Annotation-less Unit Type Inference for C

Philip Guo and Stephen McCamant
Final Project, 6.883: Program Analysis

December 14, 2005

1 Introduction

Types in programming languages are crucial for catching errors at compile-time. Similarly, in scientific applications, the units system forms a type discipline because a correct equation must necessarily include terms with units that are equivalent on both sides. Many scientific and numerically-oriented programs are written in C, but the language provides no support for attaching units to variables. Thus, programmers cannot easily tell whether the units for variables in their programs are consistent. We propose to create an analysis that can infer the unit types of variables in C programs such that the units type discipline holds across all expressions in the program. Unlike previous work which relies on the user to annotate some types and then checks the rest for consistency, we propose to infer types with no user annotations at all. This completely automatic analysis produces a most-general system of units for a program, which include any consistent unit system the programmer may have intended as a special case. This most-general solution can then be specialized interactively by the programmer to give human-readable names to units, a process which requires much less programmer interaction than specifying each variable's units one by one. Our analysis can be used by programmers to find bugs as indicated by the inferred unit types not matching up with their intuition and as a starting point for annotating their code with units using an annotation framework.

The correct use of physical units and dimensions is an important aspect of correctness for many computer programs. On one hand, many programming errors lead to programs whose results do not have the expected units, analogously to the common experience of students in introductory physics courses. For instance, an experiment performed by Brown [Bro01] showed that checking the units in a short procedure written to calculate a function used in particle physics revealed three separate errors. Conversely, programs with unit errors can harbor subtle bugs. To cite a particularly costly error, an on-ground system used in the navigation of the NASA Mars Climate Orbiter spacecraft failed to convert between pound-seconds and newton-seconds in calculating the impulse produced by thruster firings, due to a programmer error when updating a program used for a previous spacecraft to include the specification of a new model of thruster [EJC01]. This root cause, along with inadequate testing, operational failures, and bad luck, eventually led to the loss of the spacecraft as it was destroyed in the Mars atmosphere on September 23rd, 1999 [Mar99]. As these examples indicate, more careful attention to physical units could improve software quality, and because the rules governing unit correctness are simple, much of such checking can be performed automatically.

In fact, extensions exist for many languages allowing programmers to specify the units of variables and constants so that they can be checked by a compiler. However, though many of the potential benefits of unit annotations would be realized in the maintenance of existing systems, adding unit annotations by hand to large existing programs would be very expensive. Instead of requiring a developer to specify each unit type individually, we believe that a better approach is to automatically infer a general set of unit types based only on the assumption that a program's use of units is consistent. Such inferred types could be useful for many kinds of automatic checking, such as warning a programmer when a change to one function is inconsistent with the units of variables in another one. Such a most-general unit system also provides a more efficient way for a developer to add human-readable units to a program: he or she must only specify a few of the unit types in a program, and the rest can be automatically assigned to be consistent.

2 The Units Inference Problem

This section provides some additional details about the problem of units inference as we approach it, and defines terminology that will be used in the rest of the paper.

A first important distinction is between the concepts of dimension and unit. A *dimension* is an attribute that can be measured: for instance, length or mass. A *unit* is a standard amount of some dimension, in which other measurements of that dimension can be expressed: for instance, inches or kilograms. Every physical quantity has an associated unit and dimension: the dimension is determined by the unit, but there can be multiple units that correspond to the same dimension. For instance, meters per second and miles per hour both measure speed. Dimensions and units obey the same algebraic rules: for instance, the product of two values must have the product of their dimensions, and also the product of their units. Because of this similarity, the techniques we discuss are generally applicable to checking either the dimensions or the units in a program. For instance, in the statement `len1 = 2.54 * len2`, one might either assign `len1` and `len2` the dimension length, so that `2.54` is dimensionless, or one might assign `len1` the units of centimeters, `len2` the units of inches, and `2.54` the units centimeters per inch. However, because a program with the correct units will necessarily have the correct dimensions, we will generally speak simply of "units". As an exception, we refer to quantities with no units as "dimensionless" rather than "unitless", since the former term is standard.

Units inference is the problem of assigning a *unit type* (or just a "unit" for short) to all of the *program variables* in a piece of code such that the algebraic rules of units are satisfied. Among the program variables we count both true variables and numeric literals. Because unit correctness applies to operations between variables, the units for a single variable cannot be defined in isolation: rather one should consider a *unit system* in which all of a program's units are expressed. A unit system is distinguished by its set of *basic units*, those in which every other unit can be expressed. For instance, in the standard SI (metric system), the basic units include the kilogram, second, meter, and ampere, and the *derived unit* volt can be expressed as $\mathsf{kg} \cdot \mathsf{m}^2/(\mathsf{A} \cdot \mathsf{s}^3)$.

In considering different unit systems for a single program, it can be helpful to partially order them according to a relation we call subsumption. A unit system S_2 subsumes a unit system S_1, written $S_1 \sqsubseteq S_2$, if every basic unit of S_1 can be expressed in terms of basic units of S_2. Intuitively, S_2 is 'more expressive' than S_1. As an extreme case, the unit system with no basic units (in which every quantity is dimensionless) is subsumed by any unit system. If two unit systems are each subsumed by the other, we say they are equivalent.

