Dimentia: Detecting Logic Errors with Dimensional Analysis

Goran Žužić and Sol Boucher

May 3, 2016

Abstract

We present Dimentia*, a system that uses static analysis to detect possible program logic errors related to dimensionality mismatches. Unlike prior work, the system is completely automatic: it doesn't require modifying the software to be tested. We showcase its usefulness for finding logic errors by presenting a small case study using examples from programming competitions.*

1 Introduction

Logic errors are a common type of bug that can be notoriously hard to catch. While there is no hope of detecting all logical mistakes in source code, one can try to design specialized software to detect a specific class of probable bug. To this end, we propose a static bug detection technique that uses dimensional analysis to flag possible programmer mistakes. We propose one such static bug detection technique that tracks the manner in which variables are used throughout a compilation unit.

Examine the C++ source code in Listing 1. If we remove the statement `/ sq_dist(...)` marked in red, the code contains a dimensional bug. To see the bug, it is important to know that the variables `v`, `g` and `pts[]` hold 2-D points. Let's denote the dimension of the unit distance of this 2-D plane as E. It can be inferred that the dimension of `dot_prod` is E^2. Without the statement in red, the dimension of `alpha` is also E^2. But this implies that `pts[i].x + alpha*v.x` will be adding together variables of dimension E and E^3, a clear dimensional bug. The proposed method vows to catch such bugs.

Listing 1: Example of a dimensional bug

```
double dot_prod = (g.x-pts[i].x)*v.x + (g.y-pts[i].y)*v.y;
double alpha = dot_prod / sq_dist(v, {0, 0});
if (...) {
  Pt cand = {pts[i].x + alpha*v.x, pts[i].y + alpha*v.y};
  ...
}
```

The example from Listing 1 comes from a competitive programming event, and although it has been edited for clarity, it illustrates a mistake that one of the authors actually made during the contest.

2 Related work

Checking dimensional consistency is a classical topic in software analysis. Intermixing physical units with native program code and utilizing dimensionality checking appeared in Ada [3] and Pascal [1, 2]. These approaches share a common theme: they first define a language extension used to associate a variable with an expressive physical unit, then use rule-based inference to catch dimensionality bugs.

For instance, [3] builds a package allowing the programmer to declare general classes of data (e.g. `DISTANCE` and `TIME`) as well as specific units (e.g. `CM` and `SEC`). Variables and constants can then be annotated with these types of metadata, respectively. The author's package requires the programmer to define data classes' multiplicative relationships, which is done by supplying matrices of coefficients. When it encounters an expression in the program, the package retrieves these matrices by looking up the operands' classes by units. It then verifies that the program's computations are consistent with this dimensionality information.

Although past work in dimensional analysis offers rich semantic checking of programs thanks to its perfect understanding of variables' data classes, this comes with the cost of requiring significant programmer effort. Data classes must be explicitly defined and related, and every non-dimensionless variable must be explicitly related to a unit. We observe that, while having such detailed information enables very accurate verification, it is possible to perform dimensional analysis without any additional information about the program or its variables.

This approach shifts the focus from one of directly detecting *correctness* violations to one of finding *consistency* violations: it lacks the information to identify semantic errors when they appear in isolation, but given a larger program, is able to identify conflicting uses of values. Moreover, because no additional labor is imposed on the programmer, such a system is more likely to be used, especially in situations such as competitive programming, where rapid detection of bugs is important but development time that is paramount.

3 Our approach

In order to associate a dimension with a variable (or more generally, a semantic type), we define the scalar *degree* of a semantic type as the sum of exponents on its units; for instance, lengths' (often expressed in meters) degree would be 1, areas' (often expressed in meters squared) would be 2 and accelerations' (often expressed in meters per seconds square, $[ms^{-2}]$) would be -1.

We write $°(\mathtt{x}) = d$ to indicate that the type associated with the source program variable (or temporary register) `x` has degree d, so if `l` were a length and `a` were an area, we might write $°(\mathtt{l}) = \mathtt{1}$ and $°(\mathtt{a}) = \mathtt{2}$. In processing a program that provides no information about data classes, we aim to deduce the degrees of variables based on their actual use in the program.

