Tangible Program Histories

Todd A. Proebsting
Benjamin G. Zorn

May 2000

Microsoft Research
Technical Report MSR–TR–2000–54

Microsoft Research
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052 USA

Tangible Program Histories

Todd A. Proebsting Benjamin G. Zorn
Microsoft Research

May 26, 2000

Abstract

We propose a new language feature, a program history, that significantly reduces bookkeeping code in imperative programs. A history represents previous program state that is not explicitly recorded by the programmer. By reducing bookkeeping, programs are more convenient to write and less error-prone. Example program histories include a list that represents all the values previously assigned to a given variable, an accumulator that represents the sum of values assigned to a given variable, and a counter that represents the number of times a given loop has iterated. Many program histories can be implemented with low overhead.

1 Introduction

We propose a new programming language feature, a program history, that makes writing programs more convenient by eliminating tedious bookkeeping code in imperative programs. A program history captures aspects of *past* program state implicitly, thereby freeing the programmer from many error-prone and program-cluttering bookkeeping chores.

(Imperative programming languages have been criticized for requiring too much bookkeeping for many years [1]. Backus proposed the functional programming paradigm as an alternative that could eliminate such bookkeeping, but as yet, the functional paradigm has not achieved widespread use.)

We illustrate program histories with code to average the values in a list.

Without program histories

```
p = aList;
sum = 0;
count = 0;
while (p != NULL) {
    count += 1;
    sum += p->value;
    p = p->tail;
}
print("average %f\n", sum/count);
```

With program histories

```
p = aList;
while (p != NULL) {
    x = p->value;
    p = p->tail;
}
print("average %f\n", sum<x>/length<x>);
```

In the example, the notation <x> indicates the "history of x", which is the sequence of values assigned to x.[1] We envision many functions such as sum, length, min, max, etc. being defined for histories. By eliminating explicit bookkeeping code—like initializing and updating accumulator values—histories make programs shorter and simpler.

Program histories are motivated by the ease with which novice spreadsheet users can manipulate data. Spreadsheets make iteration implicit by having operations over rows or columns. Our program histories provide a similar functionality by explicitly exposing the history of values of a variable to the programmer for direct or aggregate manipulation (e.g., sum, average, etc. [6]).

Histories are not limited to logs of values assigned to a variable. Consider the problem of printing a comma-separated list of values. Normally, this requires explicitly maintaining a flag or counter to distinguish the first loop iteration from the rest.

Without program histories

```
p = aList;
firstTime = true;
while (p != NULL) {
    if (firstTime) {
        firstTime = false;
    } else {
        printf(", ");
    }
    printf("%d", p->value);
    p = p->tail;
}
```

With program histories

```
p = aList;
while (p != NULL) {
    if (count<while> != 1) {
        printf(", ");
    }
    printf("%d", p->value);
    p = p->tail;
}
```

The notation `count<while>` refers to the iteration count of the `while` loop. (More precisely, the number of times the loop has *begun* execution.) We use the name "`while`" as a predefined name that refers to the innermost `while` loop.

Histories provide programmers with convenient access to past program state. To maintain the same information explicitly—without histories—requires substantial effort. Even when programmers must maintain this historical program state (as with the examples above), the necessary bookkeeping code is a burden. Program histories liberate programmers from these chores.

We believe that program histories can have efficient implementations. For instance, if `average` is the only function computed on <x>, then a compiler could generate code that simply maintains the sum and count of the assignments to `x`—which is the optimal solution. Typically, with a small amount of analysis, using histories will add little or no additional overhead over the explicit approach.

Program histories are part of a our broader research goal, which is to improve programming language design in ways that make writing programs more convenient. We believe that programmer convenience is significantly under-emphasized in most language design. For many programs and programmers, the time to write a program is the most critical consideration. Programming environments (like Visual Basic's) and programming languages (like Perl and Tcl) are successful, in part, because they require less programming effort than alternatives. Languages like Ada support

[1] All uses of <> in this paper refer strictly to histories and are not to be interpreted as C++ templates.

programming-in-the-large, while our goal is to support programming-in-a-hurry. By eliminating bookkeeping and unnecessary code, program histories improve programmer convenience. Program histories complement another innovation for programmer convenience that we have proposed, which we call programming shorthands [7].