3 Annotation-less Unit Type Inference

We present a technique that first automatically infers unit types from the source code of C programs and then allows a user to annotate program variables with named units. The goal of our annotation-less unit type inference technique is to construct a unit system for a program which is general enough to cover whatever consistent system the programmer might have intended. To be precise, the goal is to find a unit system with the minimum number of basic units that subsumes any correct unit system for a program. There may be many equivalent unit systems with this property: the choice between them is arbitrary.

Our tool builds up constraints over variables and units based on a static source code analysis and solves the constraints to find the minimum number of *inferred basic units* of a unit system that subsumes the correct unit system. After constraint solving, each variable's units are expressed in terms of the inferred basic units. The user can then interactively annotate variables with *user-defined units* (whose components are *user-defined basic units*) to specify a subsumed (more specific) unit system.

As a running example, consider the following program that calculates the total kinetic and potential energy of a moving mass:

```
int main() {
    double mass, velocity, height, kinetic, potential;
    /* ... initialize relevant variables ... */

    kinetic = 0.5 * mass * velocity * velocity;
    potential = mass * height * 9.8;

    printf("Total energy is: %g J\n", kinetic + potential);
    return 0;
}
```

Our tool first generates constraints over the units of program variables based on operations between them such as addition and multiplication (Section 4.1). For instance, one constraint over the units of the variables `potential`, `mass`, `height`, and the constant `9.8` is that the units of `potential` must be the product of the units of `mass`, `height`, and `9.8`. We then simplify the constraints (Section 4.2) in various ways, including merging together variables that must have the same units into one set (e.g., `kinetic` and `potential`, which must have the same units because they are added together within the `printf` statement). We then solve the constraints (Section 4.3) to determine the minimum number of basic units (4 in this example) and each variable's units expressed in terms of those inferred basic units. The basic units have no names, so we will refer to them by the numbers 1-4. Here are the results from solving the constraints on our example:

```
Variables:
  1: velocity                          Units: (unit 1)
  2: mass                              Units: (unit 2)
  3: constant 0.5                      Units: (unit 3)
  4: constant 9.8                      Units: (unit 4)
  5: height                            Units: (unit 1)^2 * (unit 3) * (unit 4)^-1
  6: kinetic, potential                Units: (unit 1)^2 * (unit 2) * (unit 3)
```

This solution provides the most general set of units that are consistent with the constraints imposed by operations within the program. However, these units will likely be meaningless to the user, whose notion of units includes physics terms such as meter and second, not the unnamed basic units presented above. Our tool includes an interactive user interface that allows the user to assign named user-defined units to variables (Section 4.4). For this example, the user can annotate `velocity` with the units meter/second, `height` with the units meter, and so forth. Our tool combines the named units that the user provides with the inferred basic units from the constraint solver to assist the user in two main ways: First, it can automatically fill in the units of certain variables after the user enters in units for other ones, thus lessening the user's annotation burden. Second, such inferred units can alert the user to annotation mistakes or program bugs when the units of a variable do not match expectations.

```
Variables:
  1: velocity                          Units: meter second^-1
  2: mass                              Units: kilogram
  3: constant 0.5                      Units: dimensionless
  4: constant 9.8                      Units: meter second^-2
  5: height                            Units: meter
  6: kinetic, potential                Units: kilogram meter^2 second^-2
```

The above snapshot shows a completed session of our tool on this example. The user only needs to annotate 4 out of the 6 variables (because there are 4 inferred basic units) and the tool infers the units for the rest of the variables.

4 Technique and Implementation

Here are the four stages of our inference system:

1. **Constraint generation** - Analyzes the program's source code and produces constraints that must hold between the units of different variables (Section 4.1).
2. **Constraint simplification** - Simplifies the constraints by applying both meaning-preserving transformations and heuristics which lose information but make sense in practice (Section 4.2).
3. **Constraint solving** - Solves constraints using linear algebra techniques and outputs a minimal set of inferred basic units (Section 4.3).
4. **User interface for guided annotations** - Allows the user to provide user-defined units for variables interactively while being guided by the solved constraints (Section 4.4).

4.1 Constraint generation

The input to this stage is C source code, and the output is a set of constraints on the unit types of variables. This stage is implemented as OCaml code which is compiled with the CIL analysis framework [NMRW02]. CIL pre-processes and parses C code to produce an abstract syntax tree; our code then walks through each expression in the program, outputting constraints along the way. A fresh unit type variable is created for each variable and numeric literal in the original program, and for each subexpression. (In the rest of this paper, we will often use the term "variable" as an abbreviation for "unit type variable".) Then, certain kinds of expressions produce constraints between the type variables. Our tool operates on unit exponents, so that unit multiplication is represented by addition and unit exponentiation by scalar multiplication; this allows us to use the usual terminology of linear algebra. In the rules that follow, we use x_e to represent the unit type for an expression e:

- Addition (`+`), subtraction (`-`), assignment (`=`) and comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`) operators give equality constraints between unit variables. For instance, from `a = b + c`, the tool generates the constraints $x_{\texttt{b}} = x_{\texttt{c}}$ and $x_{\texttt{a}} = x_{\texttt{b}}$.
- A multiplication (`*`) operator gives a sum constraint between unit variables. For instance, from the code `a = b * c`, the tool generates the constraint $x_{\texttt{a}} = x_{\texttt{b}} + x_{\texttt{c}}$.
- A division (`/`) operator gives a difference constraint between unit variables. For instance, from the code `a = b / c`, the tool generates the constraint $x_{\texttt{a}} = x_{\texttt{b}} - x_{\texttt{c}}$.
- The square-root function `sqrt` gives a sum constraint according to the multiplicative relationship $\sqrt{x}\sqrt{x} = x$. For instance, from `a = sqrt(b)`, the tool generates the constraint $x_{\texttt{b}} = x_{\texttt{a}} + x_{\texttt{a}}$. (This constraint is equivalent to the more obvious $x_a = \frac{1}{2}x_b$, but has the advantage of not requiring extra constraint syntax beyond that required for multiplication above. We have also considered giving a special treatment to the `pow` function when its second argument is a constant, but the need did not arise in our case studies.)