This is done by establishing consistency rules that must be satisfied when a mathematical operation is applied between variables. For instance, an instruction $z := x + y$ only makes sense if $°\mathtt{z} =° \mathtt{x} =° \mathtt{y}$. The implemented list of rules is shown in Table 1.

Instruction	**Equation generated**
$z := x + y$	$°(z) = °(x) = °(y)$
$z := x - y$	$°(z) = °(x) = °(y)$
$z := x \cdot y$	$°(z) = °(x) + °(y)$
$z := x/y$	$°(z) = °(x) - °(y)$
$x = y$	$°(x) = °(y)$
$x < y$	$°(x) = °(y)$

Table 1: Rules for degree consistency

Note that each of rules is essentially a linear equation between the arguments' degrees. At the end of instruction processing, we are left with a single matrix consisting the system of such equations, the remaining task is to find out whether each variable was used in an internally consistent manner throughout the program.

Denoting the matrix by A and the list of degrees by x, our consistency rules furnish a linear system $Ax = 0$. Note that the system is homogeneous, so $x = 0$ is always a valid (consistent) solution. In order to identify a potential bug, we define a *dimensionless type*, a type that cannot have a non-zero degree without violating the constraints. For instance, a instruction of the form $z = z \cdot a$ would imply that the type of a is dimensionless (must have degree 0). The presence of variables with dimensionless type indicates there is no consistent way to assign a nontrivial unit to them, and by extension that there may be a logic error.

While this approach can help us find potentially inconsistent variables, it doesn't directly allow us to pinpoint the locations of possible bugs in the program. To accomplish this, we flag a source code line as potentially inconsistent if removing it from the program would reduce the number of dimensionless types.

3.1 Implementation

We created the tool *Dimentia* that follows the approach described in Section 3. It is implemented as an LLVM 3.7 analysis pass; as such, it is invoked using the framework's `opt` optimization utility and can accept either LLVM IR assembly code or an LLVM bitcode object. The decision to implement as a pass rather than as part of the language frontend was made to keep it language-agnostic. In fact, since our method doesn't require programmer annotations, it should be possible to run it unmodified on programs compiled by any LLVM frontend, regardless of source language (although we've only tested with Clang). However, since our aim is to detect errors in the source program, we must be able relate registers back to their high-level variables and map instructions to the source lines that generated them; for this, we use debugging annotations generated by the frontend, and require that the user has compiled in debug mode.

Dimentia actually consists of two separate passes, but the first is solely responsible for processing debugging information: it builds lookup tables that bidirectionally associate registers with source program variables. To do this, it walks the `globals` section of the `llvm.dbg.cu` compilation unit debugging metadata, then searches for `llvm.dbg.declare` and `llvm.dbg.value` intrinsic instructions in the body of the program. After the pass completes, these lookup tables are provided to a second pass that's responsible for the actual dimensional analysis.

The bulk of the work done by this pass is in examining all of the program's instructions and generating corresponding degree equations. At the beginning of the program, a column is allocated for each source program variable, as enumerated by the first pass. Thereafter, as each instruction is processed, entries for operands that map directly to source program variables are placed directly in the columns for those variables in order to keep the size of the matrices down. Whenever an instruction involves a temporary register, the pass checks whether a column has been allocated for it. If so, the column is reused; otherwise, a new one is created and this action recorded so future instructions are aware of it. (Because LLVM shares `Constant` objects globally, we skip generating equations that include compile-time constants in order to avoid reporting false positives resulting from phantom associations between degrees.) Each equation is stored in a variable-width array because the presence of temporary registers means that the number of columns isn't known until all instructions have been processed.