In the remainder of this paper, we more carefully introduce the concept of program histories and motivate their use with examples. Our goal is not to advocate a particular set of history features or a particular syntax for using them, but to explore their possible applications.

2 Name-bound Histories

Many histories are associated with programmer-named entities. Two obvious candidates for name-bound histories are program variables and functions, both of which exhibit time-varying behavior. Histories capture this behavior by exposing the sequence of values directly to the programmer. The history of a variable, `x`, denoted <x>, is the sequence of values that have been assigned to `x` since the start of program execution. Likewise, the history of a function, `f`, denoted <f>, is the sequence of values that `f` has returned.

The simplest way to access a history is to treat it as a list with indexing: <x>[i] represents the i^{th} element of <x> (i.e., the i^{th} value assigned to `x`). Similarly, length<x> would represent the number of values in the history.

The utility of histories increases with the number of built-in operations defined over them. Just as spreadsheets contain many functions that operate over rows or columns, histories benefit from a similar set of built-in operations.

Reduction operations that compute summaries of the history would include *sum*, *length*, *min*, *max*, *mean*, *trimmedMean*, *variance*, *mode*, etc. We illustrate the use of these operations with further examples.

To compute the maximum value of an element of an array:

Without program histories

```
int max = a[0];
for (i = 0; i < ARRAY_SIZE; i++) {
    if (max < a[i]) {
        max = a[i];
    }
}
printf("Max is %f\n", max);
```

With program histories

```
for (i = 0; i < ARRAY_SIZE; i++) {
    x = a[i];
}
printf("Max is %f\n", max<x>);
```

To print out a list of values read from input:

Without program histories

```
intVector list;
while (!eof(aFile)) {
    int x = read(aFile);
    list.append(x);
}
for (i = 0; i < list.length(); i++) {
    printf("%d %f", i, list[i]);
}
```

With program histories

```
while (!eof(aFile)) {
    x = read(aFile);
}
for (i = 0; i < length<x>; i++) {
    printf("%d %f", i, <x>[i]);
}
```

Some uses of histories do not lend themselves to side-by-side comparisons of code. For example, the code required to determine whether or not a variable has ever been assigned a value (i.e., if it is initialized), could require maintaining a flag at every possible assignment location. With histories, the code is simple:

```
printf("x %s been assigned", length<x> == 0 ? "has not" : "has");
```

Likewise, the history of outcomes of function calls is often of interest to programmers. For example, the following prints the number of warning messages issued by reporting the number of calls to the function `warning`.

```
printf("%d warning message(s) printed", length<warning>);
```

Note that to maintain this information explicitly would require tedious bookkeeping that maintained a global variable. That variable would have to be incremented either at all call sites, or within the routine itself.

To count the number of input records read while processing a file (assuming the function `gets` was used):

```
printf("%d lines of input read", length<gets>);
```

Often, programmers may be interested in less than a complete variable history, such as the values assigned to a variable inside a specific loop. As a result, we define a function on all histories, `reset` that resets the history to a null sequence.

3 Syntax-bound Histories

Some program entities with history do not have programmer-defined names. Locations within a program have simple histories: how many times has program execution visited this location? We use the operation `count` to refer to the number of times a program point has been executed.

We propose a simple scheme for naming program points based on user-defined labels. If a programmer wants to refer to the history of a control construct, they label the construct. For example, the following code demonstrates one way to compute how many times a portion of program guarded by a conditional has executed.