This constraint collection can be viewed as a very simple static analysis. It is flow-insensitive, because the desired result of our tool is a single unit type for each program variable. Our analysis is currently also context-insensitive. For most functions, a context-insensitive analysis is sufficient, but some functions are used with arguments of more than one unit type. Such functions are most common in the standard library:

for instance, generic math functions like `pow`, memory allocators like `malloc`, and input-output routines like `printf`. Since such functions are not defined in the source code available to our tool, it generates no constraints for them. However, if such a *unit polymorphic* function is defined, the results of our tool suffer: usually the only consistent typing makes the arguments to such a function dimensionless. In considering examples, we have found cases where a number of other kinds of sensitivity might improve our tool's results: field sensitivity (giving different units to different fields of a structure), distinguishing different definitions of a single variable (as by SSA conversion), and giving different unit types to different elements of statically-sized arrays. However, it is unclear whether the benefit of such changes would be worth the effort in both tool implementation and analysis runtime, and many would also require adding a cast-like mechanism to the target unit annotation system.

Our implementation takes a very conservative approach to pointers and aliasing: all of the values accessed via a pointer must have the same units. A natural implementation of this approach would be to create a family of pointer types for each unit type: for instance, besides a type for "meters", there would also be a type "pointer to meters", another "pointer to pointer to meters", and so on. However, because it does not seem possible to represent pointer types in linear algebra, our analysis takes a simpler approach of not recording the information about which level of pointer a value contains: the pointer generated by taking a reference to a meters value is again simply labelled "meters". We believe this is reasonable because the program has already been checked under the C type system, which distinguishes levels of pointers. A confusing result might still be produced if a program, say, casts a pointer to an integer value and then multiplies this integer by another dimensioned value, but such situations seem unlikely in practice.

Our analysis is interprocedural: it simply creates constraints requiring that the arguments to a function have the same unit types as the function's formal parameters, and similarly for the function's return value. To conservatively estimate which functions might be called by code that uses a function pointer, we use a pointer analysis that is supplied with CIL. Though few of the programs we have considered make extensive use of function pointers, this analysis might also be a source of imprecision for programs that do.

The following figure shows the relevant constraints produced in the energy example, to the right of the lines of code that produce them. Note the unnamed intermediate variables `x11`, `x12`, `x13`, `x17`, and `x18`, which correspond to non-atomic expressions: for instance, `x11` is the unit type of the expression `0.5 * mass`.

```
int main() {
    double mass, velocity, height, kinetic, potential;
    /* ... initialize relevant variables ... */

    kinetic = 0.5 * mass * velocity * velocity;             // variable velocity  is x8
                                                            // variable mass      is x9
                                                            // constant 0.5       is x10
                                                            // x11 = x10 + x9
                                                            // x12 = x11 + x8
                                                            // x13 = x12 + x8
                                                            // variable kinetic   is x14
                                                            // x14 = x13
    potential = mass * height * 9.8;                        // constant 9.8       is x15
                                                            // variable height    is x16
                                                            // x17 = x9 + x16
                                                            // x18 = x17 + x15
                                                            // variable potential is x19
                                                            // x19 = x18
    printf("Total energy is: %g J\n", kinetic + potential); // x14 = x19
    return 0;
}
```

4.2 Constraint simplification

It would be perfectly correct to take the constraints directly as extracted from the program, convert them into matrix form, and solve them as we will describe in Section 4.3. However, the linear algebra constraint solving technique requires time and space that grow more than linearly in the number of unit type variables, and are significant in practice even for relatively small programs. Our constraint generation stage usually produces on the order of as many constraints and unit type variables as there are lines of source code, and the matrices used in the constraint solving stage will have at least as many rows and columns as there are constraints and variables, respectively. Thus, it is important in practice to simplify the unit constraints before solving them.

To allow the most freedom in the implementation of the constraint generation stage, constraint simplification is performed by a separate program (a Perl script). The simplification has three parts:

- First, the constraint equations are simplified algebraically, removing intermediate variables by replacing the variable from the left hand side of a constraint, wherever it appears, with the right-hand side of the constraint, and then removing the constraint. In the example, the constraints `x11 = x10 + x9` and `x12 = x11 + x8` can be replaced by a single constraint `x12 = x10 + x9 + x8`.

- Second, program variables and constants whose type variables are transitively constrained to be equal are merged into a single set; they are then considered together for the rest of the analysis. In the example, the type variables `x14`, `x18`, and `x19`, which correspond to the program variable `kinetic`, a sub-expression, and the program variable `potential`, can be grouped together in this way.

- Finally, the type variables are partitioned into sets whose unit types can be inferred independently: i.e., if one considers a graph in which type variables are nodes and edges connect variables that appear together in a constraint, each connected component of the graph is completely independent. Each subset of this partition can then be processed separately for the rest of the analysis. (It is often the case, including in our running example, that all of the variables with meaningful physical units lie in one connected component.)

