2::r2) =
   if (|x1| = |x2|) then
     case (r1,r2) of
       ([],[]) => unify (x1,x2) |
       _ =>  unify (x1,x2) ⊕ unifylist(r1,r2)
   elsif (|x1| > |x2|) then
     unifylist( refine(x1,|x2|) @ r1, x2::r2)
   else /* (|x1| < |x2|) */
      unifylist( x1::r1, refine(x2,|x1|) @ r2)
   fi

fun unifyarrays( (n1,e1), (n2,e2) ) =
   let
     fun exp (t,1) = t |
         exp (t,i) = t ⊕ exp(t,i-1)
     val m = least-common-multiple(|e1|,|e2|)/|e1|
     val x1 = exp(e1, m)
     val x2 = exp(e2, m)
     val z = (n1* |e1| / |x1|) ⊗ x1
   in
     unify (x1, x2);
     z
   end

// The main procedures
datatype constraint =
           termvar ≅ termvar |
           termvar ⪰ L

fun processConstraint (x ≅ y) = unify (x,y) |
   processConstraint (x ⪰ c) = update(x,c)

fun solve listOfConstraints =
   apply processConstraint listOfConstraints
```

Figure 12: The unification and type analysis algorithms.

```
 // Assume some suitable way of generating
// names for new atoms.
type AtomId
val gensym:  unit -> AtomId

datatype AtomicReference = AtomId \ int
datatype AtomicTree =
       atom of (AtomId * int * int) |
       AtomicTree ⊕ AtomicTree |
       int ⊗ AtomicTree

// Assume a suitable implementation of ``program
// variables'' with the following signature
type PgmVar
val termVarOf:  PgmVar -> termvar
val setAtomicTree:  PgmVar * AtomicTree -> unit
val getAtomicTree:  PgmVar -> (AtomicTree option)

datatype DataRef =
       ProgVar of PgmVar |
       DataRef [ int :  int ] |
       DataRef \ int

fun flatten(x,mu) =
   case (findval x) of
     atomic (c, l) => atom (gensym(),l,mu) |
     y ⊕ z => (flatten(y,mu)) ⊕ (flatten(z,mu)) |
     i ⊗ y => i ⊗ (flatten(y,mu*i))

fun atomicTreeOf pgmvar =
   case (getAtomicTree pgmvar) of
     SOME atomicTree => atomicTree |
     NONE => (
       setAtomicTree (pgmvar, rightAssociate(
           (flatten (termVarOf pgmvar,1)) );
       atomicTreeOf (pgmvar);
     )

fun treesOf(ProgVar x) = { atomicTreeOf x } |
   treesOf(d [i:j]) =
       { subrange (t, i, j)  |  t ∈ treesOf(d) } |
   treesOf(d\n) = ∪t∈treesOf(d) breakup(t, |t|/n)

fun leaves(atom (a,l,mu),m) = [ a \ (mu/m) ] |
   leaves(x1 ⊕ x2,m) = leaves(x1,m) @ leaves(x2,m) |
   leaves(i ⊗ x,m) = leaves(x,m*i)

fun 𝒟1[d] = { leaves(t,1)  |  t ∈ (treesOf d) }

fun rightAssociate ((x1 ⊕ x2) ⊕ x3) =
     rightAssociate(x1 ⊕ (x2 ⊕ x3)) |
   rightAssociate (x1 ⊕ x2) =
     (rightAssociate x1) ⊕ (rightAssociate x2)
   rightAssociate (i ⊗ x) = i ⊗ (rightAssociate x)
   rightAssociate (atom (a,l,mu)) = atom (a,l,mu)

fun breakup (t,s) =
   if (|t| = s) then { t } else
   case t of
     x1 ⊕ x2 =>
       if (|x1| > s) then
         breakup(x1,s) ∪ breakup(x2,s)
       else
         { head(t,s) } ∪ breakup( tail(t,s+1), s)
       fi |
     i ⊗ x => breakup(x,s)

fun subrange (t, i, j) = head ( tail(t,i), j-i+1 )
```

Figure 13: The atomization algorithm (part 1).

```
 fun tail (t, 1) = t |
   tail (x1 ⊕ x2, i) = tail (x2, i - |x1|)

fun head (x1 ⊕ x2, i) =
       if (i > |x1|)
         then x1 ⊕ head (x2, i - |x1|)
         else x1|
   head (t, s) = t
```

Figure 14: The atomization algorithm (part 2).

DAG rooted at `x` after position `n`. The function `refine (x,n)` returns the children of a concatenation node `x`. If `x` is not a concatenation node, it is first converted into one by adding a breakpoint, preferably (but not necessarily) after position `n`. Note that every leaf of the Equivalence DAG has a value (belonging to semilattice `L`) associated with it. The function `update(x,c)` updates the value associated with every leaf of the DAG rooted at `x` by `c`.

The main unification algorithm as well as our type analysis algorithm appear in Fig. 12. The basic ideas behind the algorithm were explained earlier in Sections 3, 4, and 5. The four different `unify` functions implement the actual unification. The functions `solve` and `processConstraint` show how the Equivalence DAG can be constructed and how type analysis can be performed.

The atomization algorithm appears in Figs. 13 and 14 and was explained in Section 6. We assume the existence of a function `gensym` that can be used to generate new symbols to represent atoms. As explained earlier, every top level program variable is associated with a node (a term variable) in the Equivalence DAG. The function `termVarOf` is assumed to return this.

The first step in atomization is to take the DAG rooted as this node and flatten it to construct an "atomization tree". The function `flatten(x, mu)` shows how a $\texttt{term}_{\texttt{var}}$ `x` can be flattened into a `AtomicTree`. The second parameter `mu` is used to compute the *multiplicity* of the leaves in the atomic tree, where the multiplicity of a leaf in an atomic-tree is defined to be the product of the cardinalities of all arrays in the tree that contain the given leaf.

In order to simplify some of the other functions, the generated atomic-tree is "normalized" to be in right-associative form (by the function `rightAssociate`). We assume that the imperative functions `setAtomicTree` and `getAtomicTree` let us associate an atomic-tree with every program variable. (The function `atomicTreeOf` constructs the atomic-tree and associates it with the corresponding program variable.)

Once an atomic-tree has been constructed for a program variable `X`, any data-reference based on `X` can be mapped onto a set of subtrees of the atomic tree corresponding to `X`. The function `treesOf` does this. Please note that the input data-reference can not be an arbitrary one. It is assumed to be one of the data-references in the original program with respect to which the Equivalence DAG was constructed. (Hence, the Equivalence DAG and the atomic-tree are guaranteed to have "breakpoints" corresponding to the endpoints of data-reference `d`.)

Given an atomic reference `a\i`, let us refer to `i` as the denominator of the atomic reference. The function `leaves` shows how any subtree `S` of an atomic-tree `T` can be converted into a sequence of atomic references. This essentially is the sequence of leaves of the subtree combined with an appropriate denominator. The denominator of a leaf l is simply the multiplicity of l in the tree `T` divided by the multiplicity of l in the subtree `S`.