Dynamic Variable Comparability Analysis for C and C++ Programs

Philip J. Guo PGBOVINE@CSAIL.MIT.EDU
Stephen McCamant SMCC@CSAIL.MIT.EDU
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge MA, 02139 USA

1. Introduction

Languages like C and C++ provide programmers with only a few basic types (e.g., `int`, `float`). Programmers often use these types to hold semantically unrelated values, so types typically capture only a portion of the programmer's intent. For example, a programmer may use the `int` type to represent array indices, sensor measurements, the current time, or other unrelated quantities. `pair<int,int>` can represent the coordinates of a point, a quotient and remainder returned from a division procedure, etc. The use of a single programming language representation type for these conceptually distinct values obscures the differences among the values.

```
int main() {
  int year = 2005;
  int winterDays = 58;
  int summerDays = 307;
  compute(year, winterDays, summerDays);
  return 0;
}

int compute(int yr, int d1, int d2) {
  if (yr % 4)
    return d1 + d2;
  else
    return d1 + d2 + 1;
}
```

In the program above, the three variables in `main` all have the same type, `int`, but two of them hold related quantities (numbers of days), as can be determined by the fact that they interact when the program adds them, whereas the other contains a conceptually distinct quantity (a year). `day` and `year`, the *abstract types* that the programmer most likely intended to convey in the program, are both represented as `int`.

A variable comparability analysis aims to automatically infer when sets of variables with the same representation type actually belong to the same abstract type. Sets of variables with the same abstract type are said to be *comparable*. This analysis could make the code's intention clearer, prevent errors, ease understanding, and assist automated program analysis tools. In the past, it has been performed statically using type inference, but we propose to perform a dynamic comparability analysis by observing interactions of values at run-time. We believe that a dynamic analysis can yield more precise results with greater scalability, given an execution which provides adequate coverage. We have implemented a tool called DynComp which performs this analysis for C and C++ programs.

2. Application to Invariant Detection

One specific application of a comparability analysis is to improve the performance and results of the Daikon invariant detector (Ernst, 2000). Daikon analyzes program value traces to infer properties that hold over all observed executions. Without comparability information, Daikon attempts to infer invariants over all sets of variables with the same representation type, which is expensive and likely to produce invariants that are not meaningful. For the above example, Daikon may state that `winterDays < year`. While this invariant is true, it is most likely not meaningful because the two variables belong to different abstract types (they are not comparable). Comparability information indicates which pairs of variables should be analyzed for potential invariants, which both improves Daikon's performance and helps it produce more meaningful results.

3. Static Analysis: Lackwit

The Lackwit tool (O'Callahan & Jackson, 1997) performs a static source code analysis on C programs to determine when two variables have the same abstract type. It performs type inference to give two variables the same abstract type if their values may interact at any time during execution via a program operation such as + or =. Because it does not actually execute the program, it must make conservative estimates regarding whether variables may interact, which may lead to imprecise results with fewer abstract types than actually present in the program. Also, though it is sound with respect to a large subset of C, this subset does not cover all the features used in real programs: it may miss interactions that result from some kinds of pointer arithmetic, and it does not track control flow through function pointers.

4. Proposed Dynamic Analysis

We propose a dynamic approach for computing whether two variables are comparable at program points such as procedure entries and exits. The analysis conceptually computes abstract types for values, then converts the information into sets of comparable variables at each program point (called *comparability sets*). It consists of a value analysis which occurs throughout execution and a variable analysis which occurs during each program point.

The value analysis maintains, for each value in memory and registers, a tag representing its abstract type. It associates a fresh abstract type with each new value created during execution. For a primitive representation type such as `int`, new values are instances of literals and values read from a file. Only values of primitive types receive tags; structs and arrays are treated as collections of primitive types. Two values have the same abstract type if they interact by being arguments to the same program operation such as `+` or `=`. This is a transitive notion; in the code `a+b; b+c`, the values of `a` and `c` have the same abstract type. Each program operation on two values unifies their abstract types, using an efficient union-find data structure, and gives the result the same abstract type.

The variable analysis is intended to report, for any pair of variables at a given program point, whether those variables ever held values of the same abstract type at that program point. The abstract type information that is maintained for values must be converted into abstract types for variables each time a program point is executed. In order to accommodate this, the analysis keeps a second variable-based set of abstract type information (independently for each program point) and merges the value-based information into that data structure at each execution of the program point. We are currently experimenting with several algorithms for this operation, each with different degrees of precision versus performance.

4.1 Implementation: DynComp

We have implemented a tool called DynComp for performing dynamic comparability analysis of C and C++ programs. It is built upon a framework based on dynamic binary instrumentation using Valgrind (Nethercote & Seward, 2003). It maintains a numeric tag along with each byte of memory and each register which represents the abstract type of the value stored in that location. The value analysis is performed by instrumenting every machine instruction in which values interact to unify their tags in the union-find data structure. The variable analysis is performed by pausing the program's normal execution during program points, reading the tags of the values held by relevant variables, and translating the abstract types represented by these tags to the abstract types of the variables.

4.2 Advantages of Our Dynamic Approach

Our dynamic approach has the potential to produce more precise results than static analysis because it need not apply approximations of run-time behavior but can observe actual behavior. Whereas a static tool must infer whether two variables could ever possibly interact and become comparable on any possible execution (usually by making conservative estimates), our dynamic analysis (given a test suite with adequate coverage) can tell exactly whether the two variables are comparable during actual executions.

Furthermore, our use of dynamic binary instrumentation results in a tool that can be more robust than Lackwit's source-based static approach because it only needs to deal with value interactions in memory and registers, which have relatively simple semantics. We do not need to handle complex source code constructs (such as pointer arithmetic, function pointers, or type casts) or analyze the source of or make hand-written summaries for library code (which often includes difficult-to-analyze constructs), requirements which are often difficult to implement robustly.

5. Experimental Results and Future Work

DynComp has been tested to work on moderately-sized C and C++ programs (around 10,000 lines of code). In quantitative evaluations, DynComp usually produces smaller comparability sets than Lackwit and allows Daikon to run faster and generate fewer invariants. In qualitative evaluations, the sets that DynComp produces more closely match programmer-intended *abstract types* because it does not have to make approximations about run-time behavior.

DynComp's scalability is currently limited by the memory overhead of maintaining tags, but we are currently working on garbage collection and various optimizations to overcome this limitation. In the meantime, another member of our research group is working on a Java implementation of dynamic comparability analysis.

References

Ernst, M. D. (2000). *Dynamically discovering likely program invariants*. Doctoral dissertation, University of Washington Department of Computer Science and Engineering, Seattle, Washington.

Nethercote, N., & Seward, J. (2003). Valgrind: A program supervision framework. *Proceedings of the Third Workshop on Runtime Verification*. Boulder, Colorado, USA.

O'Callahan, R., & Jackson, D. (1997). Lackwit: A program understanding tool based on type inference. *Proceedings of the 19th International Conference on Software Engineering* (pp. 338–348). Boston, MA.