In addition to the meaning-preserving simplifications described above, it is also helpful to perform some heuristic simplifications with respect to the type variables for constants. In theory, these changes might rule out some otherwise legal unit typings, but in practice we can be relatively confident that they will not rule out any desired ones, and they make later stages of inference easier for both the tool and the programmer who must eventually annotate variables with units. Our technique's default treatment of numeric constants is that each occurrence of a constant might have a different unit type, and this is clearly the best behavior for some values; for instance, one occurrence of 0 might represent the net weight of an empty bucket, while another might be the velocity of a stationary bicycle. On the other hand, floating point constants with a precise but random (in the Chaitin-Kolmogorov sense) value are likely to have only one unit type in all the places where they occur. For instance, 745.7 and $2.99792 \cdot 10^8$ are likely always the conversion factor between horsepower and watts, and the speed of light in meters per second, respectively. To avoid introducing superfluous degrees of freedom (leading to additional inferred basic units, and therefore to a need for redundant programmer annotations), our simplification stage gives all occurrences of such a value the same unit type. The only unit typings that would be ruled out by such a modification are those in which the correct units for two occurrences of, say, 745.7 were different. This merging is particularly significant when a constant is specified in the original program with `#define`; because CIL operates after the C preprocessor, our tool would not otherwise be able to link the occurrences of the constant. Similarly, certain well-known values (such as 2.71828) can be reasonably presumed to be dimensionless.

In our running example, no merging of constants occurs because no constant appears more than once, but our tool's heuristics would merge additional occurrences of 9.8 if they appeared elsewhere. Though 0.5 is dimensionless in this case, we were not confident that every occurrence of that value would be, so our heuristics do not constrain its type variable. Algebraic simplification reduces the constraints to two:

```
x19 = 2*x8 + x9  + x10
x19 =   x9 + x15 + x16
```

In this example, the two constraints produced by simplification are linearly independent, but in general the simplification algorithm is not powerful enough to remove all redundant constraints.

4.3 Constraint solving

This stage solves a set of constraints produced by the constraint generation and simplification stages to find the minimum number of inferred basic units such that any consistent unit system can be expressed in terms of those basic units. It is implemented as a MATLAB script that accepts the constraints in a matrix, uses linear algebra to solve the matrix, and outputs a solution matrix that contains the basic units and the exponent power of each basic unit assigned to each unit type variable.

We will demonstrate the constraint solving algorithm using our energy example. Here are the constraints in matrix form after simplification:

```
Constraints:      x19 = 2*x8 + x9  + x10
                  x19 =   x9 + x15 + x16

Matrix:           #  x8   x9  x10  x15  x16  x19
                  [   2    1    1    0    0   -1
                      0    1    0    1    1   -1 ]

Names:             x8: {velocity}        x9: {mass}       x10: {constant 0.5}
                  x15: {constant 9.8}   x16: {height}     x19: {kinetic, potential}
```

Observe that the constraints will always be homogeneous (i.e., there will never be any constant terms), so they can be represented in a single matrix by putting all the variables in each equation on one side and placing their coefficients into the matrix (the other side of the equation is zero). There is one row for each constraint and one column for each unit type variable. The `Names` array provides the names of the program variables that are associated with each unit type variable (notice that, due to constraint simplification, two program variables, `kinetic` and `potential`, are associated with the 6th unit type variable, `x19`).

We want to find the maximum number of basic units that can be assigned so that all the constraints are satisfied, to give the most general unit system, subject to the constraint that the basic units are *independent*: none can be expressed in terms of the others. (Recall that our tool can only find *inferred basic units*, not the basic units of a real-world units system such as SI. However, we will use the term basic units for brevity.) That number is equal to the number of columns in the matrix minus the *rank* of the matrix. The number of columns represents the number of unit type variables, and the rank represents the number of independent constraints.

To get some intuition for this formula (num. basic units = num. columns − rank), suppose that there were no constraints (rank = 0). Then the number of basic units equals the number of variables, because the most general system of units allows each variable to have its own unique basic unit. Adding a constraint allows the units of one variable to be expressed in terms of the units of all the other variables, thus reducing the number of required basic units by one. Every subsequent independent constraint added reduces the number of basic units by one. If the number of independent constraints equals the number of variables, no basic units are required and the only valid solution is to assign all variables the special null unit dimensionless (e.g., in a one-line program consisting of the statement $a = a^2$, the only solution is that a is dimensionless). The number of independent constraints cannot exceed the number of variables (a theorem from linear algebra). In practice, the rank of the constraint matrix is usually slightly less than the number of rows, because the simplification process of Section 4.2 removes many but not all redundant constraints.

We have developed a greedy constraint solving algorithm that attempts to pick basic units from among the unit type variables:

- Place constraints in matrix M
- $j = 1$
- for $i = 1$ to num. variables:
    1. Save M, then augment it with a new column for the j^{th} basic unit and a new row for the assignment of the i^{th} unit type variable to the j^{th} basic unit
    2. Transform the matrix to express all variables in terms of the j basic units and check for linear dependence among the basic units (can some be expressed purely in terms of the others?)
    3. If so, undo assignment by restoring M
    4. Else, j++ for successful assignment
    5. If (j == num. basic units), break

The algorithm iterates through the list of variables and tries to assign each of them to a new basic unit. Each assignment is made by augmenting the matrix by one column to represent the newly-added basic unit and one row to represent the newly-added constraint that a particular variable is assigned to that basic unit. If the assignment is successful, then it moves on to assign the next variable to a new basic unit. Otherwise, it tries to assign the next variable to the current basic unit. The algorithm terminates when the necessary number of basic units (as previously computed) have been assigned to variables.