Without program histories

```
int counter = 0;
while (...) {
    if (test(x)) {
        counter += 1;
        x = f(x);
    }
}
printf("count: %d", counter);
```

With program histories

```
while (...) {
    if (test(x)) {
label:
        x = f(x);
    }
}
printf("count: %d", count<label>);
```

While this technique works, it is a bit cumbersome. To eliminate the need for extraneous labels, we leverage the syntax of the language by allowing the programmer to refer to parts of a compound statement. For example, the following program prints out how many times the `then` and `else` branches of the conditional are executed:

Without program histories

```
int thenCount = 0;
int elseCount = 0;
    if (x > 0) {
        thenCount += 1;
        y = dx + dy;
    } else {
        elseCount += 1;
        y = dx - dy;
    }
printf("then: %d, else: %d",
       thenCount,
       elseCount);
```

With program histories

```
posTest:
    if (x > 0) {
        y = dx + dy;
    } else {
        y = dx - dy;
    }
printf("then: %d, else: %d",
       count<posTest.then>,
       count<posTest.else>);
```

In the example above, the names `posTest.then` and `posTest.else` refer to the `then` and `else` components of the named `if` statement—each with its own history.

For another example, we simplify a common numerical calculation idiom. Numerical computations often limit loop iterations to avoid diverging computations. With a syntax-bound history—associated with the `while` loop—no explicit count is necessary:

Without program histories

```
    int limit = 0;
    x = f(0);
    do {
        limit += 1;
        x = f(x);
        if (limit > 10000) break;
    } while (abs(x - prev<x>) > epsilon);
```

With program histories

```
    x = f(0);
    do {
        x = f(x);
        if (count<while> > 10000) break;
    } while (abs(x - prev<x>) > epsilon);
```

Histories of program points provide an effective tool for program introspection, profiling, and debugging. With histories, it is very easy for a programmer to write a function profiler that reports the number of calls to every function in the program. Likewise, it is easy to determine how many objects were allocated (by counting calls to `malloc`, for example), or to determine whether the number of allocations equals the number of deallocations. By making these data easier to accumulate, histories enable programmers to exploit this information.

4 Generalizations of Histories

Program histories conveniently expose temporal aspects of program execution that are typically unavailable to the programmer. In this section, we generalize the histories described in the previous sections. We note that in many cases, the advantages one gets from the proposed generalizations may not clearly outweigh the increase in complexity and implementation overhead they impose. We, therefore, do not specifically advocate any of these generalizations, but suggest them as interesting opportunities for future work.

4.1 Generalizing the Notion of History

Section 2 defines the history of a variable to be the sequence of values that are assigned to it. We can generalize this idea by considering the history to be the sequence of *definitions*. Specifically, each element in the sequence of definitions becomes a tuple containing the following information:

- *value:* As before, the actual value assigned.
- *location:* The program location where the assignment occurred.
- *timestamp:* The time when the assignment is made.

The `location` information could be used to answer such questions as "What value was assigned to x the last time it was updated at this source coordinate?" or "How many times was variable x updated at source coordinate y?". The `timestamp` information—measured in CPU cycles, elapsed time, absolute time, etc.—provides valuable profiling information.

4.2 Generalizing Function Call Histories

Function call histories may be similarly generalized. Function call tracing, in which function arguments and return values are displayed, is a feature commonly provided by debuggers, especially for interpreted languages such as Lisp [4]. Function histories can be extended to include additional information about each call, including the arguments, call site, time of the call, etc.

4.3 Computing Queries on Histories

Given that histories represent a collection, it is natural to consider mechanisms for querying and filtering the collection. For example, one may be interested in knowing if the value of a variable was ever less than zero. If histories include information about where the definition occurred, one may be interested in only those assignments that occurred at a particular source coordinate (e.g., one may wonder "Did **this** assignment ever result in x being less than zero?").

4.4 Associating Histories with Arbitrary Expressions

Programmers may not only be interested in the value of a specific variable, but in the history of how two or more variables are related. For example, one might express the minimum value of the expression $x - y$ over a computation using the syntax `min<x-y>`. The semantics of such a history would be that the history is extended and a new sequence value added whenever the value of either x or y change.

4.5 Associating Histories with Call Sites

Individual call sites can also have program histories. Suppose that you want to search a list for a value and report how many comparisons were required before the value was found:

```
p = aList;
while (p != NULL) {
    x = p.head();
  match:
    found = equal(p.head, key);
    if (found) break;
    p = p.tail();
}
print("searching required %d probes\n", length<match:equal>);
```

The history `<match:equal>` is the sequence of values returned by `equal` at location `match`. This history limits the history to a call-site, rather than the function's global history.