While this is all that's required to handle scalar types, vector types pose challenges of their own. In order to accommodate arrays and structured types, we handle the `getelementptr`, `load`, and `store` instructions specially. The former instruction creates a matrix column (and hence, common degree) to be shared among accesses of that *class* of equivalent locations. For arrays, we make the assumption that the same sort of data is stored throughout, so we define their class as all elements of that same array. For structured types, the semantic relationship is cross-sectional, and we define the class of a particular element as that same element of any instance of the same struct. These column redirections are stored in a separate table that is consulted only when processing the pointer operand of `load` and `store` instructions.

With all instructions processed and the individual equations produced, we resize each by adding zeroes at the end until the system forms a non-ragged matrix. We're now ready to search for dimensionless variables.

If we denote the matrix as A and the degree vector as x, we have a homogeneous linear system $Ax = 0$. Thus, every element of the null space $N(A)$ is a valid assignment to the degree vector x. Hence in order to find a variable at index i that is dimensionless, we have to check whether every vector of the null space has a 0 at the index i. Note that it is sufficient to check any spanning set of vectors of the null space for that condition.

We are left with finding a spanning set for the null space, which is easy to obtain via singular value decomposition (SVD). Rather than writing our own SVD implementation, we used the one available in Lapack 3.

The final ingredient in our approach, identification of potentially non-consistent source lines, is done in a straightforward manner: for each source line, we remove all the linear equations generated by that source line and re-run the analysis. If the number of dimensionless types decreases, we flag the line, with one exception: in order to cut down on the number of false positives, we ignore lines that eliminate temporaries from the dimensionless set without affecting any source program variables.

We then use the approach described above, combined with a final consultation of the program's debugging annotations, to identify potentially dimensionless variables and problematic lines and trace them back to source program locations that should be presented to the user.

3.2 Limitations

Dimentia presently suffers from a few noteworthy limitations.

- **Unhandled arithmetic operations.** We don't generate equations for modulus (remainder) operations or bitwise arithmetic because it's non-obvious what effect these operations have on variables' degree relationships.
- **Pointers and pointer arithmetic.** While we can handle memory accesses directly into arrays and structures, we don't perform special tracking of pointers or their modification. For this reason, performing pointer arithmetic prior to a dereference may produce unexpected results.
- **Functions, arguments, and return values.** We don't presently track the associations between call sites and functions, arguments, and parameters, or return statements and call evaluations. This means that, although we can detect dimensional errors within individual functions, we cannot tell whether a function is misused by client code.

- **Platform specificity.** The current implementation uses LLVM object addresses as hash codes as part of the process of mapping a value to the matrix column storing information about its degree. While this model works without hackery on source variables, temporaries, and arrays—we simply use the address of the corresponding `DIVariable`, `Value`, or `DIVariable` of the entire array, respectively—it requires some tweaking in order to accommodate the cross-sectional lookups necessary for associating the fields of different struct instances. To handle this case, we start with the `StructType` object describing the global definition of that struct type; however, since there are no global LLVM objects corresponding to the individual positions *within* the struct, we cannot simply use an address. Instead, we pack the offset into the struct into the top sixteen bits of the address, which works because x86-64 machines have a 64-bit word size but only a 48-bit address bus. Of course, other platforms may use these address bits, and in such a case, this approach could result in hash and equality collisions.

- **Oversized structure offsets.** As stated above, we pack structure offsets into 16 bits, which is likely to be too small for particularly large types.

- **Arrays within structures.** In order to find the offset into a struct for the purpose of associating an access with the appropriate degree class, we need the offset into that structure to be a compile-time constant. Unfortunately, when an array appears within a structure, this is not the case, so we're unable to compute the hash codes for such accesses. While it would be possible to compute the partial constants of accesses in order to collapse all accesses within the array element down, this would complicate the code for handling memory operations, and is currently not implemented.

4 Evaluation

We evaluated the tool on several programs that were written for programming competitions. Some of the examples were hand-picked; others were randomly selected. Each code sample had between 53 and 200 lines of C++ or C code. The results are depicted in Table 2.