The indication of a successful assignment is that no linear dependencies have been introduced among the basic units. A linear dependence occurs when an assignment is made such that a basic unit can be expressed purely in terms of the other basic units. For example, let's suppose that we have assigned variable 1 to basic unit 1 and variable 2 to basic unit 2. Let's also suppose that there is a constraint stating that variable 3 has the units of variable 1 raised to the 4^{th} power. Thus, when we attempt to assign variable 3 to basic unit 3, this creates a linear dependence because basic unit 3 can be expressed purely in terms of basic unit 1: (basic unit 3) $= 4 \cdot$ (basic unit 1). Intuitively, this assignment did not add any extra information, so it is marked as unsuccessful. The algorithm now attempts to assign variable 4 to basic unit 3. We check for linear dependence because basic units that were not independent would be redundant, creating extra equivalent expressions for the same unit type without increasing expressiveness.

The constraint solving algorithm is implemented as a MATLAB script that roughly follows the pseudo-code presented above. It calls the MATLAB `rref` function to put the matrix in reduced row-echelon form, which makes it efficient to check for linear dependence during each assignment. The check for linear dependence is made by using the `rank` function to find the rank of a slice of the matrix that corresponds to the basic units assigned so far expressed in terms of the variables. If the rank of the slice is less than the number of basic units assigned so far, then there is a linear dependence among the expressions of the basic units in terms of program variables, so the same dependence holds between those basic units.

Here are the results of the constraint solver on our energy example:

```
Variables:
  1: velocity                          Units: (unit 1)
  2: mass                              Units: (unit 2)
  3: constant 0.5                      Units: (unit 3)
  4: constant 9.8                      Units: (unit 4)
  5: height                            Units: (unit 1)^2 * (unit 3) * (unit 4)^-1
  6: kinetic, potential                Units: (unit 1)^2 * (unit 2) * (unit 3)
```

All assignments were successful, so the first 4 variables were assigned to the 4 basic units. The solver also calculates the units of the remaining variables in terms of the basic units so that all variables are assigned units.

4.4 User interface for guided annotations

This final stage of our tool allows the user to interactively assign user-defined units to program variables. The input is a solution matrix from the constraint solving stage and a list of variable names. The user is allowed to assign user-defined units for any program variable in any order, except that no assignment can violate the inferred constraints. A *user-defined unit* consists of 0 or more *user-defined basic units* (0 only for the special unit dimensionless). For example, the user-defined unit $\mathsf{meter} \cdot \mathsf{second}^{-1}$ consists of 2 user-defined basic units, meter and second, which belong to the SI unit system. These basic units differ from the tool's inferred basic units in that they have names; they form a unit system that is subsumed by the one formed by the inferred basic units.

Each time the user assigns a user-defined unit to a variable, the tool adds an additional constraint to equate the inferred basic units for that variable with the user-defined basic units that comprise that unit (adding additional columns for user-defined basic units if necessary). For example, if the user assigns the units of $\mathsf{meter} \cdot \mathsf{second}^{-1}$ to `velocity`, the tool adds a new constraint that equates `basic unit 1` with $\mathsf{meter} \cdot \mathsf{second}^{-1}$. This is very similar to what happens in the constraint solving stage. The tool then solves the augmented matrix, and presents the user with an updated list of variables and their user-defined units. Solving the matrix after every assignment allows the tool to automatically infer user-defined units for variables that the user did not manually annotate. The user only has to make a number of annotations equal to the number of inferred basic units, which is often far less than the total number of variables. For example, after assigning units for `velocity`, `constant 0.5`, and `height`, the tool inferred the units of `constant 9.8` without the user having to annotate it:

```
Variables:
  1: velocity                          Units: meter second^-1
  2: mass                              [Units not yet established]
  3: constant 0.5                      Units: dimensionless
  4: constant 9.8                      Units: meter second^-2
  5: height                            Units: meter
  6: kinetic, potential                [Units not yet established]
```

This feature is useful in two ways: First, it alleviates the burden of making the user annotate additional variables, especially ones that have complex units consisting of many components. Second, it provides the user with confidence that the units that he or she has been assigning thus far are correct. Conversely, it can alert a user either to bugs in the program or to errors in the annotation if the inferred units do not match expectations.

When this stage begins executing, all variables are expressed in terms of the inferred basic units provided by the constraint solving stage. As the user annotates variables with user-defined units, the tool tries to automatically annotate other variables with user-defined basic units in a way that is consistent with the inferred basic units. Often a variable is expressed as a mixture of inferred basic units and user-defined basic units. As soon as a variable can be expressed solely in terms of user-defined basic units, then the inference is complete for that variable, and it is reported to the user. This stage completes when the inference is completed for all variables, so that all can be expressed solely in terms of user-defined basic units. At this point, the transformation from inferred basic units to user-defined basic units is complete, and there is no more need for the inferred basic units.