4.6 Reference Histories

One can imagine a history that represents the sequence of values *read* from a particular variable. The length of this sequence gives profiling feedback regarding how hot this variable is.

4.7 Histories of Dynamically Allocated Objects

The histories of heap-allocated objects are also interesting. Naming heap objects requires care, but is not difficult. Does `<ptr>` refer to the history of `ptr` or the history of what `ptr` refers to on the heap? Histories of heap-allocated objects would have many uses. They could be used to profile heap object usage. Programmers might also be interested in knowing what heap objects are allocated and then never referenced at all—a surprisingly frequent situation (e.g., see Seidl's data in [8]).

5 Implementation

Maintaining a list that represents the complete history of all values ever assigned to every variable is impractical. Even maintaining a list for a single variable can be impractical if that variable is assigned many values. (One can easily find examples of loops that would generate enormous histories.) Fortunately, useful operations on program histories, such as computing the average over a numeric variable's history, can be computed by maintaining a simple accumulator. In fact, all

"reduction" operations can be computed with a single accumulator (e.g., sum, count, max, etc.). Implementing reduction operations requires only updating the accumulator at all assignments to the given variable. The overhead is no greater than the overhead of the explicit computation.

For more general kinds of histories and operations on them, many implementation questions remain open research issues. In future work, we intend to consider these problems in greater detail to better understand their implementation issues.

6 Related Work

Program histories cannot easily be reproduced using functional abstraction because variable updates, for example, are typically scattered widely throughout a program text. Similarly, object-oriented techniques provide little help towards implicitly providing the benefits of program histories.

Active variables are a language mechanism that allows procedures to be invoked whenever a variable is read or written [2]. As such, the active variable mechanism could be used to implement some of the aspects of programming histories that we have described. Program histories differ from active variables in that they are a higher-level language feature intended for the specific purpose of making programming more convenient by providing *implicit* computation. In addition, program histories generalize beyond variables to procedure function calls and syntactic program entities. SNOBOL's trapped variables provide benefits similar to those of active variables, but do not anticipate the benefits of program histories [5].

Several programming languages expose variable history directly as a consequence of their design. To date, the languages in this class have been side-effect free languages that use histories as a way to integrate the imperative nature of variable update in loops in a side-effect free way. Languages in this class include Szymanski's Equational Programming Language (EPL) [9] and Derby's EQ [3]. While these languages provide mechanisms akin to program histories, the emphasis differs from our emphasis on improving programmer convenience in imperative programming languages. Furthermore, the languages do not attempt to generalize variable histories to syntactic constructs.

7 Summary

We have proposed program histories as a language feature for reducing bookkeeping in programs. Program histories expose temporal aspects of program execution that would otherwise require explicit bookkeeping code. We have presented simple examples of program histories that make programming more convenient and have simple and efficient implementations. We discuss how the simple examples of program histories might generalize.

Acknowledgments

Chris Fraser and David Hanson provided helpful feedback on these ideas.

References

[1] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] Daniel G. Bobrow. *The LOOPS Manual*. Xerox, Inc., Palo Alto, California, 1983.

[3] Thomas Derby, Robert Schnabel, and Benjamin Zorn. A new language design for prototyping numerical computation. *Scientific Programming*, 5:279–300, 1996.

[4] Franz Inc. *Extended Common Lisp Reference Manual*, 1986.

[5] David R. Hanson. Variable associations in SNOBOL4. *Software—Practice and Experience*, 6(2):245–254, April 1976.

[6] Clayton Lewis and Gary M. Olson. Can principles of cognition lower the barriers to programming? In *Empirical Studies of Programmers: Second Workshop*, pages 248–263, 1987.

[7] Todd A. Proebsting and Benjamin G. Zorn. Programming shorthands. Technical Report MSR-TR-2000-03, Microsoft Research, Redmond, WA, January 2000.

[8] Matthew L. Seidl and Benjamin G. Zorn. Predicting references to dynamically allocated objects. Technical Report CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, CO, January 1997.

[9] Boleslaw Szymanski. *EPL—Parallel Programming with Recurrent Equations*, pages 51–104. ACM Press, 1991.