Name	SLOC (IR)	Runtime	Flagged variables	Flagged lines	Known bugs
`quadrilateral.c`	107 (561)	0.040s	4	3	Yes
`proposal-full.cpp`	88 (4522)	4.622s	10	4	Yes
`cf-1.cpp`	53 (3059)	0.139s	0	0	No
`cf-2.cpp`	77 (3896)	1.175s	1	1	No
`cf-3.cpp`	200 (11350)	4m14s	0	0	No

Table 2: Evaluation results

Here we present a small case study of the various cases we encountered in the code.

Our first example is a 107 SLOC hand-written program that does various area computation. At one point, we purposely made an error in the error calculation code. Our tool flags three lines in total, two of which are depicted in the following listing. There is no way to actually decide which line is the erroneous one, as the only thing our tool can infer is their inconsistency. *Dimentia* makes it easy to find this particular bug thanks to the set of lines it outputs.

```
if(shape == SHAPE_RECTANGLE) {
  area = param[0] * param[1];
} else { // shape == SHAPE_SQUARE
  area = param[0] + param[0];
}
```

In the introduction we had an (edited) code excerpt from `proposal-full.cpp`, a real programming contest solution with a bug. Its full version is shown below. Variables `pts[]` and `gmid` are 2-D points with fields `.x` and `.y`. Our tool highlights four lines in total, two of which are shown in red below. (The other two are false positives related to the propagation of dimensionless variables throughout the code.) The two flagged lines indicate that if we assume that points have a degree d, then `alpha` should have degree $2d$ and `pts[pi].x + alpha*v.x` would be adding together variables of degree d and $3d$, implying $d = 0$. In this case, our tool's output is nearly perfect and would immensely help us find the bug.

```
Pt v = {pts[pj].x - pts[pi].x, pts[pj].y - pts[pi].y};
double alpha = (gmid.x-pts[pi].x)*v.x + (gmid.y-pts[pi].y)*v.y;
if (alpha >= -eps && alpha <= 1+eps) {
  Pt cand = {pts[pi].x + alpha*v.x, pts[pi].y + alpha*v.y};
  double sum = 0;
  REP(gi, ng) sum += get_sq_dist(gens[gi], cand);
  ans = min(ans, sum);
}
```

In the source file **cf-2.cpp** we encounter the next code segment:

```
if (ccw(pts[i], pts[k], pts[j]) > 0) {
  dp[i][j] += dp[i][k] * dp[k][j];
  dp[i][j] %= mod;
}
```

The line in red is flagged. The `dp` array represents the number of objects that that can be constructed with some kind of recursive structure. Since this value is calculated by multiplying the number of ways its substructures can be constructed, it is definitely dimensionless (can not have a unit associated with it). The reported variable/line is dimensionless, but it does not correspond to a bug; we expect other instances of dynamic programming problems to trigger similar false positives.

5 Conclusion

Our experience with *Dimentia* demonstrates that this dimensional inference technique works surprisingly well. It very effectively discovers most dimensional bugs and the reporting makes it very easy to find and fix the cause of such errors. Moreover, after changing our handling of `Constants` and equations that only eliminate dimensionless temporaries, the only false positives we've encountered have been in programs that make use of dynamic programming.

6 Future work

In order to create a tool that would be suitable for use in production, the limitations discussed earlier would need to be addressed. It may also be desirable to be able to check across compilation units: especially if support for function argument and return value checking were implemented, this would allow inference of the relationship between these quantities even when functions were implemented in a different module of a codebase than they were used. Another barrier to adoption is the system's runtime on large programs or those with multiple template specializations. Fixing this may require a smarter method of pinpointing the lines responsible for dimensionality bugs (instead of the current approach of iteratively removing equations).

7 Distribution of credit

Goran 50%

Sol 50%

References

[1] A. Dreiheller, B. Mohr, and M. Moerschbacher. Programming pascal with physical units. *ACM Sigplan Notices*, 21(12):114–123, 1986.

[2] N. Gehani. Units of measure as a data attribute. *Computer Languages*, 2(3):93–111, 1977.

[3] P. N. Hilfinger. An ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):189–203, 1988.