5 Experimental Results

We ran our tool on C programs ranging from 50 to 50,000 lines of code and performed quantitative and qualitative evaluations. We found most test programs by performing a web search for programs that performed calculations with physics units (using names for SI units and the word `printf` in our search string):

Program	LOC	Time (sec.) generate	simplify	solve	Number of constraints	Program variables	Simplified var. sets	Basic units
`energia`	46	0.31	0.04	0.13	76 / 10	33	10	4
`quasi`	491	0.51	0.16	0.43	579 / 40	219	29	3
`schd`	633	0.55	0.27	5.95	1041 / 85	452	94	34
`bike-power`	680	0.39	0.14	5.09	544 / 76	273	72	29
`demunck`	1005	0.84	0.36	8.91	2099 / 139	923	102	22
`starexp`	4974	0.87	1.35	100.31	7995 / 550	3517	543	197
`oggenc`	58413	51.00	3.28	168.93	27864 / 777	11045	502	163

Table 1: Results gathered from running all the stages of our units inference tool except for the interactive user interface. For the number of constraints, the value to the left of the slash is the number before constraint simplification and the value to the right is the number after simplification.

- `energia` - This small program computes the kinetic and potential energies of a mass launched from a specified height. It is in fact quite similar to our running energy example, except that it comes from a physics class at the University of Rome (La Sapienza), so all of the identifiers are in Italian.
- `quasi` - This program, originally developed for a class assignment and now maintained in the public domain by Eric Weeks, draws a Penrose tiling of a portion of the plane (i.e., a quasicrystal) as a Postscript file.
- `schd` - This program by Mike Frank of the University of Florida reversibly simulates the evolution of a particle wavefunction according to the Schrödinger equation, and displays the results in an X window.
- `bike-power` - This program from Ken Roberts of Columbia University computes a table of the energy used when riding a bicycle at various speeds, taking into account a number of other factors including the weight of bicycle and rider, the grade of the slope, and the coefficient of friction between the tires and the road.
- `demunck` - This program is a simulation of the propagation of electrical impulses through a series of concentric spheres of various conductivities, used to approximate the processes that distort the readings of an electroencephalogram (EEG).
- `starexp` - This is a relatively large program by James A. Green which computes a number of characteristics of main sequence stars based on their brightness and spectral type, using models from astrophysics.
- `oggenc` - This, the largest of our test programs, was not found by a web search as the others were; it is a common Linux multimedia utility that we have worked with in previous research. It applies a lossy psycho-acoustical algorithm to compress audio data into an MP3-like format known as Ogg Vorbis. Unlike general purpose compression, this encoding relies on extensive signal processing. Though we at first used it just to test our tool's scalability, reading the code revealed a number of physical units, including ones for sampling rates and frequencies, elapsed times, and sound intensities.

5.1 Quantitative Evaluation

Table 1 demonstrates that our tool scales well to moderate-sized programs. The bulk of the running time is in the MATLAB constraint solving script, mostly in two calls to the `rref` function. We have optimized the implementation so that `rref` only needs to be called twice regardless of the number of variables or constraints (our initial implementation called `rref` once for every iteration of the loop, which was much slower). The solver's run time is proportional to the number of constraints and the square of the number of variables, or

roughly cubic in the size of the original program. By contrast, the constraint simplification process requires only slightly more than linearly many steps in terms of the number of constraints, so spending time on simplification saves time overall. In fact, the constraint simplifier is crucial for the scalability of our tool because it greatly reduces the size of the constraint matrix, usually by several orders of magnitude.

The number of inferred basic units was always much less than the number of program variables, which greatly lessens the annotation burden on the user during the user interface stage. Without our tool, the user needs to annotate every program variable with units, but using our tool, the user only needs to make a number of annotations equal to the number of basic units.

For `schd` and `bike-power`, we removed the definitions of some functions used for memory allocation or command-line input which were unit polymorphic (see Section 5.2 for discussion), and for `oggenc`, we used a previously prepared version that had been modified to compile as a single file; the other programs were unmodified. The constraint generation stage for `oggenc` took an especially long time because of the pointer analysis that CIL performs to resolve function pointer targets, which here uses the default settings for precision. Most of the time may be saved by supplying options to perform the analysis in a less precise manner; we have not evaluated whether doing so affects the overall results of our tool.

5.2 Qualitative Evaluation

To assess the usability and precision of our tool, we have performed qualitative evaluations on our test programs by using our user interface to annotate program variables with units. When performing these evaluations, we simulated the behavior of someone who is familiar with the code and tried to annotate variables with simpler units before ones with more complex units.

We performed a fairly detailed annotation session for `bike-power`. We determined the correct units of variables by inspecting the source code and comments, and we confirmed that the tool's results matched those units. Although there were 29 basic units in this program, we only had to make 12 annotations before all of the interesting variables (48 out of 72) were correctly labeled with units. The remaining 17 annotations were required to mark identical copies of a conversion factor constant defined as a C preprocessor `#define` (CIL works on source code after preprocessing). From our experiences with this program and several others (e.g., `schd`, `demunck`), we believe that it is usually possible for the user to make fewer annotations than the required number (number of basic units) and still have the tool infer units for most of the interesting program variables.

We began our interactive session for `bike-power` by annotating variables that seemed to have easy-to-specify units such as dimensionless, kilogram, pound, and $\mathsf{meter} \cdot \mathsf{second}^{-1}$ (velocity). Sometimes after we annotated a variable, our tool was able to infer that several other variables had the exact same units. For example, after we specified that a particular constant represented a velocity, the tool reported that several other constants were also velocities, which we confirmed to be correct. This helps reduce the number of variables that we had to annotate (recall that both constants and variables must be annotated since constants may have units as well). Although this is useful, the tool's inference functionality is more powerful when it can infer units that are different than the units that we have entered thus far. For example, after annotating several variables, the tool reported that the units of a variable `P_t` was $\mathsf{meter}^2 \cdot \mathsf{second}^{-3} \cdot \mathsf{kilogram}$, which happens to be watts expressed in SI basic units. This saves the user a lot of work because it is difficult to remember such complex units and can be error-prone to enter them in manually.

In the process of annotating units for `bike-power`, we discovered the following bug in the comments of its source code:

```
#define mOs_per_miOhr 0.44704                /* meters/second per kilometers/hour */
#define mOs_per_kmOhr (1000.0 / 3600.0)      /* meters/second per miles/hour */
```

The comments next to these two conversion factors have been mistakenly switched. Because we relied on the comments to determine the set of correct units for the program, we entered in incorrect units for `mOs_per_kmOhr`. The tool has no way of informing the user of when an annotation is incorrect. However,

we suspected that there was a bug when we saw the tool inferring units for other variables that did not match our expectations. We were able to quickly track the source of the problem back to the source code comments.

The experience of using the tool is similar for all of the programs, and for the larger programs we did not assign units to every variable, so we discuss the remaining programs in less detail. In the `energia` example, we easily assigned units to each program variable. For the `quasi` program, we were able to assign the units of centimeters to a number of variables that measured length, the most important dimension in the program. In `schd` and `bike-power`, our initial results were unsatisfactory in that variables we expected to have a dimensioned type were inferred to be dimensionless. This failure was caused by the context-insensitivity of our analysis: for instance, in `bike-power`, a single function was used to parse numeric command line arguments in various units. Because our analysis can currently assign only one set of units to the result of a function, the only consistent solution was for the function to return a dimensionless value, which propagated through much of the rest of the program. To work around this limitation, we removed the definitions of these functions from the source code, causing our tool to generate no constraints from their use. After this change, the results for both programs were satisfactory: `bike-power` is discussed above, and for `schd` we were able to assign around a dozen units consistently to various variables. In addition, our tool automatically inferred the correct units for Coulomb's constant of electrostatic attraction, which would have been cumbersome to compute by hand.

For the larger programs, unfortunately, our attempts to assign units revealed imprecisions of our implementation that we could not easily work around. In `demunck`, a calculation which in the original paper describing the technique was performed using a matrix was coded with a 2-by-2 multidimensional array; unfortunately, the different entries in the matrix had different units, while our analysis assumes that all the elements of an array have the same units. In `starexp`, a single global variable named `d` is used repeatedly (in code that appears to have been cut and pasted) when reading floating-point values of a number of different units, causing them all to be considered dimensionless. For `oggenc`, we attempted to assign the units of hertz (1/sec) to the numeric literal 44100, which represents the sampling rate of CD-quality audio. Unfortunately, because it is field insensitive, our tool's result is that this constant must be dimensionless: a single structure is used to include a number of parameters to the encoding library, including both the sampling rate and some quantities measured in seconds, but our analysis can assign only one unit to all the elements of the structure.

6 Related work

Scientists and engineers have long recognized that checking whether the results of a calculation have the expected units is an effective practice to recognize and prevent errors, and many domain-specific systems incorporate unit information in measurements. It should not be surprising, therefore, that adding unit information to programming languages has been the topic of frequent research. Some languages include sufficiently powerful extension mechanisms that systems for unit type information can be added within the language: an example is the combination of templates and operator overloading in C++ [Bro01]. However, a more common approach has been to consider language extensions, which can be suggested for any language, and allow the designer freedom in the choice of syntax. It is natural to see unit information as a kind of type, so many approaches have extended the type systems of then-common strongly-typed general purpose languages, such as Pascal in the 1980s [Bal87] or more recently Java [vD99]. An alternative approach for a language like Java is to treat units as a kind of class [ACL+04]. Perhaps surprisingly, research applied to languages used specifically in the scientific community is in the minority; examples include Petty's unit extension for FORTRAN [Pet01], and a unit checking tool for the 'language' of Microsoft Excel spreadsheets [ASK+04].

To reduce the burden of adding unit annotations, several researchers have suggested embedding unit types in a type system like that of ML that supports type inference. The earliest work to draw a connection between unit types and ML-like typechecking was that of Wand and O'Keefe [WO91]; they suggest an inference technique using linear algebra much like that originally suggested by Karr and Loveman [KL78]. The inference our tool performs is similar, but made more complicated by the fact that our system must

infer a most general set of basic units; in previous systems, the set of basic units was fixed or user-supplied. A variant of the algebraic technique which allows only integral exponents is developed more extensively by Kennedy [Ken94, Ken96, Ken97]. He also gives theoretical results on the expressiveness of such a system: for instance it is impossible to implement the square-root function so as to have the correct polymorphic type.

Such previous inference techniques differ from the problem we consider in that they presume that the set of basic units, and the units for values such as constants, are specified by the programmer, just as in ML a program must declare data-types and their constructors. Assuming the program is correct, this information is enough to determine the unit type of each value in a program. By contrast, we consider programs with no unit information, for which a large number of valid unit typings exist. Without any annotations, our technique determines a most-general unit typing (one with the largest number of independent basic units) that is consistent with the program's operations. This most general typing is already suitable for many kinds of automated checking: for instance, any modification to a program that would violate a most-general unit typing would also violate any more specific typing. Of course, if the unit types of a program are to be related to standard physical units, the need for some programmer annotation is unavoidable. In previous systems, the programmer provides unit types at all of the syntactic locations required by the language, and the inference system then checks if those annotations are consistent. In our system, a consistent typing is automatically generated, and the programmer then assigns names to a subset of the generated types, from which the tool infers names of the rest.

An approach like ours that performs analysis first has two major advantages: first, if the program is correct, it allows the programmer to construct a complete unit typing with a minimal number of annotations, one per basic unit in the most general typing. In previous systems, the annotation burden is proportional to static size of the program, which is usually much larger. On the other hand, if the program's unit correctness is unknown, it is necessary for a programmer to examine each program variable and use their domain understanding to determine what units would be correct. The difference between our approach and previous ones is that for many such variables, a programmer using our system would simply verify the correctness of an automatically-produced unit type, when in previous systems they would write such a type themselves. Though conceptually the programmer's work is the same in these cases, the increased automation of our approach should make the programmer's work easier in most cases.

Because previous systems have required a significant amount of program annotation, they have rarely been evaluated on large or pre-existing programs. In particular, previous languages that have included unit type inference [WO91, Ken97] have been based on languages not commonly used by scientists or engineers, and have not been evaluated in realistic case-studies. Experimental results, like practical uses, have been focused on systems that require extensive annotation. A very promising, but small and uncontrolled, case study was performed by Brown [Bro01]: he had an independent programmer implement a simple physics function using SIunits, and the unit checking led to the discovery of three separate errors. Antoniu et al. used their spreadsheet-checking system [ASK+04] to find previously unknown bugs in three published spreadsheets, two in which results were labelled incorrectly and one involving an incorrect calculation. Because our system does not require annotations, it can naturally be applied to pre-existing programs, and we have already tested it on some such programs. Improving the efficiency of the tool to operate on large programs, and devising experiments to evaluate the precision of its results without manual comparisons, are directions we plan to investigate in the future.

7 Future Work

- **Add context sensitivity/unit polymorphism** - Much of our tool's imprecision on larger programs can be attributed to the fact that our static analysis is context-insensitive. For functions that accept values of different units, we now constrain all values passed into that function to have the same units. We currently have special cases for common functions such as square root (`sqrt`) but would like to add unit polymorphism to avoid having to rely on special cases.

- **Specialize for standard real-world units and constants** - Our tool currently infers the most general set of basic units without consideration of actual units that a user is likely to annotate (e.g., SI units). Connecting our interactive user interface to a database of units may improve the usability of our tool and allow it to provide more accurate results with even fewer user annotations.
- **Incremental constraint solving** - The current performance bottleneck of our tool is the MATLAB constraint solving script (especially the calls to `rref`). Instead of building up a large matrix of constraints and then simplifying and solving it, we want to find a way to solve the constraints as we collect them. This could greatly improve performance and might even make a dynamic unit inference implementation feasible.
- **Produce compiler-checked annotations in source** - We would like to annotate the program's source code with the units that our tool infers and have the compiler be able to perform type checking based on these units. This would make a program more resilient against the introduction of new bugs.

8 Conclusion

We have developed a technique and implemented a tool for inferring unit types from the un-annotated source code of C programs. Our tool allows the user to assign meaningful names to units of program variables with minimal effort. It can be applied to find bugs caused by inconsistent use of units, to prevent future bugs, and to provide documentation that can aid in code maintenance. We have evaluated our tool both quantitatively and qualitatively on a variety of real-world programs of up to 50,000 lines of code and reported its strengths and limitations (some of which form the motivation for our future work).

References

[ACL+04] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr. Object-oriented units of measurement. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 384–403, Vancouver, BC, Canada, October 26–28, 2004.

[ASK+04] Tudor Antoniu, Paul Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. Validating the unit correctness of spreadsheet programs. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 439–448, Edinburgh, Scotland, May 26–28, 2004.

[Bal87] Geoff Baldwin. Implementation of physical units. *ACM SIGPLAN Notices*, 22(8):45–50, August 1987.

[Bro01] Walter E. Brown. Applied template metaprogramming in SIunits: the library of unit-based computation. In *Proceedings of the Second Workshop on C++ Template Programming*, Tampa Bay, FL, USA, October 14, 2001.

[EJC01] Edward A. Euler, Steven D. Jolly, and H.H. 'Lad' Curtis. The failures of the Mars Climate Orbiter and Mars Polar Lander: A perspective from the people involved. In *24th Annual AAS Guidance and Control Conference*, Breckenridge, CO, USA, January 31–February 4 2001.

[Ken94] Andrew Kennedy. Dimension types. In *5th European Symposium on Programming*, Edinburgh, UK, April 11–13, 1994.

[Ken96] Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, April 1996.

[Ken97] Andrew J. Kennedy. Relational parametricity and units of measure. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 15–17, 1997.

[KL78] Michael Karr and David B. Loveman III. Incorporation of units into programming langauges. *Communications of the ACM*, 21(5):385–391, May 1978.

[Mar99] Mars Climate Orbiter Mishap Investigation Board. Phase I report, November 1999.

[NMRW02] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction: 11th International Conference, CC 2002*, Grenoble, France, April 8–12, 2002.

[Pet01] Grant W. Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience*, 31(11):1067–1076, September 2001.

[vD99] André van Delft. A Java extension with support for dimensions. *Software: Practice and Experience*, 29(7):605–616, June 1999.

[WO91] Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 479–483, 1991.